17 czerwca 2016

Michał Czupryniak
Michał Werda
Bartosz Kaczorowski

Podstawy teleinformatyki -
dokumentacja końcowa

Rozpoznawanie stanu gry w warcaby

1 Wstęp

Celem projektu była reprezentacja stanu gry w warcaby na ekranie komputera oraz sprawdzanie poprawności ruchów. Zakresem naszej pracy było przechwytywanie ustawienia pionków na planszy, wykrywanie ruchu oraz sprawdzenie poprawności ruchu.

Przy wyborze tematu kierowaliśmy się możliwością pracy nad ciekawym projektem, który w rezultacie da wymierne efekty wizualne. Dzięki realizacji tego projektu poznaliśmy wiodącą bibliotekę do ropoznawania obrazu OpenCV jak również nowy język programowania, Python.

2 Wykorzystywane narzędzia i biblioteki

Przy tworzeniu projektu używaliśmy:

- interpreter python w wersji 2.7
- IDE PyCharm

W projekcie wykorzystywalismy:

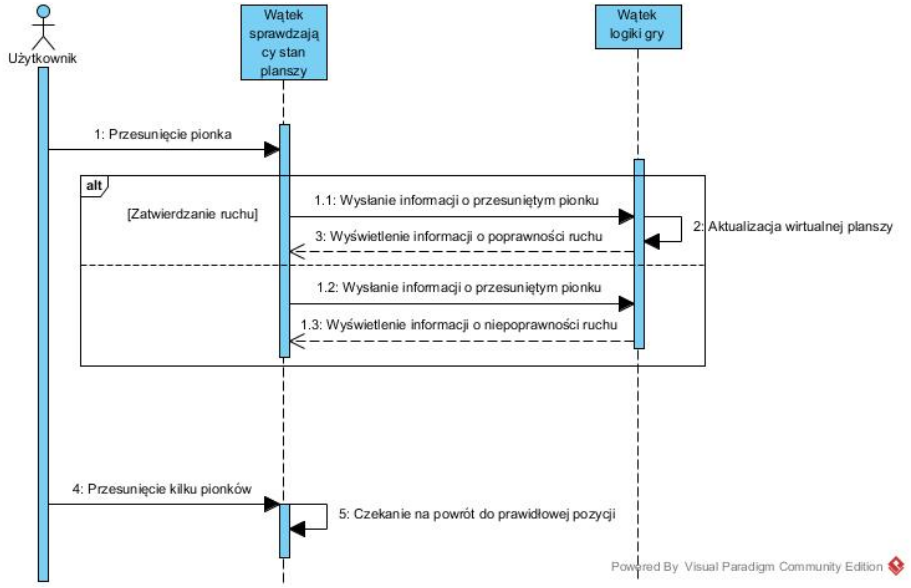
- biblioteka OpenCV
- biblioteka checkers 0.2

Wykorzystana kamera - Creative Live! Cam Sync HD 720p

Rysunek 1: Kamera internetowa marki Creative

3 Model systemu

Cała aplikacja oparta jest na dwóch wątkach. W wątku głównym cyklicznie jest sprawdzany stan planszy do gry, a w drugim wątku wyświetlana jest wirtualna plansza z ostatnim dobrym ustawieniem pionków na planszy. Wątek główny po wykryciu zmiany na planszy wysyła informacje jaki pionek z jakiego do jakiego pola się ruszył do wątku drugiego. Wątek drugi otrzymując taką informację ocenia ruch pod względem zgodności z zasadami i aktualizuje wirtualną planszę bądź wyświetla informację o niepoprawnym ruchu.

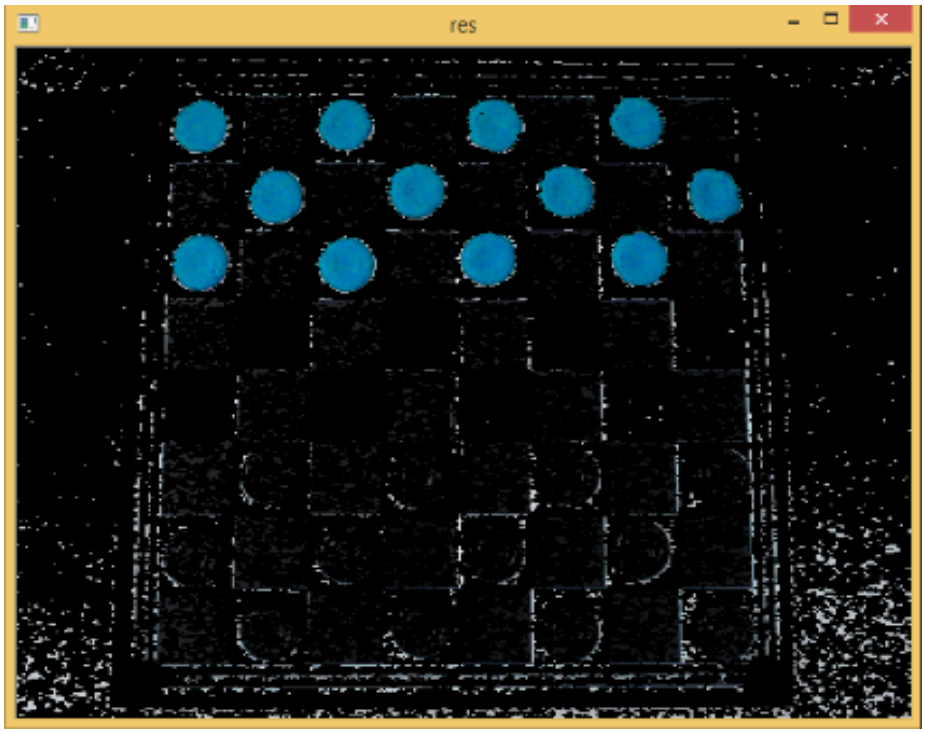


Rysunek 2: Diagram sekwencji działania programu.

