Podstawy Telefinformatyki - dokumentacja

Marcin Borysiewicz
136218 marcin.borysiewicz@student.put.poznan.pl

Kornelia Maik
135465 kornelia.maik@student.put.poznan.pl

Patryk Wenz
136325 patryk.wenz@student.put.poznan.pl

Piotr Zieliński
136333 piotr.r.zielinski@student.put.poznan.pl

11 września 2020

Spis treści

1 Opis projektu

Projekt *Checkers* to system służący do sprawdzania poprawności ruchów wykonywanych podczas gry w warcaby. System rozpoznaje obraz z gry w warcaby oraz wizualizuje stan gry na komputerze.

System został opracowany tak, aby był w stanie poprawnie opisać bieżące położenie i rodzaj pionków na planszy poprzez rozpoznanie ich koloru (przynależności do gracza) i stwierdzenie, czy dany pionek jest damką.

Zaimplementowany został moduł badający ruchy wykonane przez graczy i weryfikujący ich poprawność - moduł ten rozszerza funkcjonalność systemu o badanie nie tylko zgodnego z założeniami gry stanu planszy, ale też ciągłe badanie poprawnego przebiegu gry.

Przykładowym zastosowaniem systemu jest monitorowanie gier przeprowadzanych w ramach turniejów w warcaby. Kamera umieszczona nad stołem, na którym umieszczona jest plansza przekazywałaby obraz do systemu, który monitorowałby poprawne ruchy wykonywane przez uczestników turnieju i informowała prowadzących o ewentualnym łamaniu zasad lub zakończeniu gry.

System może być zastosowany jako podstawa rozbudowanego systemu wykorzystującego elementy sztucznej inteligencji przewidujące możliwe do wykonania ruchy, które prowadziłyby do zwycięstwa gracza. Innym rozbudowaniem systemu może być bot wykonujący dozwolone w danym momencie ruchy, z którym możliwe jest przeprowadzenie rozgrywki.

Prezentację działania systemu można znaleźć pod tym linkiem:

LINK DO FILMU NA YOUTUBE

2 Podział prac

Autor	Podzadanie
Marcin Borysiewicz	Wizualizacja stanu gry
Kornelia Maik	Moduł wizyjny, grafiki plansz
Patryk Wenz	Moduł wizyjny
Piotr Zieliński	Moduł zasad gry

3 Opis implementacji

System został zaimplementowany poprzez trzy moduły: moduł wizyjny, moduł zasad gry i wizualizator planszy. Moduł wizyjny pozwala na rozpoznawanie obrazu planszy i stanu gry, czyli położenia pionków oraz ich rodzaju: przynależności do jednej ze stron gry oraz tego, czy dany pionek jest damką. Moduł zasad gry weryfikuje każde posunięcie zgodnie z zasadami rozgrywek warcabowych, wykrywając zarówno błędne ruchy, jak i niewykonane bicia czy ruch w nie swojej turze. Wizualizator planszy wyświetla graficzną reprezentację bieżącego stanu gry.

3.1 Podprogramy zdefiniowane w systemie

Podprogramy zostały opisane z podziałem na moduł i plik, do którego należą. Moduł wizyjny jest w implementacji nazwany jako Vision Module, natomiast moduł zasad gry przyjmuje nazwę Rules Module.

Vision Module
calibration.py

- def path_to_src(+dir: str, +filename: str) -> -path:str
  - Argumenty wejściowe:
    - dir - *str*: nazwa folderu,
    - filename - *str*: nazwa pliku.
  - Argumenty wyjściowe:
    - path - *str*: absolutna ścieżka do pliku,
  - Opis:
    Funkcja tworząca absolutną ścieżkę do pliku (zwykle z obrazem) w danym folderze projektu. Zwraca łańcuch tekstowy ze ścieżką.
- def load_image(+filename: str) -> -picture: np.array
  - Argumenty wejściowe:
    - filename - *str*: nazwa pliku.
  - Argumenty wyjściowe:
    - picture - *np.array*: obraz w postaci macierzy numpy,

- Opis:

  Funkcja wczytująca kolorowy obraz o danej nazwie. Zwraca obraz w formacie tablicy dwuwymiarowej bilblioteki numpy.

- def save_configs(+filename: str) -> -path: str
  - Argumenty wejściowe:
    - filename - *str*: nazwa pliku.
  - Argumenty wyjściowe:
    - path - *str*: absolutna ścieżka do zapisania pliku konfiguracyjnego.
  - Opis:

    Funkcja tworząca absolutną ścieżkę do pliku config. Zwraca łańcuch tekstowy ze ścieżką.

- def resize(+scale_percent: int, +image: np.array) -> -scaled_image : np.array
  - Argumenty wejściowe:
    - scale_percent - *int*:wartość, do której ma być przeskalowany obraz.
    - image - *np.array* obraz wejściowy w postaci macierzy numpy.
  - Argumenty wyjściowe:
    - scaled_image - *np.array*: przeskalowany obraz w postaci macierzy numpy.
  - Opis:

    Funkcja pozwalająca na przeskalowanie obrazu bazowego do zadanego rozmiaru (powiększenie lub zmniejszenie). Zwraca obraz o zmienionych wymiarach.

- def load_configs(+filename="config.txt": str) -> -params: dict
  - Argumenty wejściowe:
    - filename - *str* - nazwa pliku do wczytania.
  - Argumenty wyjściowe:
    - params - *dict*: słownik z parametrami konfiguracyjnymi.

- Opis:

  Funkcja pobierająca parametry konfiguracyjne z pliku tekstowego. Zwraca słownik z wymaganymi danymi to inicjalizacji.

- def find_circles(+img: np.array) -> -board: list(dict())

  - Argumenty wejściowe:

    * img - *np.array* - obraz w potaci macierzy numpy.

  - Argumenty wyjściowe:

    * board - *list(dict())*: lista słowników opisująca planszę.

  - Opis:

    Funkcja pozwalająca na znalezienie wszystkich okręgów na obrazie. Wykorzystuje funkcję szukającą konturów na obrazie. Dzięki wcześniejszej konfiguracji parametrów pozwala na wykrycie okręgów na dowolnym obrazie. Zwraca listę słowników, w którym zawarte są informacje o położeniu i kolorze pionka.

- def calibrate_params(+filename: str):

  - Argumenty wejściowe:

    * filename - *str* - nazwa pliku do zapisania konfiguracj.

  - Opis:

    Funkcja pozwalająca na dostosowanie parametrów konfiguracyjnych dla danej planszy. Otwiera menu konfiguracyjne, w którym dostosowujemy wartości dwóch parametrów, tak aby wszystkie okręgi zostały wykryte.

- def get_block_size(+image: np.array): -> -h, -w: int

  - Argumenty wejściowe:

    * image - *np.array* - obraz w postaci macierzy numpy.

  - Argumenty wyjściowe:

    * h - *int*: wartość wysokości pojedynczego bloku.
    * w - *int*: wartość szerokości pojedynczego bloku.

  - Opis:

    Funkcja pobierająca parametry konfiguracyjne z pliku tekstowego. Zwraca słownik z wymaganymi danymi to inicjalizacji.

- def get_block_id_by_cords(+cords: tuple(), +blocksizes: tuple()): -> -id: int
  - Argumenty wejściowe:
    - cords - *tuple()* - wartości wsółrzędnych na obrazie danego bloku.
  - Argumenty wyjściowe:
    - id - *int*: wartość identyfikatora danego bloku.
  - Opis:

    Funkcja obliczająca pozycje numer bloku na planszy po po podaniu wartości x, y na obrazie oraz rozmiaru bloku. Zwraca wartość id bloku.

- def mouse_drawing(event, x, y, flags, params):
  - Argumenty wejściowe:
    - event - *event* - rodzaj akcji wywoływanej przez mysz,
    - x - *int* - wartość współrzędnej szerokości, na której znajduje się mysz,
    - y - *int* - wartość współrzędnej wysokości, na której znajduje się mysz.
    - flags - *flag* - flagi przekazywane w wydarzeniu.
    - params - *args** - dodatkowe parametry.
  - Opis:

    Funkcja pomocnicza do zwymiarowania planszy na obrazie. Pozwala na ręczne wybranie rogów planszy.

- def calibrate_image(+img: np.array):
  - Argumenty wejściowe:
    - image - *np.array* - obraz w postaci macierzy numpy.
  - Opis:

    Funkcja pozwalająca na ręczne wybranie punktów planszy, a następnie zmianę perspektywy na poziomą.

image_ processing.py

- def get_corners(+img: np.array, param1=2, param2=3, param3=0.04): -> -corners: list(tuple())
  - Argumenty wejściowe:
    - img - *np.array* - obraz w postaci macierzy numpy.
    - param1 - *float* - wartość parametru dla funkcji *cornerHarris*.
    - param2 - *float* - wartość parametru dla funkcji *cornerHarris*.
    - param3 - *float* - wartość parametru dla funkcji *cornerHarris*.
  - Argumenty wyjściowe:
    - corners - *list(tuple())*: lista współrzędnych wszystkich wierzchołków.
  - Opis:

    Funkcja pozwalająca na automatyczne wyszukanie rogów planszy. Wykorzystuje funkcję Harrisa do wykrywania rogów. Zwraca listę ze współrzędnymi wszystkich rogów znajdujących się na planszy.

