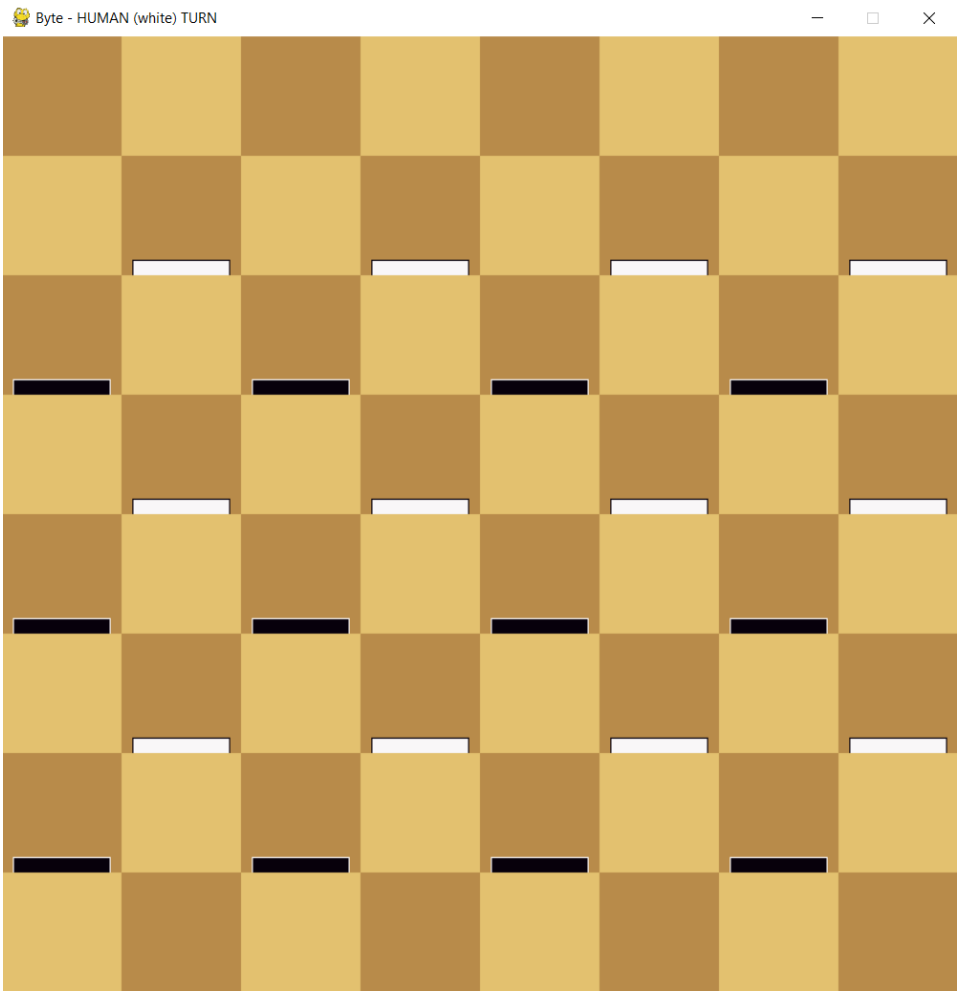
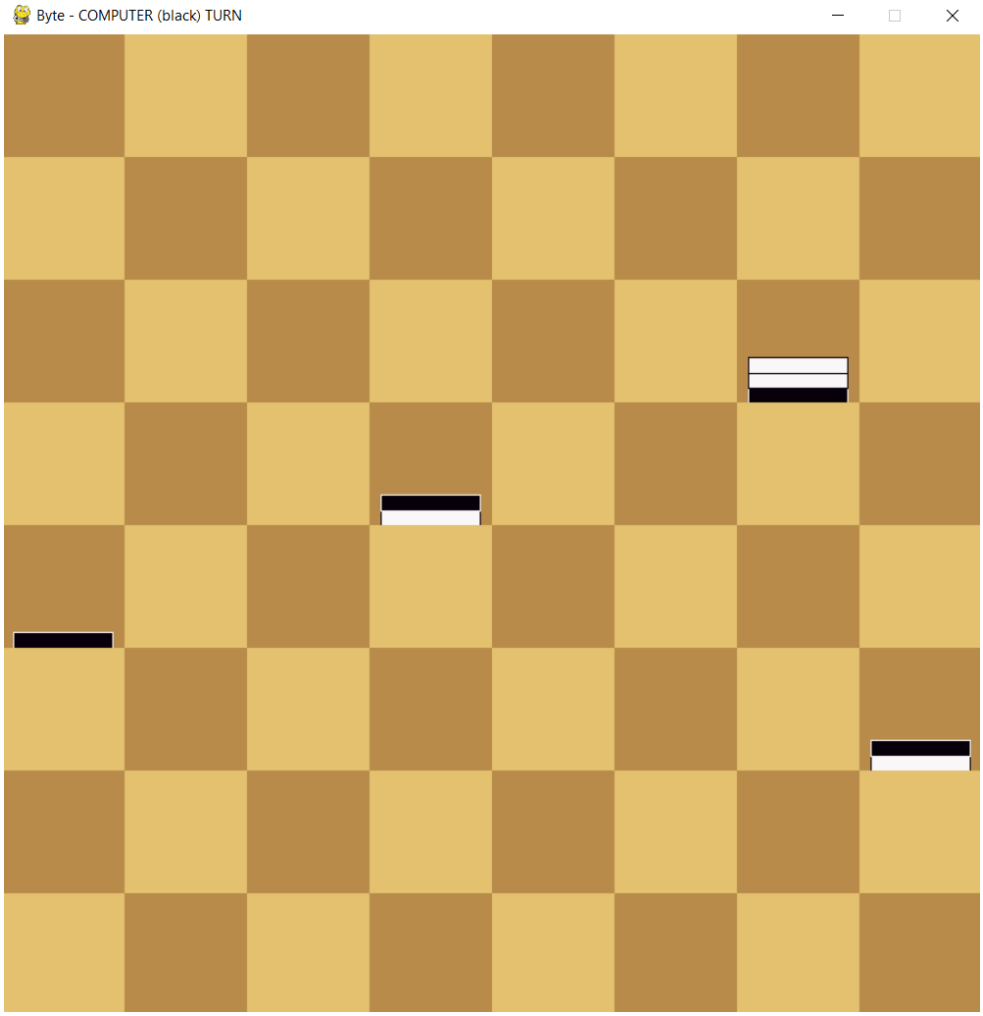
DOKUMENTACIJA – BYTE