4 Implementacja

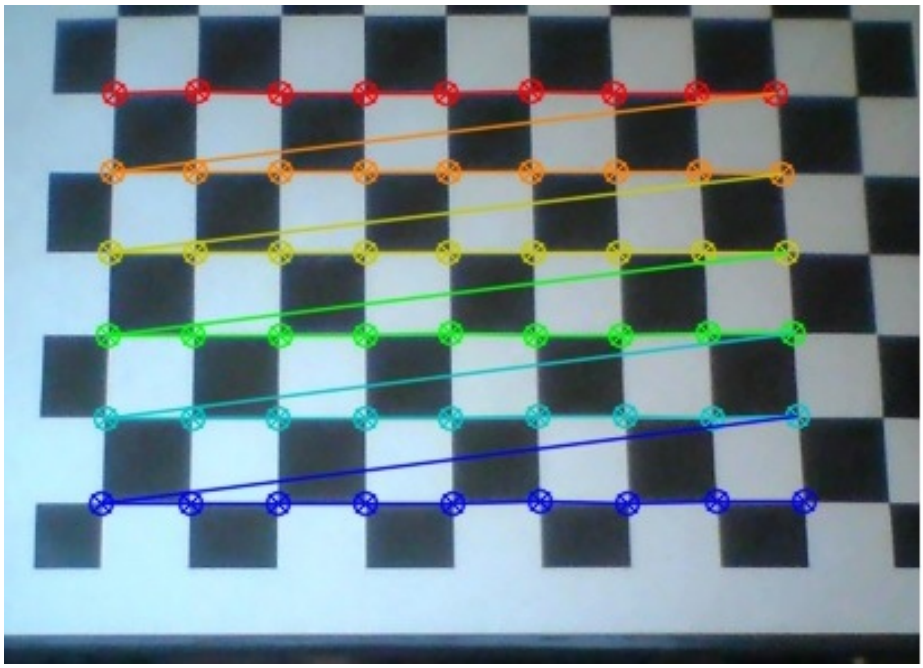
4.1 Znajdywanie położeń pionków na szachownicy

Podstawową funkcjonalnością w naszym projekcie było rozpoznawanie kształtów, obiektów przesyłanych przez kamerę. Pierwszą rzeczą, którą udało nam się odpowiednio rozpoznawać były kolory.



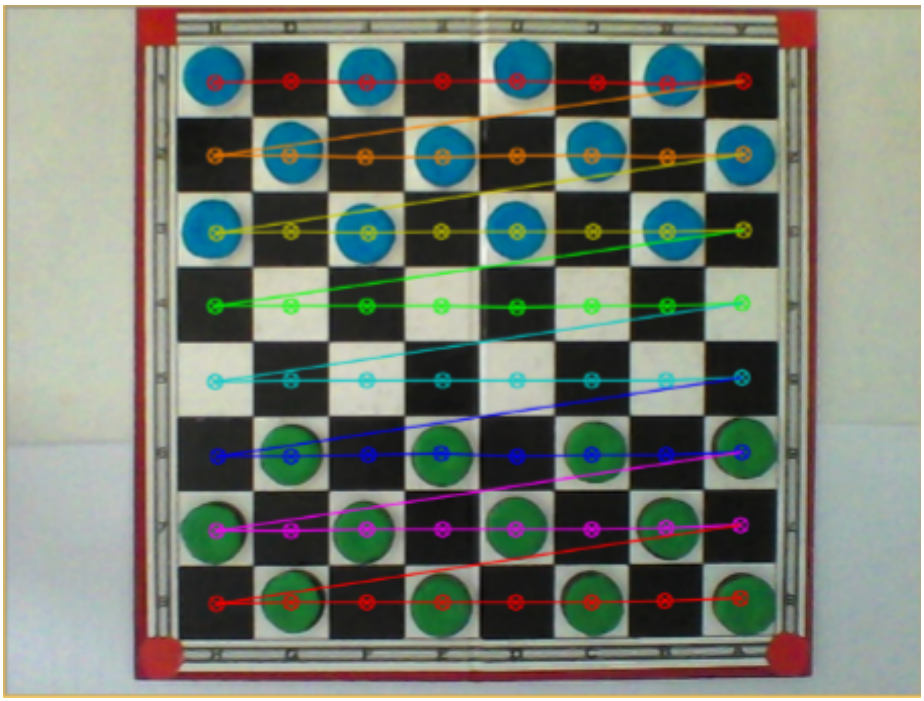
Rysunek 3: Detekcja niebieskiego koloru na szachownicy.

Kolejnym krokiem była detekcja narożników wewnątrz szachownicy. Wykorzystaliśmy do tego funkcję z biblioteki OpenCv *findChessboardCorners*. Funkcja zwracała czy określona ilość narożników (w szachownicy o wymiarach 8x8 znaleziono 49) oraz macierz zawierającą położenia narożników.



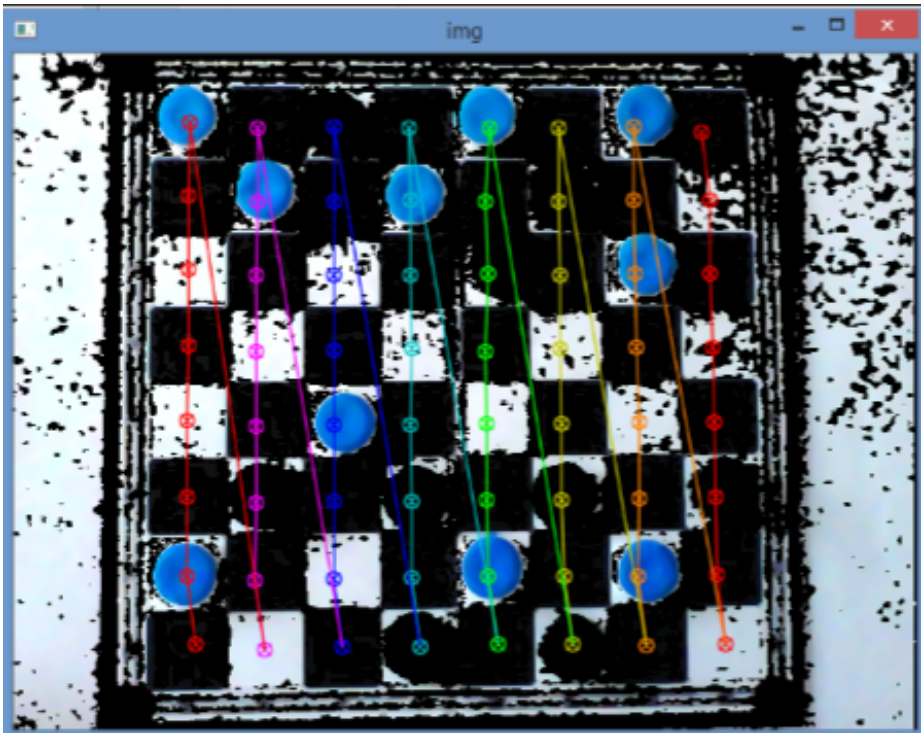
Rysunek 4: Detekcja krawędzi pól na szachownicy

Mając wyznaczone położenia narożników możliwe zostało wyliczenie dodatkowych położeń, aby macierz z wyznaczonymi punktami liczyła 64, a nie 49. Niezbędną operacją było również przesunięcie wektorowe wszystkich punktów tak, aby znajdowały się na środku pól szachownicy.