- def get_valid_corners(+corners: list(tuple()): -valid_corners: tuple()
  - Argumenty wejściowe:
    - corners - *list(tuple())* - lista współrzędnych wszystkich wierzchołków.
  - Argumenty wyjściowe:
    - valid_corners - *tuple()*: współrzędne znaczących wierzchołków.
  - Opis:

    Funkcja znajdująca wśród wszystkich rogów cztery najważniejsze - lewy dolny i górny oraz prawy dolny i górny planszy.

- def generate_chessboard(): -> -chessboard: list(dict())
  - Argumenty wyjściowe:
    - chessboard - *list(dict())*: lista słowników opisujących poszczególne pola planszy.

– Opis:

Funkcja generująca listę słowników opisu kolejnych pól na planszy. Zawiera informacje na temat id, koloru pionka, pola i damek. Zwraca listę z zainicjowanymi podstawowymi informacjami na temat planszy (pola o pionku ustawione na *None*, a o damce na *False*).

- def is_crown(+given_area: np.array) -> -flag: Boolean:
  - Argumenty wejściowe:
    - given_area - *np.array* - fragment obrazu w postaci macierzy numpy do sprawdzenia pod kątem bycia damką,
  - Argumenty wyjściowe:
    - flag - *Boolean*: flaga określająca czy dana figura jest damką.
  - Opis:

    Funkcja sprawdzająca, czy na danym fragmencie obrazu znajduje się damka. Funkcja analizuje liczbę rogów na okręgu pionka. Jeśli jest zgodna z liczbą rogów na koronie zwraca wartość *True*, jeśli nie *False*.

- def get_board_from_border(+img: np.array, +points: list()): -> -result: np.array
  - Argumenty wejściowe:
    - img - *np.array* - obraz w postaci macierzy numpy.
    - points = *list()* - punkty, w których obraz ma zostać wycięty.
  - Argumenty wyjściowe:
    - result - *np.array*: wycięty obraz w postaci macierzy numpy.
  - Opis:

    Funkcja umożliwiająca zmianę perspektywy obrazu oraz usunięcie zbędnego tła. Zwraca obraz z samą planszą,

- def find_pieces(+img: np.array):
  - Argumenty wejściowe:

    * img - *np.array* - obraz w postaci macierzy numpy.
  - Opis:

    Funkcja wykrywająca wszystkie pionki i ewentualne damki. Edytuje zawartość listy słowników na temat aktualnego wyglądu planszy.

- def initialize_config_colors(+img: np.array):
  - Argumenty wejściowe:
    * img - *np.array* - obraz w postaci macierzy numpy.
  - Opis:

    Funkcja sprawdzająca jakiego koloru są pionki umieszczone na szachownicy. Sprawdza ich ilość i informuje o niepoprawnym ustawieniu początkowym. Dodaje dane dotyczące wartości RGB kolorów do pliku konfiguracyjnego w celu pobrania wartości w późniejszych etapach.

- def swap_dict_keys_and_vals(+d: dict()): -> -new_d: dict()
  - Argumenty wejściowe:
    * d - *dict()* - słownik do zmiany kluczy z wartościami.
  - Argumenty wyjściowe:
    * new_d - *dict()* - słownik z zamienionymi kluczami z wartościami.
  - Opis:

    Funkcja pomocnicza umożliwiająca zamianę kluczy z wartościami w słowniku. Zwraca zmieniony słownik.

Wizualizacja

- def create_board() -> new_board: list(list(int))
  - Argumenty wyjściowe:
    - Reprezentacja szachownicy w formie listy dwuwymiarowej
  - Opis:

    Funkcja zwracającą pustą listę dwuwymiarową typu **int** o rozmiarze 8 na 8.

- def place_starting_pieces(+board: list(list(int)))
  - Argumenty wejściowe
    - board - list(list(int)) - pusta szachownica
  - Opis:

    Funkcja "ustawia"pionki na podanej w parametrze szachownicy przypisując konkretnym komórkom wartości 1 (czarny pionek) lub 2 (biały pionek).

- def draw_board(+board: list(list(int)))
  - Argumenty wejściowe
    - board - list(list(int)) - szachownica do narysowania
  - Opis:

    Funkcja za pomocą biblioteki *PyGame* rysuje na ekranie obecną zawartość szachownicy podanej w parametrze.

- def get_cell_coordinates(+cell_no: int) -> coords: list(int)
  - Argumenty wejściowe
    - cell_no - int - numer pola w notacji PGN
  - Argumenty wyjściowe
    - coords - list(int) - lista zawierająca współrzędne pola na szachownicy
  - Opis:

    Funkcja oblicza i zwraca listę dwuelementową zawierającą współrzędne zadanego w parametrze pola na szachownicy
- def get_cell_no(+x: int, +y: int) -> cell_no: int
  - Argumenty wejściowe
    - x - int - Odcięta pola na szachownicy
    - y - int - Rzędna pola na szachownicy
  - Argumenty wyjściowe
    - cell_no - int - Numer pola w notacji PGN
  - Opis:

    Funkcja oblicza i zwraca numer pola w notacji PGN na podstawie współrzędnych pola w szachownicy.

- def move_piece(+board: list(list(int)), +cell_from: int, +cell_to: int)
  - Argumenty wejściowe
    - board - list(list(int)) - szachownica, na której wykonujemy ruch
    - cell_from - int - nr pola z którego przesuwamy pionek
    - cell_to - int - nr pola na który przesuwamy pionek
  - Opis:

    Funkcja przemieszcza w strukturze szachownicy element z podanego w parametrze **cell_from** pola na pole podane w parametrze **cell_to**.
- def remove_piece(+board: list(list(int)), +cell_no: int)
  - Argumenty wejściowe
    - board - list(list(int)) - szachownica, z której usuwamy pionek
    - cell_no - int - nr pola z którego usuwamy pionek
  - Opis:

    Funkcja usuwa ze struktury szachownicy zawartość podanego w parametrze **cell_no** pola.

Rules Module
rules.py

- def try_to_get_move_category(+board1: Board, +board2: Board) -> -move: dict()
  - Argumenty wejściowe:
    - board1 - *Board*: reprezentacja poprzedniej szachownicy
    - board2 - *Board*: reprezentacja obecnej szachownicy
  - Argumenty wyjściowe:
    - move - *dict()*: reprezentacja ruchu wykonanego pomiędzy poprzednią, a obecną sytuacją na szachownicy w notacji PGN z uwzględnieniem zbitej bierki
  - Opis:

    Funkcja sterująca systemu zasad, korzystająca z podfunkcji, decyduje czy wykonany ruch jest prawidłowy
- def get_difference(+board1: Board, +board2: Board) -> -fields: list()

  - Argumenty wejściowe:
    * board1 - *Board*: reprezentacja poprzedniej szachownicy
    * board2 - *Board*: reprezentacja obecnej szachownicy
  - Argumenty wyjściowe:
    * fields - *list()*: pola różniące się pomiędzy poprzednią, a obecną szachownicą
  - Opis:
    Funkcja uzyskuje na podstawie znanego stanu poprzedniej i następnej szachownicy pola różniące się w tej ostatniej.

- def was_this_regular_move(+changes: list(dict())) -> -regular_move: Boolean
  - Argumenty wejściowe:
    * changes - *list(dict()*: różniące się pola z uwzględnieniem stan pola przed i po ruchu
  - Argumenty wyjściowe:
    * regular_move - *Boolean*: czy był to zwykły ruch (o 1 pole)
  - Opis:
    Funkcja analizuje różniące się pola i decyduje, czy dany ruch był zwykłym ruchem.

- def was_this_regular_taking(+changes: list(dict())) -> -regular_move: Boolean
  - Argumenty wejściowe:
    * changes - *list(dict()*: różniące się pola z uwzględnieniem stan pola przed i po ruchu
  - Argumenty wyjściowe:
    * regular_move - *Boolean*: czy było to zwykłe bicie
  - Opis:
    Funkcja analizuje różniące się pola i decyduje, czy dany ruch był zwykłym biciem (1 bierka zbita przez inną, zwykłą bierkę, bądź w taki sposób)

- def was_this_queen_taking(+changes: list(dict())) -> -regular_move: Boolean

  - Argumenty wejściowe:
    * changes - *list(dict()*: różniące się pola z uwzględnieniem stan pola przed i po ruchu
  - Argumenty wyjściowe:
    * regular_move - *Boolean*: czy było to bicie przez damkę
  - Opis:

    Funkcja analizuje różniące się pola i decyduje, czy dany ruch był biciem wykonanym przez damkę (1 bierka zbita przez damkę, odległą i/lub kończącą ruch dalej niż 1 pole od bierki zbitej)

- def was_this_queen_move(+changes: list(dict())) -> -regular_move: Boolean
  - Argumenty wejściowe:
    * changes - *list(dict()*: różniące się pola z uwzględnieniem stan pola przed i po ruchu
  - Argumenty wyjściowe:
    * regular_move - *Boolean*: czy był to ruch wykonany przez damkę
  - Opis:

    Funkcja analizuje różniące się pola i decyduje, czy dany ruch był ruchem wykonanym przez damkę (o więcej niż 1 pole)

- def check_if_taking_is_required(+board_2: Board, +move_made: str)
  - Argumenty wejściowe:
    * board_2 - *Board*: reprezentacja stanu szachownicy po ruchu
    * move_made - *str*: reprezentacja wykonanego ruchu w notacji PGN
  - Opis:

    Funkcja analizuje całą szachownicę i sprawdza, czy po wykonaniu ruchu, będzie konieczne bicie, które jest w warcabach obowiązkowe.

3.2 Struktury danych wykorzystywane w systemie

Implementacja systemu wymagała od zespołu zaplanowania struktur danych wykorzystywanych w procesie przetwarzania planszy w celu pozyskania z niej danych na temat stanu gry.

Moduł VisionModule pobiera obraz z planszy i opisuje bieżący stan gry poprzez listę słowników. Każdy element listy reprezentuje pole na planszy, a jego budowę pokazuje poniższy słownik:

```
{
    "ID" : ID_FIELD_NUMBER
    "PIECE" : PIECE_COLOR_TYPE
    "CROWN" : IS_CROWN
    "FIELD" : FIELD_COLOR
}
```

W parach klucz, wartość kolejnymi kluczami jest identyfikator pola, kolor pionka stojącego na polu (lub jego brak, jeśli pole jest puste), fakt, czy pionek jest damką oraz kolor pola.

Wizualizator partii do przechowywania stanu szachownicy korzysta z dwuwymiarowej tablicy liczb całkowitych. Każda komórka w tej strukturze odpowiada jednemu z 64 pól na szachownicy. Wartość pola tablicy określa jaka bierka znajduje się na danym polu według poniższego schematu:

```
0 : EMPTY FIELD
1 : BLACK PIECE
2 : WHITE PIECE
3 : BLACK KING
4 : WHITE KING
```

4 Technologia

Do realizacji projektu użyto języka Python 3.8, programowano w środowisku PyCharm Community Edition 2020.2.1. Biblioteki użyte w projekcie to:

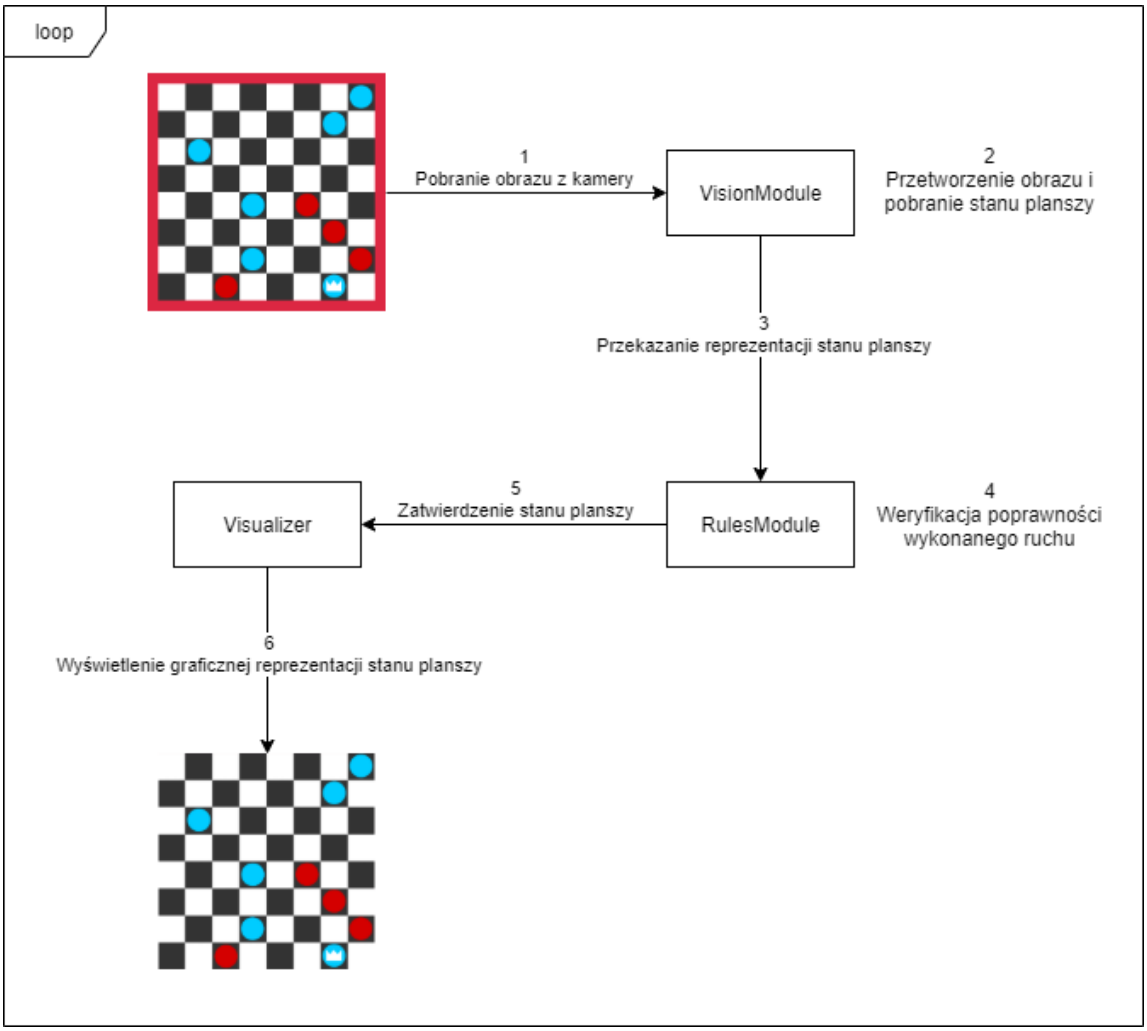
- *cv2* - biblioteka wykorzystywana podczas obróbki obrazu,
- *numpy* - biblioteka wykorzystywana do operacji numerycznych,
- *json* - biblioteka zapewniająca struktury danych używane podczas konfigurowania działania systemu,

- *pygame* - biblioteka wykorzystywana do wizualizacji wykonanych ruchów.

5 Architektura systemu

Budowę systemu przedstawia schemat współpracy modułów podczas powtarzalnego schematu działania wykonywanego przez system. Schematem tym są kolejno:

1. Pobranie obrazu z kamery.
2. Przetworzenie obrazu i pobranie z niego bieżącego stanu planszy przez moduł **VisionModule**. Moduł tworzy reprezentację stanu planszy przez wykrycie na niej pozycji pionków i ich rodzaju - koloru i faktu bycia damką.
3. Przekazanie reprezentacji stanu planszy do modułu **RulesModule**.
4. Weryfikacja poprawności wykonanego ruchu przez moduł **RulesModule**. Moduł kontroluje liczbę pionków na planszy, ich odpowiednie bieżące położenie na polach planszy i zmianę tych położeń, czyli ostatnio wykonany ruch. Informuje o ewentualnej konieczności wykonania wielokrotnego zbicia.
5. Poinformowanie modułu **Visualizer** o zatwierdzeniu bieżącego stanu planszy.
6. Wyświetlenie graficznej reprezentacji planszy przez moduł **Visualizer**.