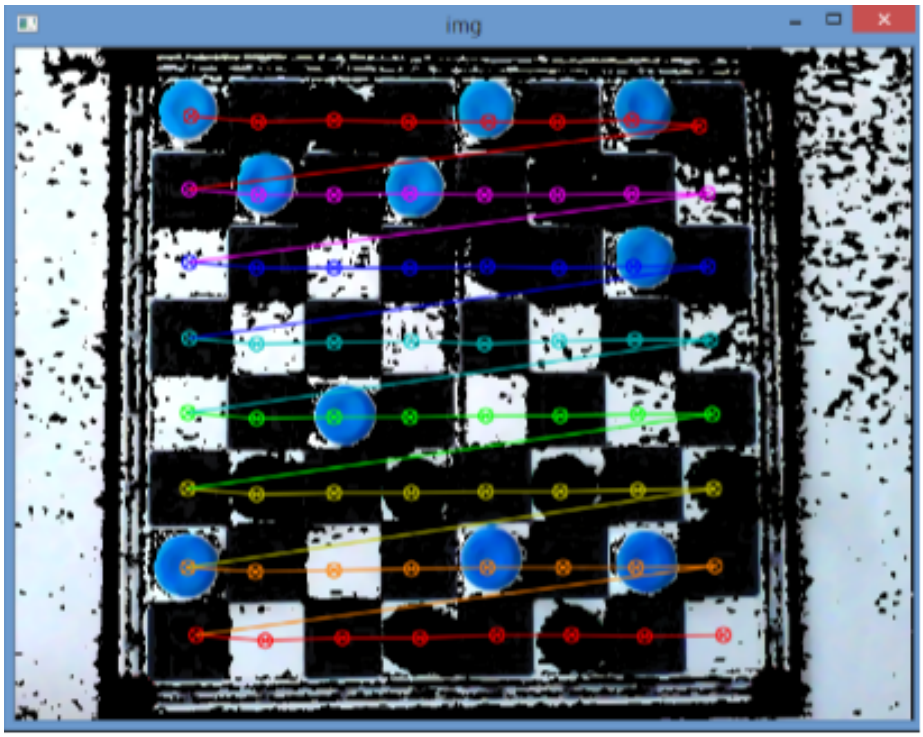
Rysunek 5: Wyznacznone środki pól na szachownicy

Jednym z napotkanych problemów było zmieniająca się kolejność wyszukiwanych krawędzi. Powodowało to brak stałej reprezentacji macierzy.



Rysunek 6: Nieprawidłowa kolejność rozpoznanych krawędzi

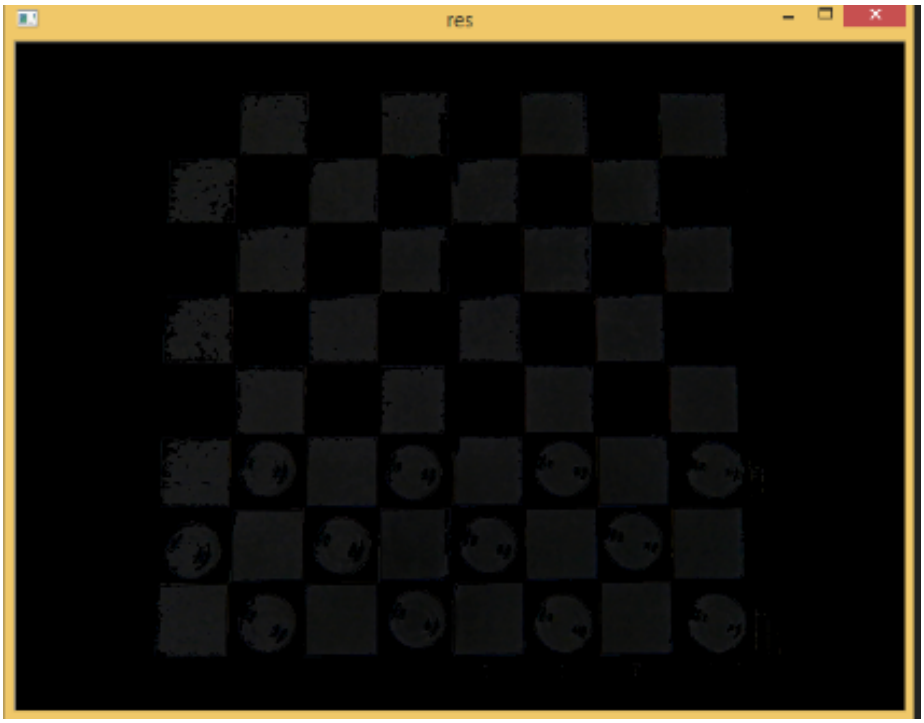
Rozwiązaniem tego problemu było sprawdzenie czy pierwszy element szachownicy znajduje się w lewym, górnym rogu przetwarzanego obrazu. Jeżeli taki warunek niebyłby spełniony to macierz zostaje obrócona o 90, 180 lub 270 stopnii.



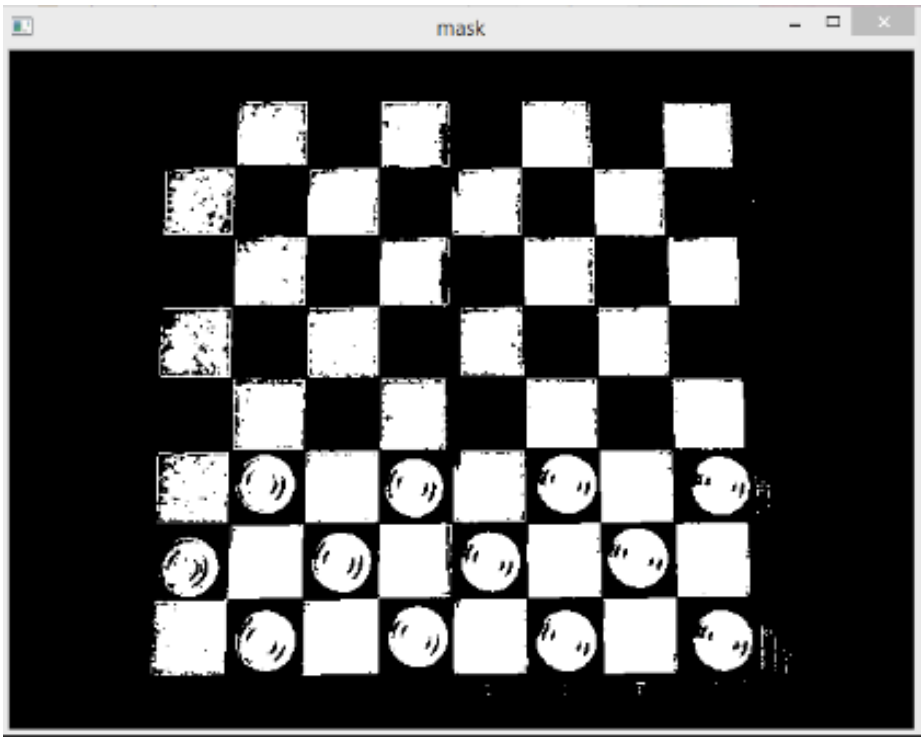
Rysunek 7: Prawidłowa kolejność rozpoznanych krawędzi

4.2 Rozpoznawanie kolorów

Do wykonywanego rozpoznania pionków na planszy konieczna była zmiana kolorów, w jakich są reprezentowane. Powodem tego były problemy z rozpoznawaniem, czy na danym białym polu znajduje się biały pionek, czy też nie. Postanowiliśmy zmienić kolor pionków białych na niebieskie, a czarnych na zielone. Na poniższych dwóch rysunkach zaprez

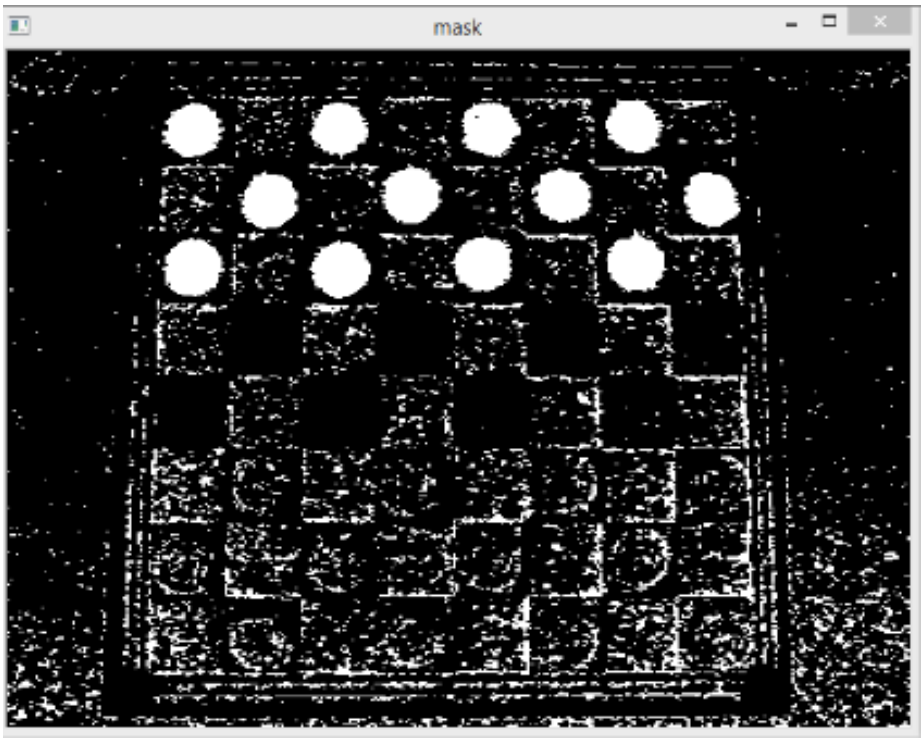


Rysunek 8: Rozpoznawanie koloru czarnego na planszy



Rysunek 9: Maska dla koloru czarnego

Dla przybliżenia różnicy poniżej ukazano maskę dla koloru niebieskiego.

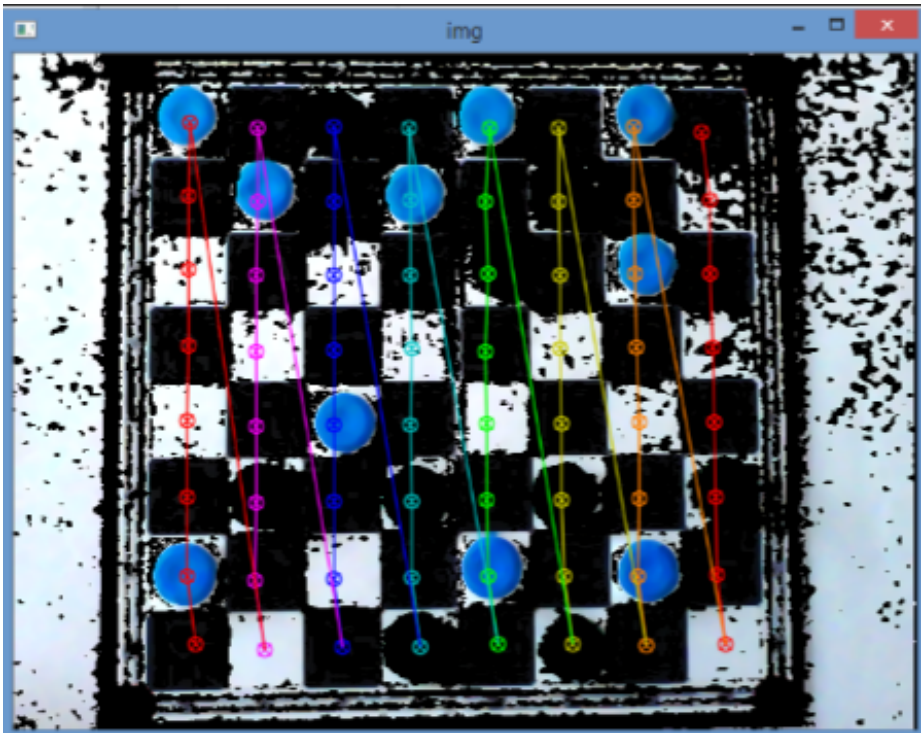


Rysunek 10: Maska dla koloru niebieskiego

Zmiana kolorów pionków na planszy skutecznie ulepszyła poprawność rozpoznawanych barw.