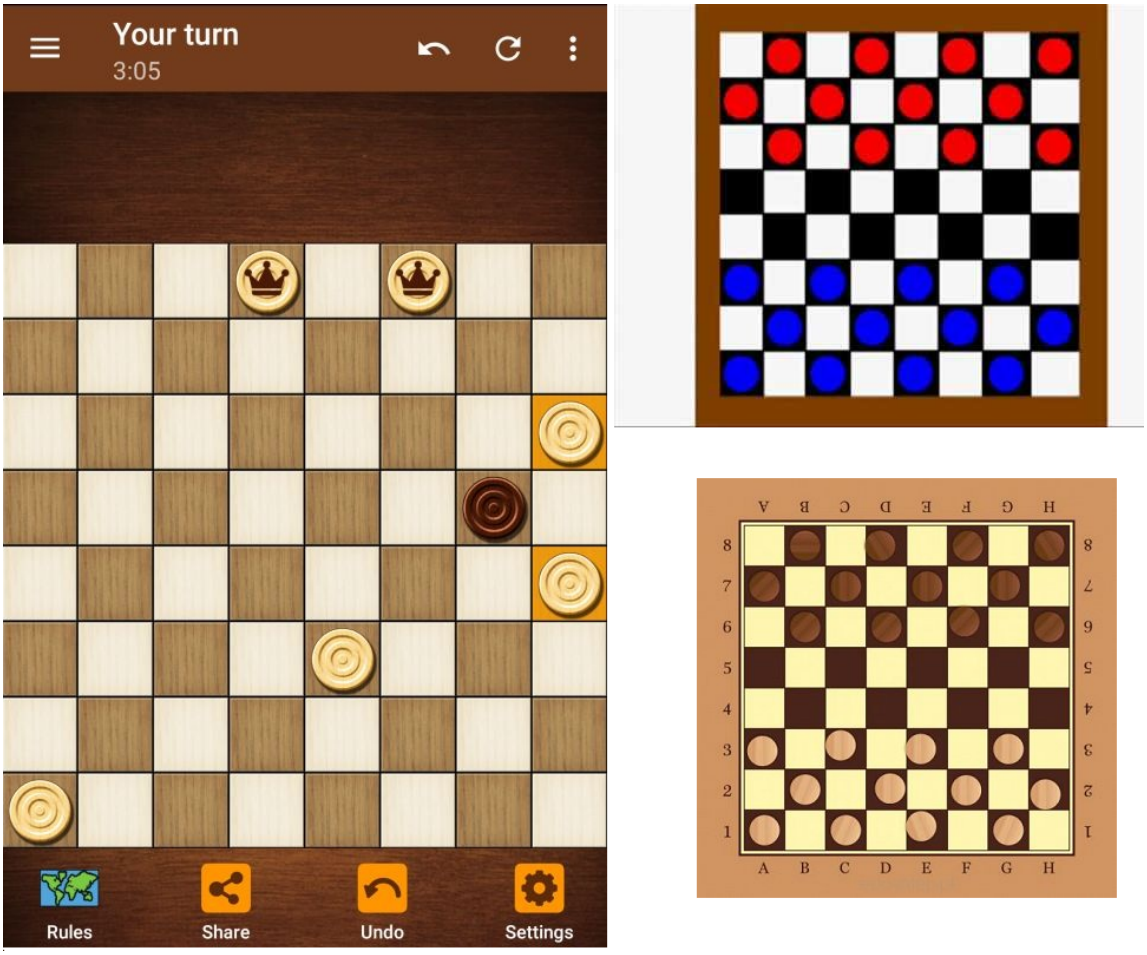


Rysunek 1: Schemat architektury systemu

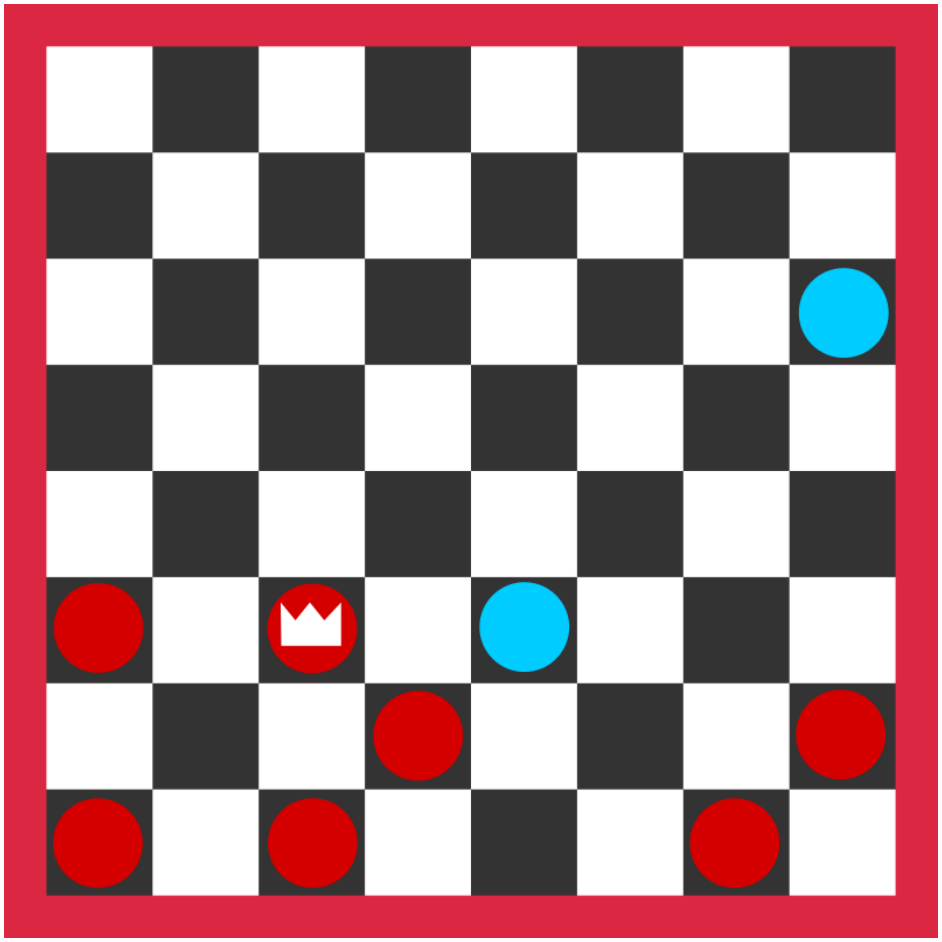
6 Napotkane problemy i ich rozwiązania

6.1 Brak fizycznych warcabów

Ze względu na brak fizycznej wersji gry byliśmy zmuszeni znaleźć inne rozwiązanie. W związku z tym zdecydowaliśmy się na własne plansze do gry w formacie *png* bazowane na obrazach komputerowej wersji warcabów np. zgodnych z grafiką poniżej.



Rysunek 2: Przykładowa gra w warcaby.



Rysunek 3: Obraz przekazywany do systemu.

W celu symulowania żywej rozgrywki dodaliśmy tło (w tym przypadku różowe), aby lepiej pokazać fakt, że hipotetyczna kamera „na żywo" prawdopodobnie wykryłaby coś więcej poza samą planszą.

System na wejściu otrzymuje sekwencyjne kolejne ruchy w postaci nowych plansz. Każda plansza jest przetwarzana, a system dokonuje porównania między obecną a kolejną planszą. Moduł zasad otrzymuje wszystkie informacje o położeniu i rodzaju pionków na planszy, odczytuje je, a następnie porównuje obydwie plansze i decyduje, czy ruch został wykonany poprawnie (jeśli nie to wysyłany jest stosowny komunikat o błędzie). Informacja na temat obecnego wyglądu planszy przekazywane są modułowi wizyjnemu, który uwidacznia obecny wygląd planszy i informuje użytkownika o rodzaju i poprawności działań na szachownicy.

6.2 Wykrywanie kolorów pionków

W początkowej fazie projektu algorytm do wykrywania kolorów pionków na planszy wyglądał całkowicie inaczej. Wykorzystywano histogram kolorów zwracany przez algorytm k - *means* dopasowujący wartości RGB do danej klasy.

Algorytm dzielił planszę na pojedyncze pola, a następnie badał wartości RGB na danym obszarze. Następowało sortowanie według ilości wystąpień danej klasy na obrazie. Dwie pierwsze wartości informowały kolejno o kolorze pola i pionka. Implementacja wyglądała następująco:

```
def create_histogram(self, cluster):
    labels = np.arange(0, len(np.unique(cluster.labels_))
    ↪  + 1)
    hist, _ = np.histogram(cluster.labels_, bins=labels)
    hist = hist.astype('float')
    hist /= hist.sum()
    return hist

def k_means_image_colors(self, image,
↪  number_of_clusters=4):
    clusters = KMeans(n_clusters=number_of_clusters)
    clusters.fit(image)
    return clusters

def get_image_colors(self, image, number_of_clusters=2):
    clusters = self.k_means_image_colors(image,
    ↪  number_of_clusters=number_of_clusters)
    hist = self.create_histogram(clusters)
    combined = zip(hist, clusters.cluster_centers_)
    combined = sorted(combined, key=lambda x: x[0],
    ↪  reverse=True)

    to_return = []
    for index, row in enumerate(combined):
        comb = dict()
        comb["index"] = index + 1
        comb["%"] = round(row[0], 2)
        comb["RGB"] = self.rgb_color(row[1])
        to_return.append(comb)
    return to_return
```

Niestety algorytm okazał się mało wydajny obliczeniowo. Ponadto, raz na kilka uruchomień dawał różne wyniki dla tej samej próbki testowej - zwracane kolory nie występowały nawet na obrazie.

Rozwiązaniem tego problemu była całkowita zmiana podejścia do problemu. Nowy algorytm wyszukuje okręgi na obrazie, zapisuje ich pozycję i pobiera wartości koloru w obszarze konturu. Wykorzystując funkcję *HoughCircles* przy odpowiedniej parametryzacji (zapewnionej dzięki funkcjom kalibracyjnym) zwracana jest lista wszystkich pionków na planszy. Liczba pól i rozmiar planszy (użyto zmiany perspektywy do określonego wymiaru) to wartości stałe, dzięki temu możemy określić wartość pola stosując określony wzór. Kolor definiowany jest za pomocą pobrania wartości RGB z środka znalezionego konturu i sprawdzenia ich wartości w pliku konfiguracyjnym kolorów w celu nadania jednego z dwóch identyfikatorów *COLOR_TOP lub COLOR_BOT*. Tworzona jest lista słowników, w których zawarte są wszystkie informacje na temat poszczególnych pól (id, kolor, pionek, damka).

6.3 Usuwanie zbędnego tła

Tło podczas identyfikacji akcji na planszy jest zbędne, a nawet szkodliwe, dlatego należy je wyeliminować. Tło bywa zmienne, dlatego szukano jak najbardziej uniwersalnej metody.

W pierwszym podejściu przed uruchomieniem programu pojawiało się okno konfiguracyjne, w którym to użytkownik wybierał ręcznie poszczególne wierzchołki planszy (kolejno lewy górny, prawy górny, lewy dolny, prawy dolny). Następnie obraz miał zmienianą perspektywę oraz był przeskalowywany do stałych wymiarów. Algorytm był efektywny, jednak dostarczał pewnych zbędnych problemów technicznych, dlatego zastąpiono go w sposób automatyczny.

W ostatecznym podejściu za pomocą funkcji *cornerHarris* wykrywane są wszystkie rogi. Następnie specjalnie dopasowana funkcja wybiera znaczące wierzchołki planszy, dzięki temu zamiast ręcznie wybierać określone punkty otrzymujemy ten sam efekt autonomicznie. Następnie, tak jak poprzednio następuje zmiana perspektywy i przeskalowanie obrazu do określonego rozmiaru.

6.4 Wykrywanie damek

Kolejnym napotkanym problemem było wykrywanie damek, gdyż nie istnieje wbudowana funkcja do ich detekcji, tak jak w przypadku okręgów.