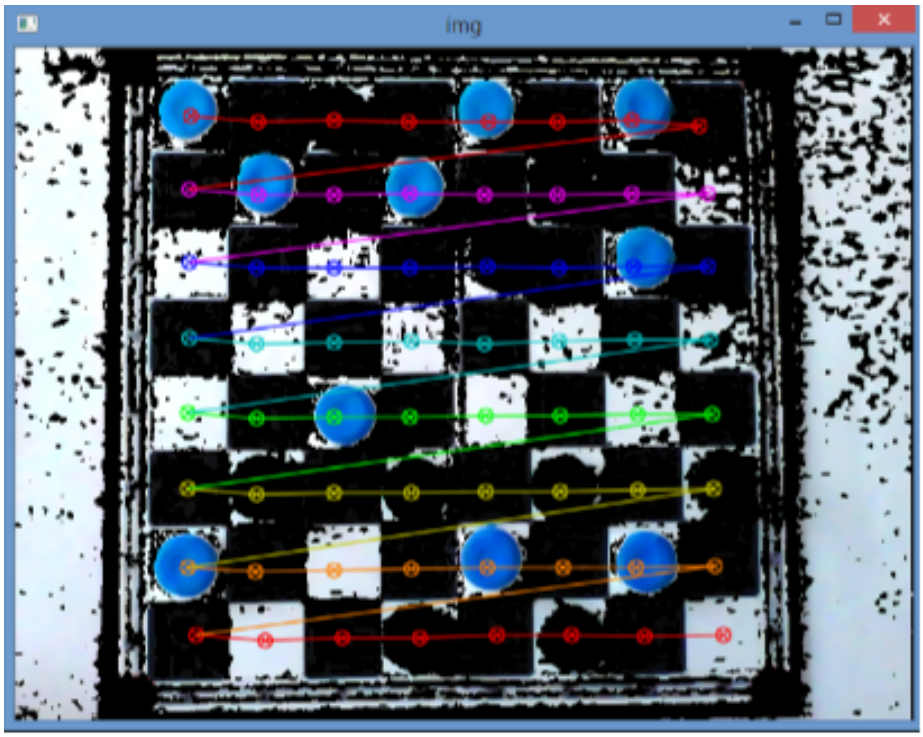
4.3 Korekcje rozpoznanego obrazu

Jednym z napotkanych problemów było zmieniająca się kolejność środków pól. Powodowało to brak stałej reprezentacji macierzy.



Rysunek 11: Nieprawidłowa kolejność rozpoznanych środków pól

Rozwiązaniem tego problemu było sprawdzenie, czy macierz środków pól jest dobrze zorientowana (jeżeli jest obrócona, moduł wykonuje symetryczne przekształcenie w drugą stronę). Odpowiedzialne są za to metody rotate i alter_chessboard_middles.



Rysunek 12: Prawidłowa kolejność rozpoznanych środków pól

System nakłada filtr medianowy (5px, co odpowiada próbkowaniu 25 pikseli naraz) na obraz oraz delikatnie wyostrza kontrast. Następnia sprawdza charakterystyczne cechy wskazanych obszarów obrazu (relacje, w jakich są w stosunku do siebie kanały R G i B).

```
def array_to_color(array):
    if Sampler.diff_bgr(array) < 40 and array[0] < 110:
        return 'black'
    if Sampler.diff_bgr(array) < 40 and array[0] > 130:
        return 'white'
    if Sampler.diff_bgr(array) > 80 and array[0] > array[1] > array[2]:
        return 'blue'
    if array[0] < array[1] > array[2]:
        return 'green'
```

Kiedy barwa piksela zostanie dopasowana do jednego z kolorów, zwracana jest słowna nazwa tej barwy.

4.4 Budowanie reprezentacji

Funkcja tworzy tablicę o wymiarach 8x8. Następnie dla każdego znalezionego punktu charakterystycznego szachownicy (czyli każdego ze środków pól) przypasowuje oznaczenie zależnie od wyników, jakie zwraca metoda opisana w poprzedniej podsekcji. Tworzony jest pusty obraz o wymiarach 500x500 pikseli.

Następnie w pętli nakłada się mniejsze "kafelkiżeprezentujące stan pola, przechowywane w folderze ze źródłami. Każdy kafelek ma wymiary 50x50 pikseli. Jeżeli kolor nie został rozpoznany, pole jest reprezentowane jako przekreślony na czerwono kwadrat.

4.5 Wykorzystanie biblioteki checkers 0.2

Jako część (moduł) sprawdzania poprawności ruchów wykorzystaliśmy bibliotekę checkers 2.0. Biblioteka ta w oryginalnej wersji nie daje dostepu do funkcji zmieniania ruchu, jest to domyślnie obsługiwane zdarzeniami przechwytywanymi z kliknięć myszki na wirtualną planszę.

Aby móc operować z poziomu kodu na zmianach planszy należało opakować bibliotekę w dodatkowe funkcję takie jak:

```
# funkcja ustawia numer pola, z ktorego pionek rusza sie w aktualnym ruchu
# oraz numer aktualnie poruszanego pionka
# przyjmuje dwa argumenty: numer pola i numer pionka
def set_piece(self, piece_square, piece):
        print "i am in set_piece"
        if self.DEBUG_PRINT_FUNCTIONS:
            pass;
            print "set_piece"
        try:
            self.piece_square = piece_square
            self.piece = piece
        except ValueError:
            return
        self.got_piece = 1

        if self.check_piece():
        # positive numbers are failure, for check_piece
            self.piece_square = None
            self.piece = None
            self.got_piece = 0

# funkcja ustawia numer pola,
# na ktore ma przesunac sie pionek w aktualnym ruchu
# jako argument przyjmuje numer pola
def set_square(self, square):
        print "i am in set_square"
        if self.DEBUG_PRINT_FUNCTIONS:
            pass;
            print "got_square_click"
        if self.got_piece:
            self.square = square
            if self.DEBUG:
                print "got square:", self.square
            self.got_move = 1
```

, które są odpowiednikami funkcji dostępnych w bibliotece, ale jako argumenty przyjmują odpowiednie wartości zamiast zdarzenia typu event.

4.6 Problem komunikacji z obiektem w osobnym wątku Tkinter

Biblioteka checkers 0.2 tworzy obiekt typu Tkinter i na jego podstawie wyświetla wirtualną planszę. Wszystkie funkcje oceniające ruch są w obrębie klasy tego obiektu, dlatego projekt ten wymagał użycia dwóch wątków. Komunikacja między wątkami została zrealizowana w oparciu o wzorzec używający wspólnego obiektu kolejki dla dwóch wątków.

```
class GuiPart:
    def __init__(self, master, queue, endCommand):
        self.queue = queue
        # Set up the GUI
        console = Tkinter.Button(master, text='Done', command=endCommand)
        console.pack()
        # Add more GUI stuff here depending on your specific needs

    def processIncoming(self):
        """Handle all messages currently in the queue, if any."""
        while self.queue.qsize():
            try:
                msg = self.queue.get(0)
                # Check contents of message and do whatever is needed. As a
                # simple test, print it (in real life, you would
                # suitably update the GUI's display in a richer fashion).
                print msg
            except Queue.Empty:
                # just on general principles, although we don't
                # expect this branch to be taken in this case
                pass

class ThreadedClient:
    """
    Launch the main part of the GUI and the worker thread. periodicCall and
    endApplication could reside in the GUI part, but putting them here
    means that you have all the thread controls in a single place.
    """
    def __init__(self, master):
        """
        Start the GUI and the asynchronous threads. We are in the main
        (original) thread of the application, which will later be used by
        the GUI as well. We spawn a new thread for the worker (I/O).
        """
        self.master = master

        # Create the queue
        self.queue = Queue.Queue()
```

```
        # Set up the GUI part
        self.gui = GuiPart(master, self.queue, self.endApplication)

        # Set up the thread to do asynchronous I/O
        # More threads can also be created and used, if necessary
        self.running = 1
        self.thread1 = threading.Thread(target=self.workerThread1)
        self.thread1.start()

        # Start the periodic call in the GUI to check if the queue contains
        # anything
        self.periodicCall()

    def periodicCall(self):
        """
        Check every 200 ms if there is something new in the queue.
        """
        self.gui.processIncoming()
        if not self.running:
            # This is the brutal stop of the system. You may want to do
            # some cleanup before actually shutting it down.
            import sys
            sys.exit(1)
        self.master.after(200, self.periodicCall)

    def workerThread1(self):
        """
        This is where we handle the asynchronous I/O. For example, it may be
        a 'select()'. One important thing to remember is that the thread has
        to yield control pretty regularly, by select or otherwise.
        """
        while self.running:
            # To simulate asynchronous I/O, we create a random number at
            # random intervals. Replace the following two lines with the real
            # thing.
            time.sleep(rand.random() * 1.5)
            msg = rand.random()
            self.queue.put(msg)

    def endApplication(self):
        self.running = 0
```

Źródło: *https://www.safaribooksonline.com/library/view/python-cookbook/0596001673/ch09s07.html*

Odpowiednio moduł logiki uruchamiany jest w wątku *GuiPart*, a przechwytywanie obrazu i ustalanie jakie zmiany na planszy zostały wykonane pracują w wątku *workerThread1*.

Po wykryciu ruchu wątek przechwytywania obrazu wstawia do obiektu ko-

lejki obiekt z polam:

- numer pola, z którego nastąpił ruch
- numer pionka poruszanego
- numer pola, na które pionek został przesunięty

4.7 Synchronizacja między wątkami

Do synchronizacji między wątkami używamy trzech głównych flag:

CHECK_COMPLETE = 0
ACCEPT_MOVE = 0
JUMBS = 0

Odpowiednio oznaczają one:

- czy sprawdzanie ruchu zakonczylo sie
- czy ruch zostal zaakceptowany
- czy aktualnie wykonywany ruch "ma bicia"

Flaga *CHECK_COMPLETE* jest zerowana w momencie rozpoznawania, przed wysłaniem do modułu logiki, który pionek został poruszony na planszy w module sprawdzanie planszy. Flaga ta ustawiana jest na jeden kiedy sprawdzanie poprawności zakończyło się.

Flaga *ACCEPT_MOVE* jest zerowana w module rozpoznawania ruchów na planszy, przed wysłaniem ruchu do modułu logiki. Flaga ta jest ustawiana na jeden w momencie gdy ruch zostanie sprawdzony i zaakceptowany. Dzięki temu moduł rozpoznawania wie czy ruch został zaakceptowany i czy ma podmienić tablice ze starej na nową.

Flaga *JUMBS* jest zerowana w module logiki, gdy następny ruch nie ma przewidzianych bić. Flaga ta jest ustawiana na jeden w momencie, gdy następny ruch ma przewidziane bicia. Dzięki temu moduł rozpoznawania wie czy zachodzą dwie zmainy na planszy czy trzy.

4.8 Reprezentacja planszy

Plansza w programie reperezentowana jest w postaci tablicy:

```
[[33, 77], [34, 78], [35, 79], [36, 80],
 [37, 81], [38, 82], [39, 83], [40, 84],
 [41, 85], [42, 86], [43, 87], [44, 88],
 [45,  0], [46,  0], [47,  0], [48,  0],
 [49,  0], [50,  0], [51,  0], [52,  0],
 [53, 65], [54, 66], [55, 67], [56, 68],
 [57, 69], [58, 70], [59, 71], [60, 72],
 [61, 73], [62, 74], [63, 75], [64, 76],]
```

Powyżej pokazane zostało ustawienie początkowe, każdy element tablicy to pole na planszy, na którym jest rozgrywana gra (w przypadku tej implementacji są to pola białe). Każda tablica tego elementu posiada dwa elementy, pierwszy z nich to numer pola na planszy, a drugi to numer pionka. Jeśli na miejscu numeru pionka jest zero, oznacza to, że na polu nie ma pionka.