Każda korona posiada regularną strukturę. Zbudowana jest z siedmiu wierzchołków. Dodano dodatkową funkcję przy identyfikacji pionków, spraw-

dzającą czy na danym polu znajduje się figura specjalna. Wycinany jest obszar konturu, na który nałożona jest funkcja *cornerHarris*. Jeśli liczba wykrytych wierzchołków to siedem (dla danych parametrów wejściowych) to dany pionek jest damką,

6.5 Określenie kolorów pionków

Plansze do gry mogą się od siebie różnić kolorami pionków, dlatego w celu zapewnienia możliwie największej uniwersalności należało stworzyć rozwiązania do zbadania kolorów pionków na planszy.

Zanim dostępna będzie możliwość wyboru wartości makro kolorów na podstawie wartości RGB należy przeprowadzić wstępną konfigurację, w której zostaną policzone pionki na danej połowie planszy, sprawdzenie, czy na przynależnym polu znajduje się odpowiedni pionek, a następnie sprawdzenie tych wartości na kolejnych polach i zapisanie ich w postaci słownika w pliku konfiguracyjnym.

6.6 Zamiana współrzędnych pól szachownicy na numer pola w notacji PGN

Zapis ruchów w notacji *PGN* dla szachownicy 64 polowej numeruje pola szachownicy od 1 do 32. W logice wizualizacji szachownicy dla uproszczenia zastosowano dwuwymiarową listę, której pola możemy identyfikować po dwóch współrzędnych **x** oraz **y**. Zaistniała potrzeba wyznaczenia funkcji, której argumentami będą dwie współrzędne pola a wartością będzie numer pola w notacji PGN - oraz funkcja odwrotna.

7 Użytkowanie i testowanie systemu

7.1 Użytkowanie

W celu uruchomienia programu dla już istniejących danych należy:

1. Posiadać *pythona* w wersji +3.6
2. Utworzyć nowe środowisko wirtualne
3. Zainstalować biblitoteki z pliku *requirements.txt*
4. Uruchomić program *checkers_pygame.py*

Jeśli chcemy przetestować program dla własnych danych wejściowych musimy przygotować plansze zgodnie ze wzorcem i umieścić je w folderze *newtest2*.

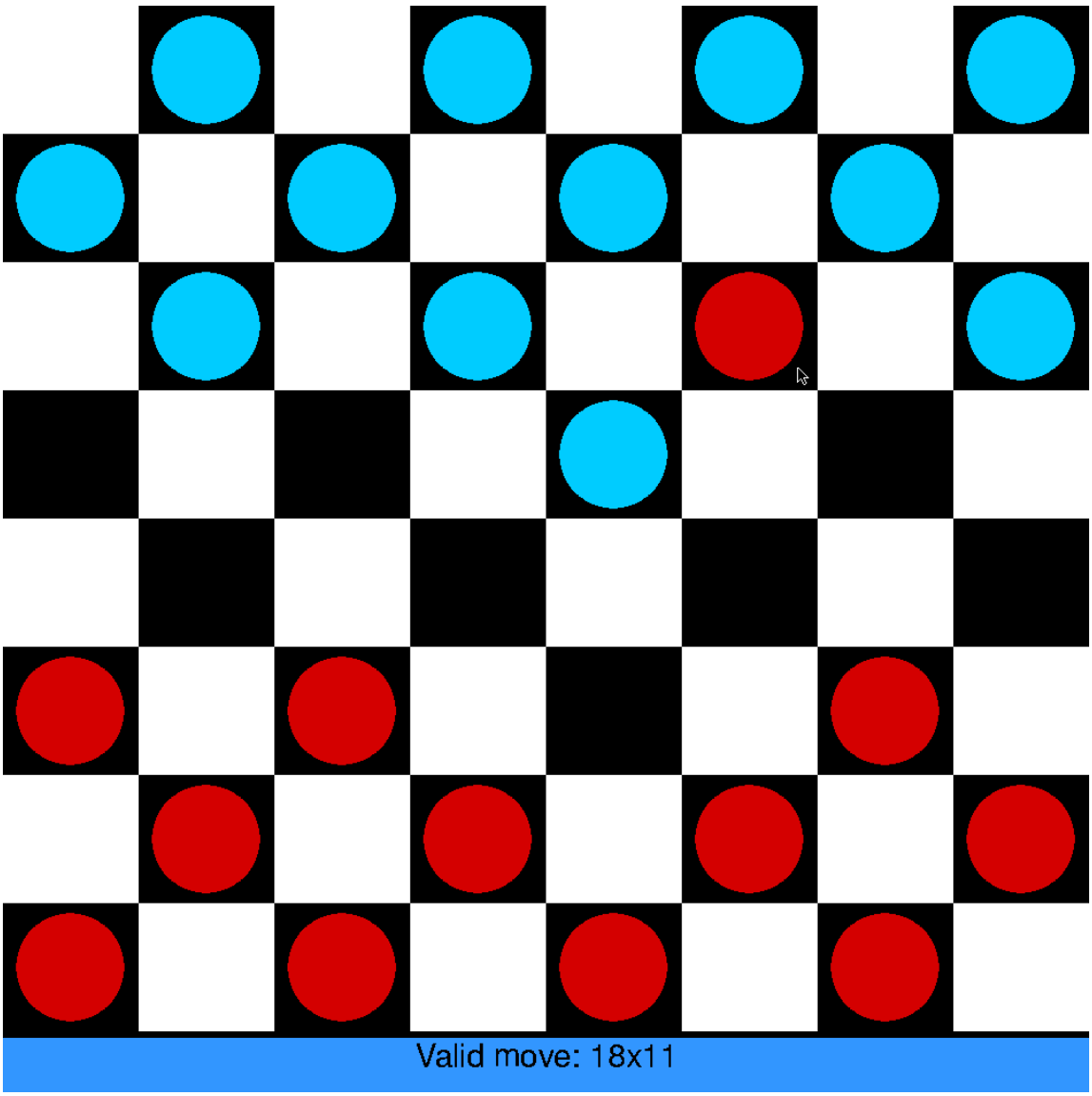
7.2 Testowanie

W ramach sprawdzenia reakcji systemu na błędne ruchy i inne scenariusze, przeprowadzono osiem testów:

1. Zbicie
2. Wielokrotne zbicie
3. Pominięcie zbicia
4. Pominięcie wielokrotnego zbicia
5. Zmiana pionka w damkę
6. Zbicie przez damkę
7. Poruszenie się pionka w złym kierunku
8. Poruszenie się pionka o za dużą liczbę pól

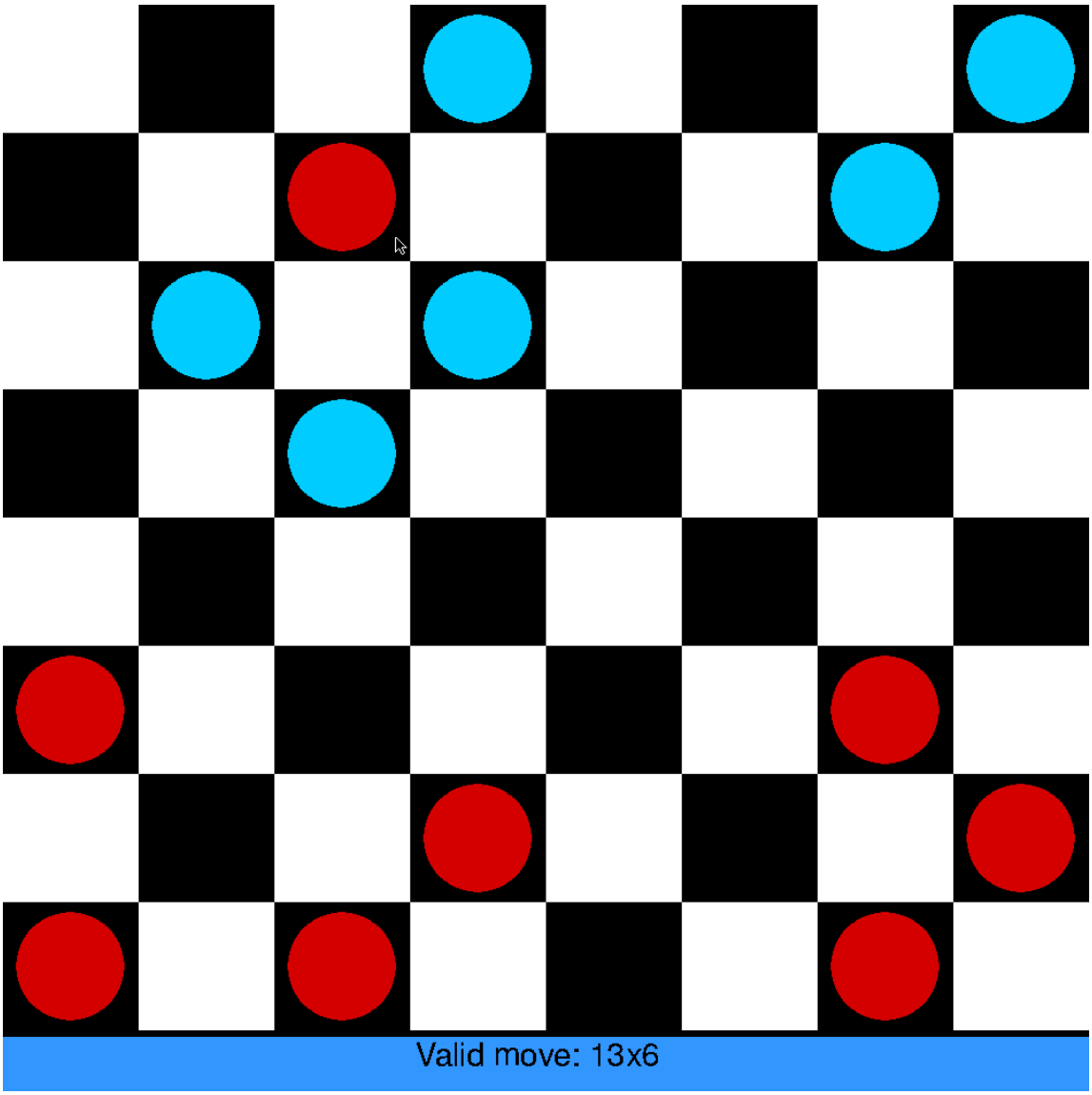
Poniżej przedstawiono odpowiedzi systemu na kolejne testy.