W programie przechowywane są również numery pionków przypisane do odpowiedniego koloru:

```
{
    'black': [65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76],
    'red': [77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88]
}
```

4.9 Wykrywanie zmian na planszy

Za wykrywanie konkretnego ruchu odpowiada funkcja *checkTables*, która za argument przyjmuje nową, zauktualizowaną tablicę. Porównuje nową tablicę ze stara przechowywaną, czyli ostatnią z prawidłowym ruchem. Wyszukuję, które pola zmieniły swoją wartość, czyli na których zmieniły się wartości odpowiadające za numer pionka. Jednocześnie zlicza ilość tych zmian. Jest kilka scenariuszy jakie może przejść ta funkcja:

- jeśli zmian było dokładnie dwie to tworzy obiekt składający się z numerów pól skąd i dokąd i numeru pionka. Nastepnie wstawia ten obiekt do kolejki i oczekuję na zmianę flagi oznaczającej zakończenie sprawdzania. Po zakończeniu sprawdzania, sprawdza czy flaga akceptacji jest ustawiona na jeden, jeśli tak to podmienia tablice, jeśli nie to wyświetla wiadomości o powrocie do stanu poprzedniego i oczekuje na zmiany.
- jeśli zmian było dokładnie trzy, a flaga odpowiadająca za bicia nie jest ustawiona na jeden to flaga akceptacji jest ustawiana na zero oraz wyświetla wiadomości o powrocie do stanu poprzedniego i oczekuje na zmiany.
- jeśli zmian było dokładnie trzy i flaga bić była ustawiona na jeden to tworzy obiekt składający się z numerów pól skąd i dokąd i numeru pionka. Nastepnie wstawia ten obiekt do kolejki i oczekuję na zmianę flagi oznaczającej zakończenie sprawdzania. Po zakończeniu sprawdzania, sprawdza czy flaga akceptacji jest ustawiona na jeden, jeśli tak to podmienia tablice, jeśli nie to wyświetla wiadomości o powrocie do stanu poprzedniego i oczekuje na zmiany.
- jeśli zmian nie było to oczekuje na dalsze zmiany.
- jeśli zmian było więcej niż trzy to wyświetla informację o powrocie do ostatniego dobrego ustawienia i oczekuje na zmiany.

```
def checkTables(self, new_table):
        count = 0
        squares_from = []
        squares_to = []
        square_from = 0
```

```
square_to = 0
piece = 0
piece_remove = []
for old_square, new_square in zip(self.main_checkers_table, new_table):
    if old_square[1] <> new_square[1]:
        if new_square[1] == 0:
            piece_remove.append(old_square[1])
            squares_from.append([old_square[0], old_square[1]])
        if old_square[1] == 0:
            squares_to.append([new_square[0], new_square[1]])
        count += 1

if count > 1 and count < 4:
    if count == 2:
        for move in squares_from:
            if move[1] == squares_to[0][1]:
                square_from = move[0]
                square_to = squares_to[0][0]
                piece = move[1]
                piece_remove.remove(piece)
        move = (square_from, piece, (square_to,))
        print count, move, piece_remove
        self.queue.put(move)

    if count == 3 and self.gui.jumps:
        for move in squares_from:
            if move[1] == squares_to[0][1]:
                square_from = move[0]
                square_to = squares_to[0][0]
                piece = move[1]
                piece_remove.remove(piece)
        move = (square_from, piece, (square_to,))
        print count, move, piece_remove
        self.queue.put(move)
    if count == 3 and not (self.gui.jumps):
        self.gui.set_accept_move(0)
        self.gui.set_check_complete(1)
    while self.gui.get_check_complete() == 0:
        print 'wait for complete'
        time.sleep(0.5)
    if self.gui.get_accept_move():
        print 'Accept move is 1'
        self.main_checkers_table = new_table
        self.gui.set_accept_move(0)
    else:
        self.gui.show_message('Get back to this state', 1)

    self.gui.set_check_complete(0)
else:
    if count != 0:
```

```
            self.gui.show_message('Get back to this state', 1)
        else:
            pass
```

4.10 Ilość przechwytywnych obrazów

To ile przechwytów obrazu nastąpi zanim sprawdzimy, który pionek gdzie się ruszy zależy od tego czy w danej kolejce jest przewidziane bicie. Jeśli nie ma bicia to dwa pola zmienią swoją wartość i będzie jeden przechwyt obrazu. Jeśli przewidywane jest bicie to będą trzy zmiany wartości pól na planszy i potrzeba dwa przechwyty obrazu: jeden przesunięcie pionka, drugi zabranie zbitego pionka z planszy.

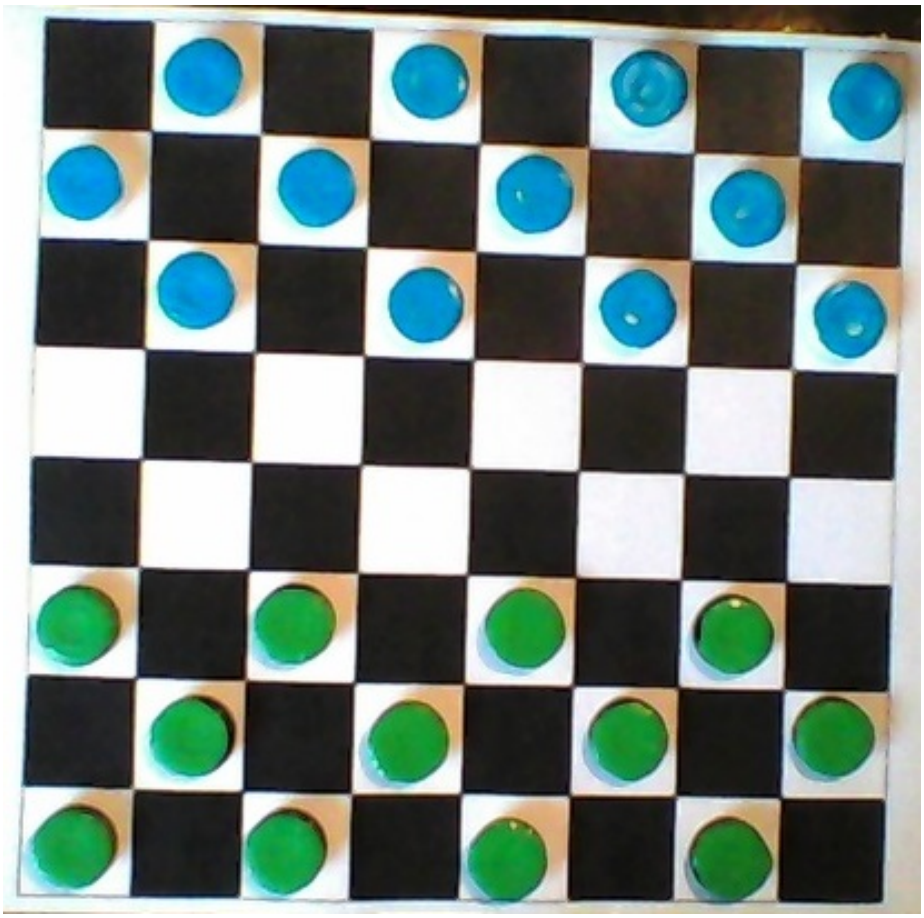
```
jump_now = FALSE
    if self.gui.jumps[1]:
        print 'get in jump'
        jump_now = TRUE
        for jump in self.gui.jumps[1]:
            for piece in self.gui.pieces[self.gui.moving]:
                # print 'jump in for: ', jump
                # print 'piece: ', piece
                if piece == jump:
                    print 'Jump now is false'
                    jump_now = FALSE
```

Realizacja przechwytu stanu gry w zależności od tego czy następuje bicie:

```
if maps is not None:
    if jump_now:
        count += 1
    path1 = path2
    map_curr = maps[0]
    map_new = maps[1]
    if count == 2 and jump_now:
        count = 0
        self.checkTables(map_curr)
    if not (jump_now):
        self.checkTables(map_curr)
```

5 Użytkowanie

Gra rozpoczyna się, gdy ustawimy pionki w pozycji początkowej (rysunek z pozycją poczatkową). Pierwszy zaczyna gracz rozgrywający pionkami niebieskimi.



Rysunek 13: Ustawienie początkowe pionków na planszy.

Jeśli przechwycony stan planszy różni się od poprzedniego stanu to oznacza, że nastąpił ruch.
Jeśli ruch odbył się jednym pionkiem w przypadku ruchu bez bicia, to jest to ruch akceptowany i wysyłany do modułu logiki gry.
Jeśli przechwycony stan planszy ma więcej różnic, a nie było przewidzianego bicia, wtedy ruch nie jest przesyłany do modułu logiki w celu oceny poprawności. Moduł logiki gry sprawdza poprawność ruchu.
Jeśli ruch jest zrobiony pionkiem, który aktualnie nie ma prawa do ruchu, posunięcie nie jest akceptowane i program czeka, aż stan planszy powróci do ostatniego poprawnego ustawienia.
Jeśli użytkownicy powrócą do poprzedniego stanu gry program odblokowuje sprawdzanie zmian na planszy.

5.1 Reprezentacja częściowych wyników

W celu wygodnego, przejrzystego prezentowania częściowych wyników działania algorytmów, stworzono klasę, której głównym zadaniem było udostępnie-

nie metod pozwalających na czytelne wyświetlenie danych pewnego typu oraz zamianę reprezentacji tablic stanów ze znakowej na liczbową.

```
    def strarr_to_intarr(array):
        int_array = copy.deepcopy(array)
        arr = []
        verse = []
        for row in int_array:
            for element in row:
                square = {
                    'white': 0,
                    'blue': 1,
                    'green': 2,
                    'black': None
                }[element]
                if square is not None:
                    verse.append(square)
            arr.append(verse)
            verse = []
        return arr
```

Metoda strarr_to_intarr przyjmuje jako parametr listę reprezentującą stan szachownicy zapisany jako ciąg liter. Na wyjście zwraca tablicę reprezentującą stan szachownicy zapisany w postaci ciągu liczb. Lista wejściowa zostaje skopiowana, następnie dla każdego jej elementu wykonnywane jest rzutowanie wartości tekstowej na liczbową. Dopuszczono możliwość wystąpienia obiektów None dla nierozpoznanych pól.

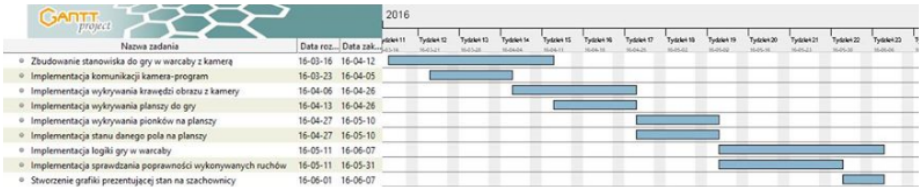
```
def image_to_text_array(image, coord, number):
        result = [[]]
        a = 0
        b = 0
        table = PrettyTable()
        pos = []
        color = []
        clear_row = ['','','','','','','','']
        for element in reversed(coord.astype(int)):
            result[b].append(Sampler.array_to_color(image[element[0][1]][elemen
            a += 1
            pos.append(str(element[0][1]) + ' ' + str(element[0][0]))
            color.append(str(image[element[0][1]][element[0][0]]))
            if a == number:
                table.add_row(result[b])
                table.add_row(pos)
                table.add_row(color)
                table.add_row(clear_row)
                color = []
                pos = []
                a = 0
                b += 1
                if b != number:
```

```
                result.append([])
        if Sampler.debug == 1:
            print(table)
        return result
```

Metoda image_to_text_array jako parametry przyjmuje obraz, listę koordynatów wszystkich środków pól. Parametr number został dodany do umożliwienia ustalenia wiersza o zmiennej szerokości (możliwość ostatecznie raczej nie była wykorzystywana).

6 Harmonogram

Prace wykonane przy projekcie staraliśmy się wykonywać według harmonogramu zaplanowanego na początku semestru. Czas potrzebny na realizację poszczególnych zadań był przybliżony w stosunku do tego jaki zakładaliśmy. Do stworzenia harmonogramu wykorzystaliśmy środowisko Gantt Project.



Rysunek 14: Harmonogram prac nad projektem

7 Podsumowanie

Praca nad systemem rozponającym stan gry w warcaby stanowi złożony projekt. Zadanie wymagało rozplanowania pracy, podzielenia obowiązków oraz przemyślenia architektury i używanych narzędzi. Zdobycie doświadczenia w rozwiązywaniu złożonych problemów stało się wartościowym doświadczeniem dla zespołu. Dekompozycja projektu na pomniejsze zadania umożliwiła rozsądne gospodarowanie zasobami ludzkimi. Realizacja projektu przyczyniła się również do wykorzystania praktycznych możliwości systemów kontroli wersji, takich jak aktualizacja, scalanie i przywracanie zasobów.

8 Literatura