1. **Zbicie**
   Po zbiciu pionka system zatwierdza bieżący stan planszy, wyświetla go i informuje o wykonanym ruchu. Pionek zostaje przesunięty na dane pole, a zbity pionek zostaje po chwili usuwany z planszy. Pojawia się napis informujący o poprawności wykonanego ruchu. System informuje o numerach pól, które uczestniczyły w wykonanym ruchu - numer pola początkowego i końcowego.



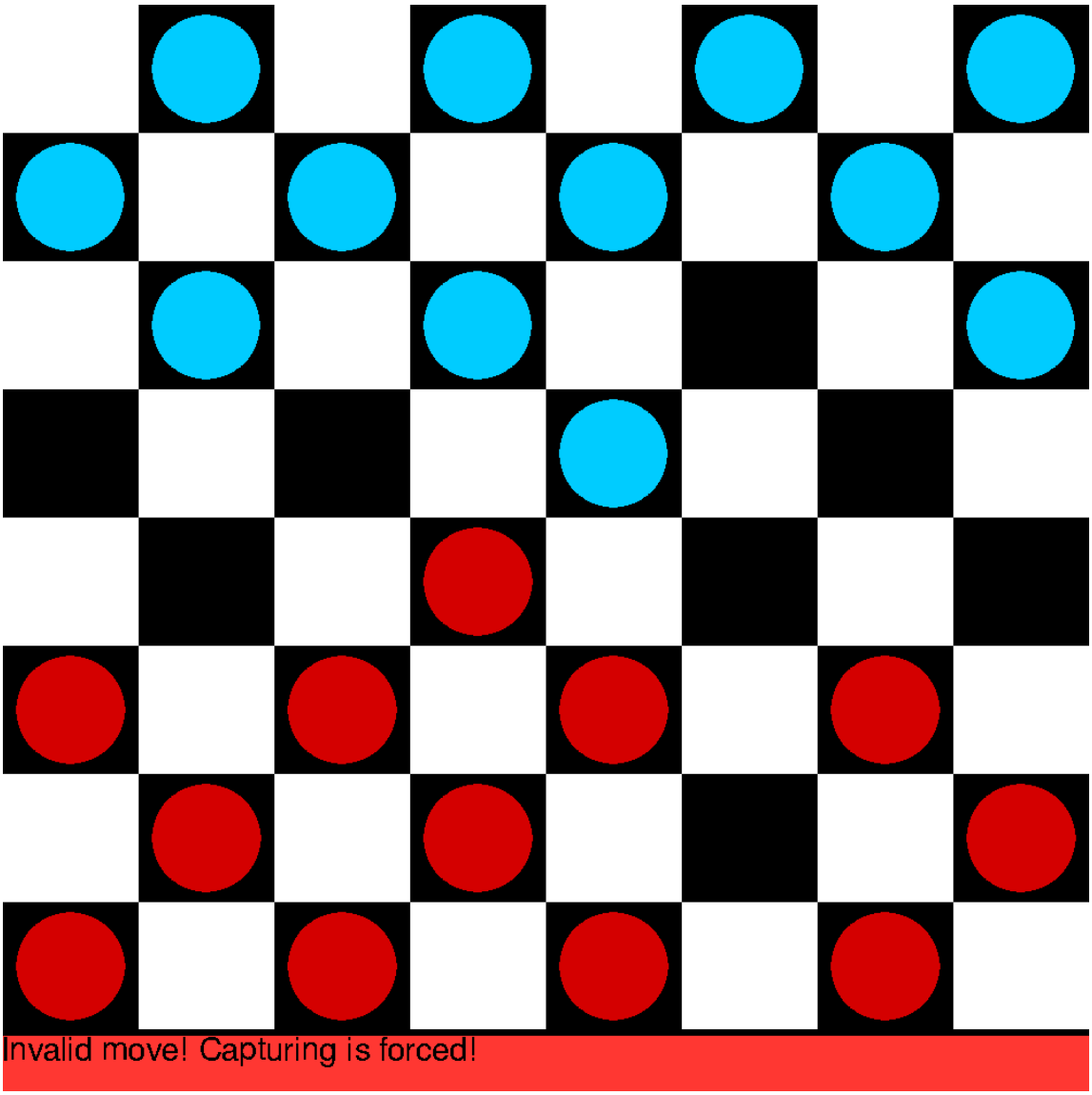
Rysunek 4: Reakcja systemu na zbicie.

2. **Wielokrotne zbicie**
   Każde ze zbić w wielokrotnym zbiciu pionka jest traktowane osobno jak pojedyncze zbicie. Dzięki temu, że system obserwuje ruchy konieczne do wykonania w danych okolicznościach, po wykonaniu jednego zbicia system nie oczekuje na ruch przeciwnego koloru. Wymaga wtedy wykonania kolejnego możliwego zbicia przez kolor, który wykonał pierwsze zbicie. Po wykonaniu kolejnego zbicia wyświetlane są informacje analogiczne do pojedynczego zbicia.



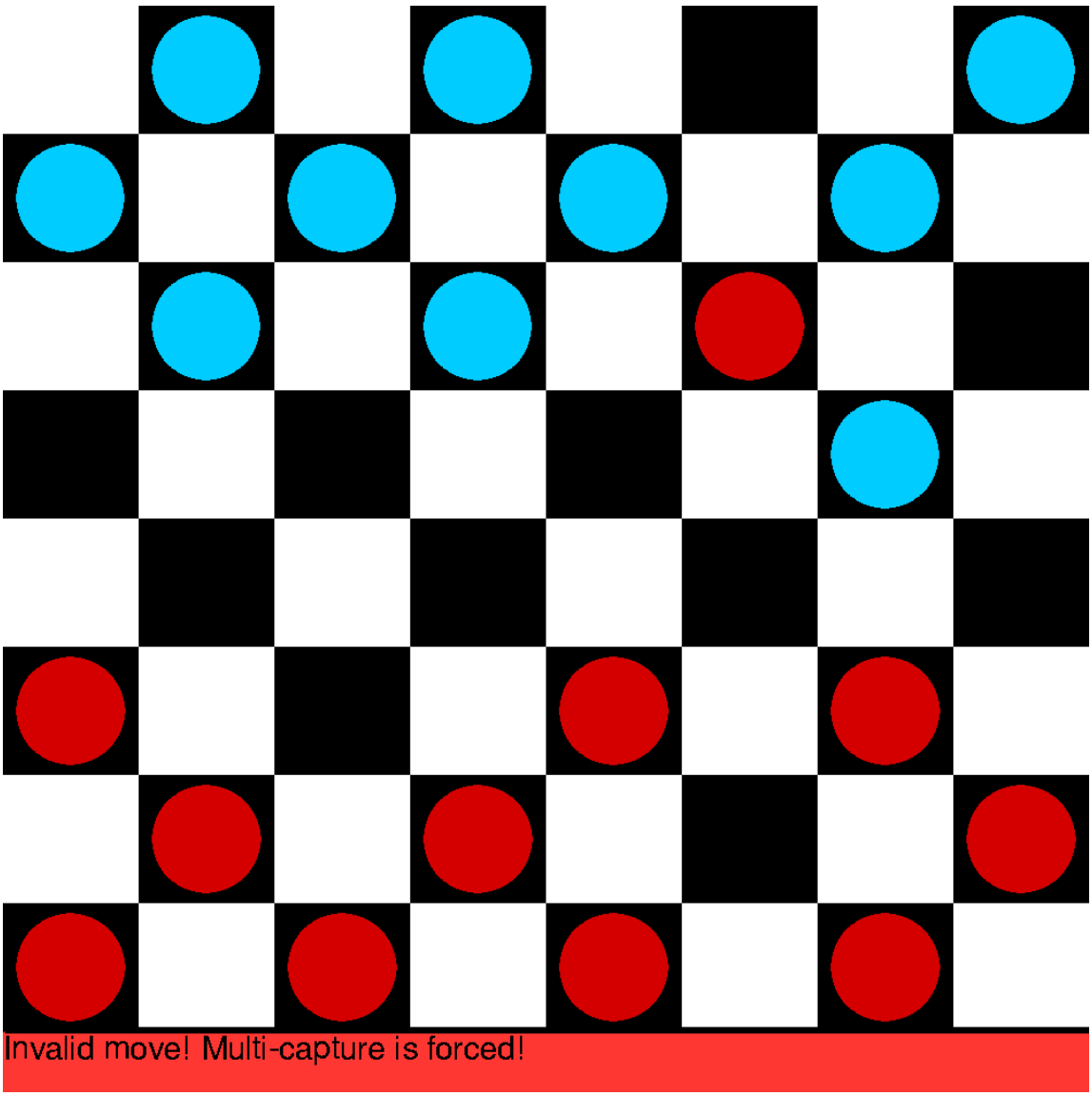
Rysunek 5: Reakcja systemu na wielokrotne zbicie.

3. **Pominięcie zbicia**
   System informuje o niewykonaniu wymaganego ruchu. Dzięki temu, że system monitoruje ruchy możliwe do wykonania w danej turze, wykonanie ruchu, który jest poprawny, ale nie powoduje możliwego do wykonania zbicia jest uznane zgodnie z zasadami gry warcaby za niepoprawne. System informuje o tym, że ruch jest niepoprawny przez wymagane wykonanie możliwego zbicia. Nie informuje jednak o numerach pól, które zbicie umożliwią, zmuszając gracza do myślenia.



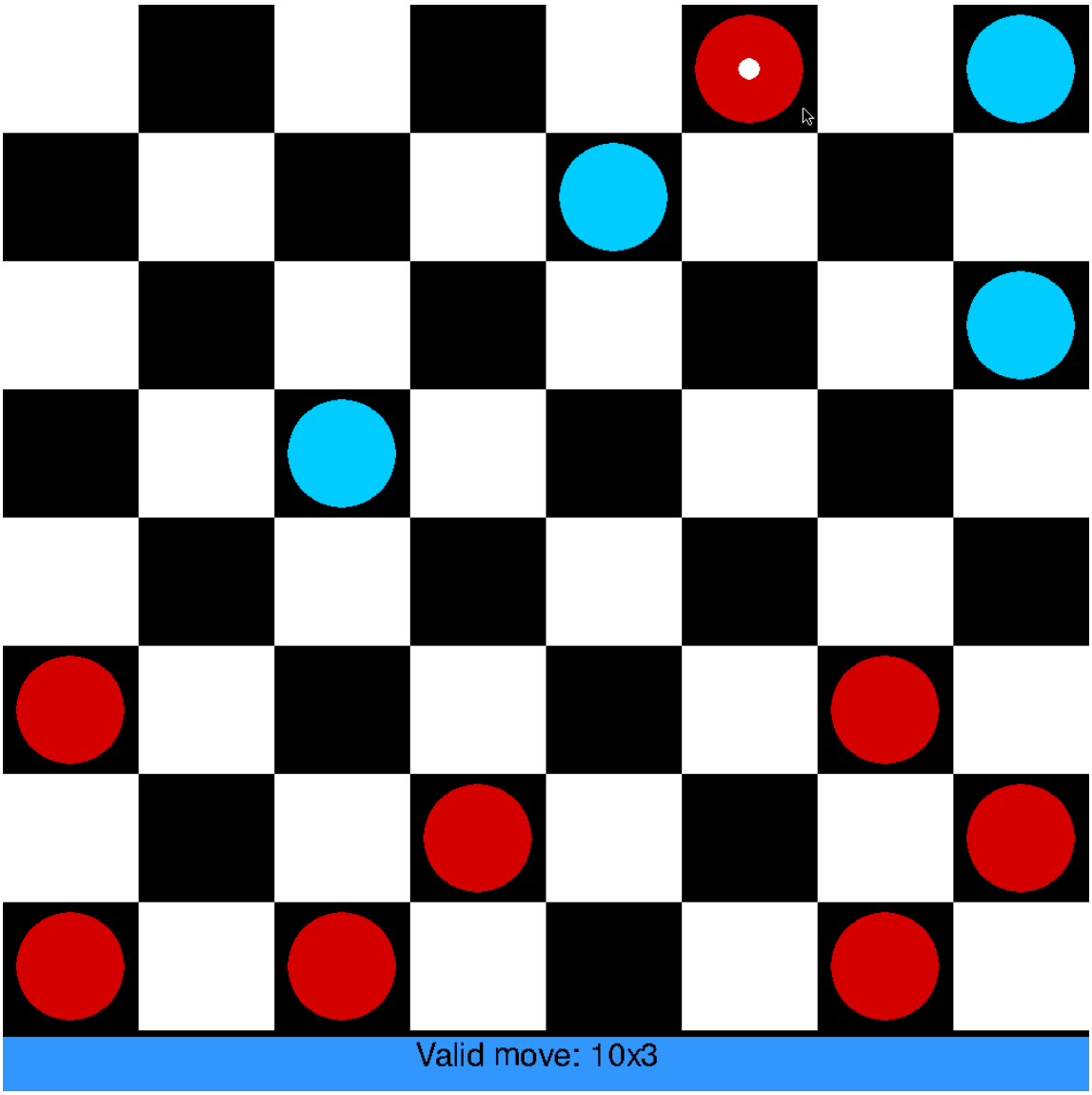
Rysunek 6: Reakcja systemu na pominięcie zbicia.

4. **Pominięcie wielokrotnego zbicia**
   System informuje o niewykonaniu wymaganego wielokrotnego zbicia. Adekwatnie do pominięcia pojedynczego zbicia, zakończenie ruchu na pojedynczym zbiciu i niewykonanie kolejnego zbicia skutkuje tym, że przeciwny kolor nie ma możliwości wykonania swojego ruchu. System wymusza na graczu wykonanie wszystkich możliwych wielokrotnych zbić.



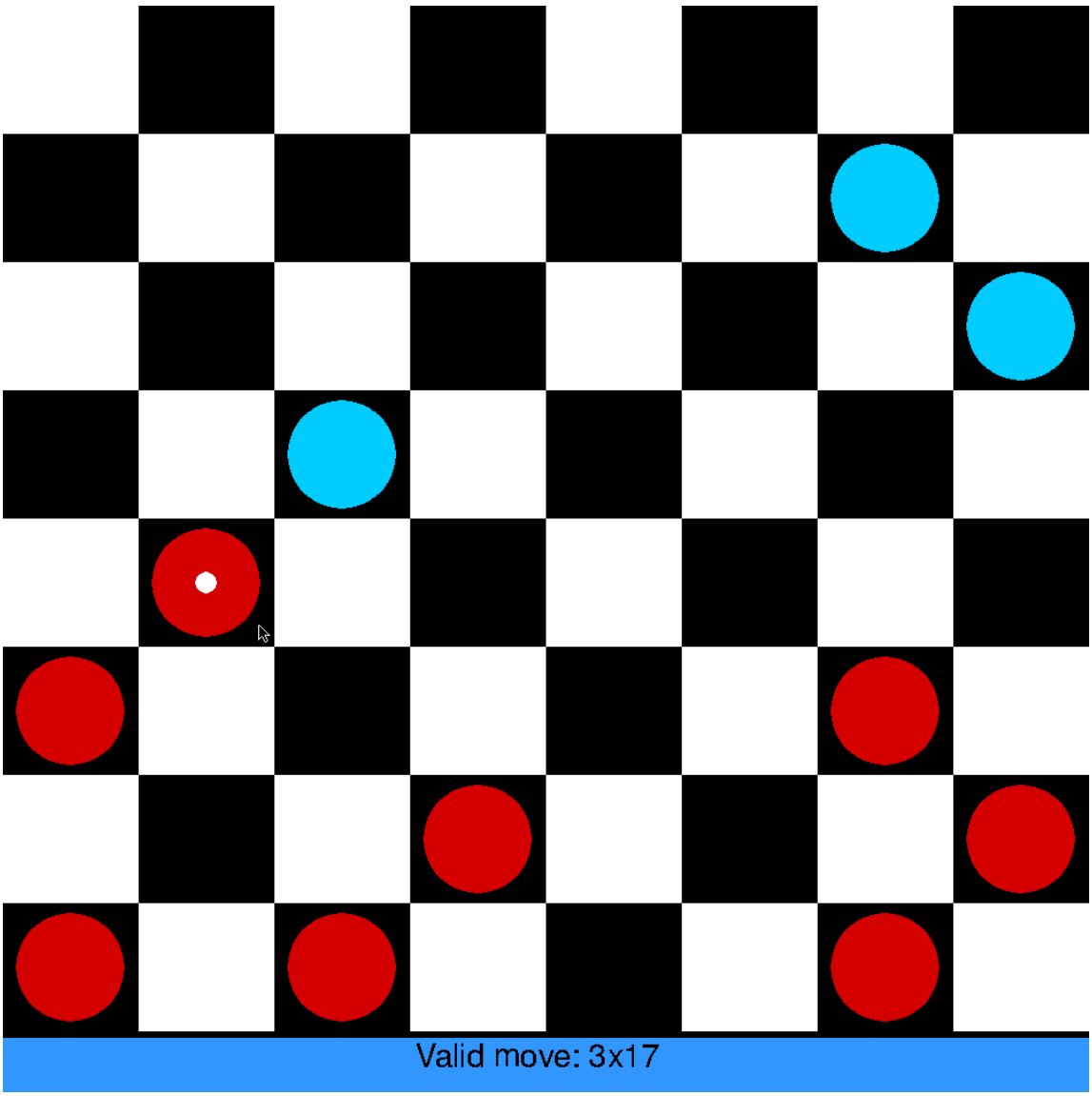
Rysunek 7: Reakcja systemu na pominięcie wielokrotnego zbicia.

5. **Zmiana pionka w damkę**
   Po przejściu na przeciwną stronę planszy pionek zostaje damką. Na graficznym przedstawieniu planszy zostaje on oznaczony kropką. Zgodnie z zasadami gry warcaby, po tym, jak pionek zostaje damką, następuje kolej przeciwnego koloru. Od teraz zmienia się liczba pól, które pionek może przejść podczas swojego ruchu.



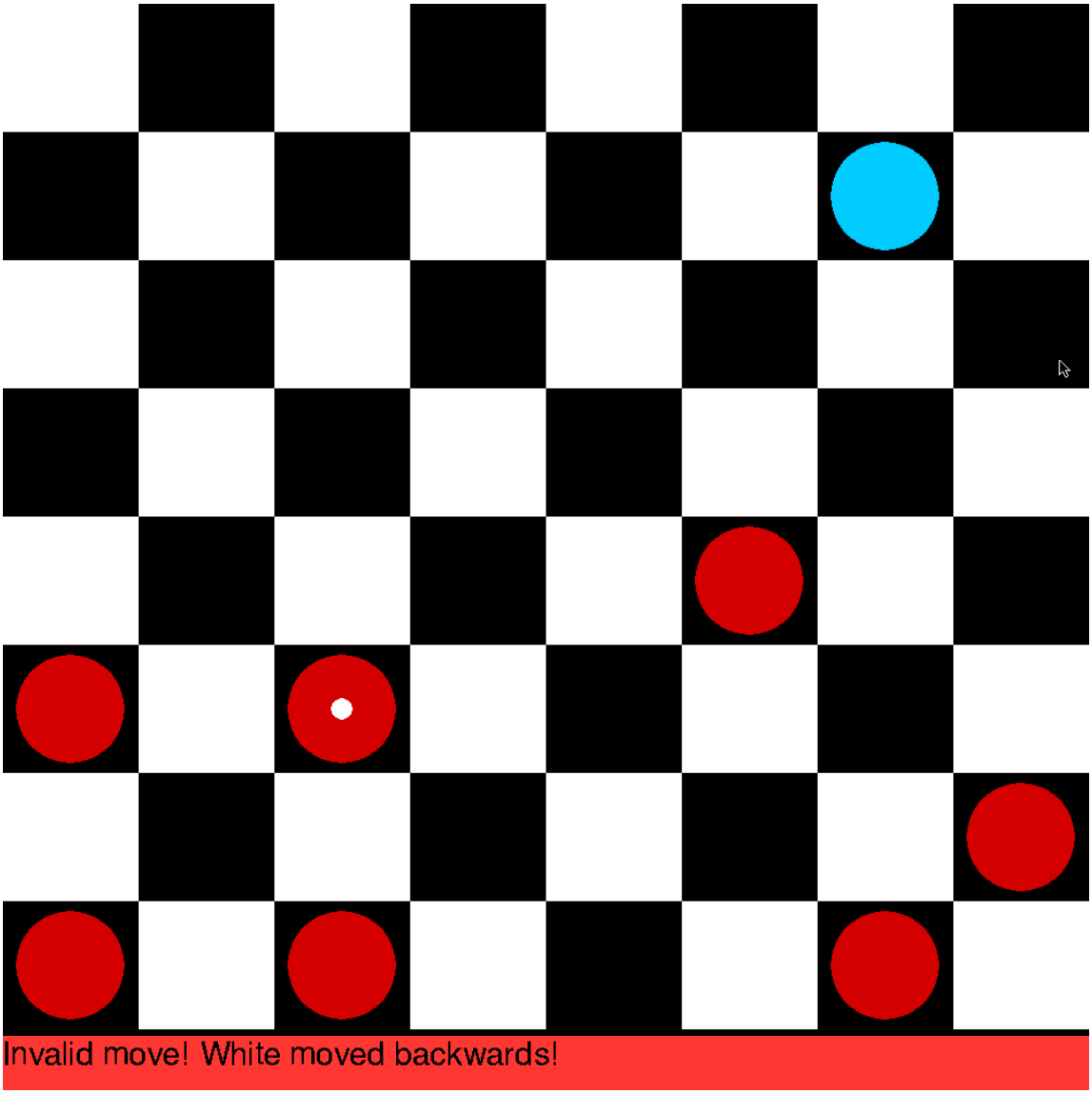
Rysunek 8: Reakcja systemu na zmianę pionka w damkę.

6. **Zbicie przez damkę**
   System reaguje tak jak na zwyczajne zbicie. Dzięki temu, że po zmianie pionka w damkę zmieniła się liczba dopuszczalnych pól, które pionek może przejść w jednym ruchu, system zatwierdza długie ruchy jako poprawne i dopuszcza zbicia tego typu. Analogicznie do zwykłego zbicia, zostaje wyświetlona informacja o poprawnym ruchu i numery pól początkowego i końcowego w ruchu.



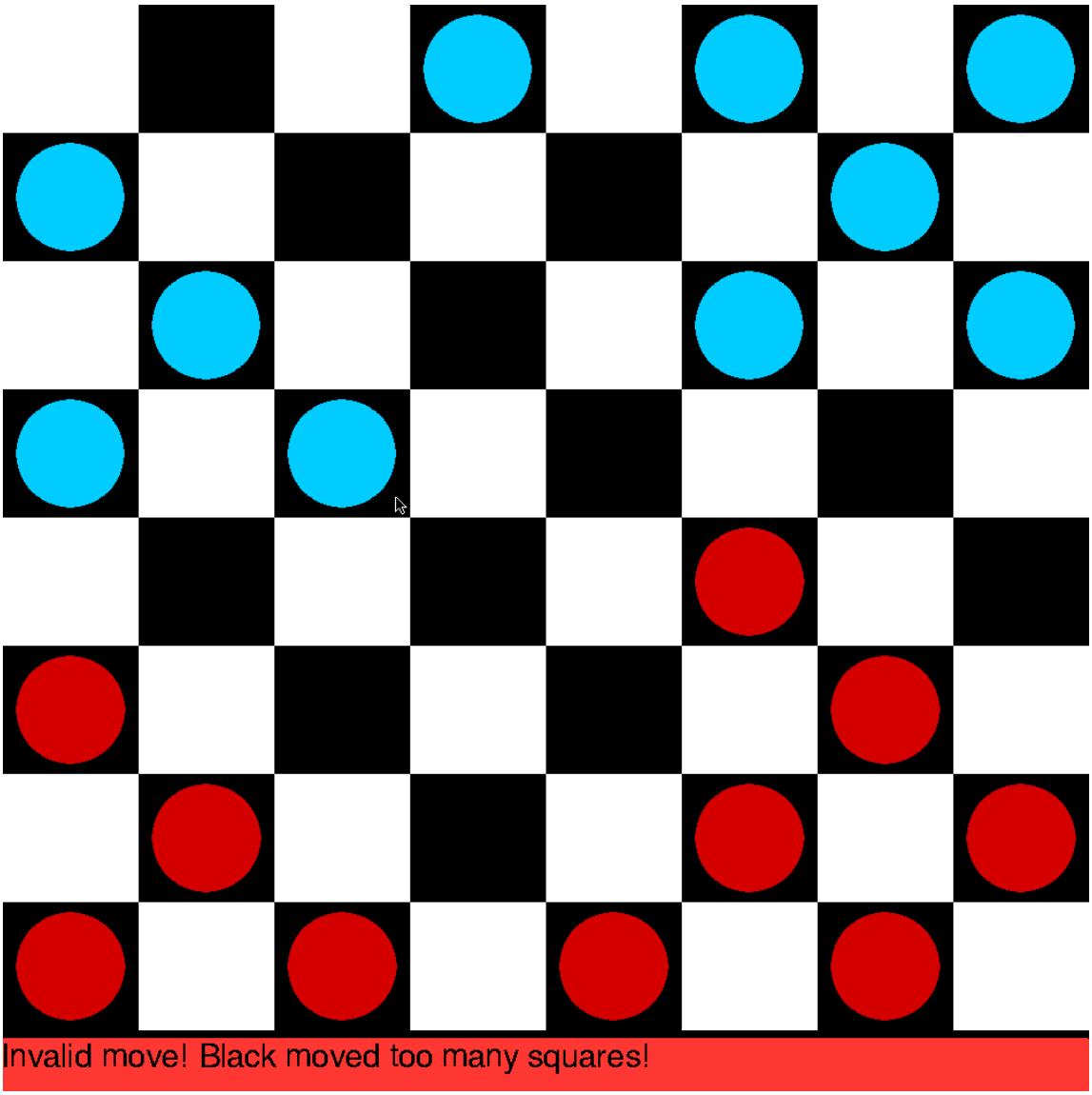
Rysunek 9: Reakcja systemu na zbicie przez damkę.

7. **Poruszenie się pionka w złym kierunku**
   Zgodnie z zasadami gry warcaby, pionek może poruszać się po przekątnych o jedno pole w przypadku ruchu bez zbicia lub o 2 pola w przypadku zbicia. Jeśli jest damką, może przejść dowolną liczbę pól, ale nadal muszą się one znajdować na przekątnej. W przypadku poruszenia się pionka w złym kierunku, na przykład o jedno pole w górę, system informuje o tym, że ruch jest niedozwolony.



Rysunek 10: Reakcja systemu na poruszenie się pionka w złym kierunku.

8. **Poruszenie się pionka o za dużą liczbę pól**
   Analogicznie do poruszenia się pionka w złym kierunku, system uznaje ruch za niedozwolony. Wyświetlana jest informacja o typie ruchu, czyli powodzie, dlaczego nie został on zatwierdzony - pionek przekroczył zbyt wiele pól planszy.



Rysunek 11: Reakcja systemu na poruszenie się pionka o za dużą liczbę pól.