Opis stanja

Tabla je predstavljena preko rečnika (dictionary). Ključ rečnika je tuple oblika `(row, column)` koji se odnosi na poziciju na tabli, dok je vrednost rečnika lista koja predstavlja stek tekona. Ključevi koji ispunjavaju uslov `(row % 2 == column % 2)` su uključeni u rečnik, što označava da će se stekovi nalaziti isključivo poljima iste boje (u našem slučaju, tamnije boje). Sa slike se vidi inicijalno stanje table, gde su prvi i poslednji redovi prazni, dok se u svim ostalim redovima nalazi po jedan token (stek sa jednim tokenom) na odgovarajućim poljima.



Inicijalno stanje



Prikaz stanja

Token.py

Svaki token ima svoje koordinate (red i kolonu) i nivo, što ukazuje na njegovu poziciju na tabli. Boja tokena se određuje na osnovu prosleđene vrednosti, a uzima se u obzir i njegove dimenzije (širina i visina). Tokeni imaju mogućnost da menjaju svoj status selekcije. Takođe, tokeni mogu biti pomereni na drugu poziciju, što je ključna funkcionalnost. Svaki token ima jedinstveni identifikator koji mu omogućava da se razlikuje od ostalih tokena.

```python
import color.colors as colors


7 usages
class Token:
    id = 0


    def __init__(self, row, column, color, width, height, level=1):
        self.row = row
        self.column = column
        self.level = level
        self.color = colors.BLACK if color == 0 else colors.WHITE
        self.width = width
        self.height = height
        self.selected = False
        self.border_thickness = 1
        self.id = Token.id

        Token.id += 1
```

```python
1 usage
def change_selected_status(self, status=None):
    if status is None:
        self.selected = not self.selected
    else:
        self.selected = status

1 usage
def move(self, dest_row, dest_column, dest_level):
    self.row = dest_row
    self.column = dest_column
    self.level = dest_level


def __repr__(self):
    return f'id:{self.id};row:{self.row};column:{self.column};level:{self.level}'
```

Board.py

Klasa Board služi kao centralni element za upravljanje igračkom tablom. Ona inicijalizuje tablu sa definisanim veličinama polja i postavlja osnovne parametre igre, uključujući boje polja i trenutnog igrača.
U klasi se nalaze funkcije za menjanje statusa selektovanih tokena, što omogućava igračima da interaktivno biraju i upravljaju svojim figurama.
Posebno važna je funkcija za inicijalizaciju table, gde se tokeni postavljaju na određene pozicije u zavisnosti od njihovih redova i kolona, stvarajući početni raspored igre.
Funkcija za isticanje kliknutog tokena omogućava igračima da identifikuju i selektuju token na osnovu njegove lokacije na ekranu, dok funkcija za pomeranje steka tokena omogućava kompleksne poteze gde igrači mogu premeštati svoje tokene na različite delove table. Ova funkcija takođe sadrži logiku za proveru validnosti poteza, uključujući provere da li su ciljana polja susedna i da li je potez u skladu sa pravilima igre.

```python
from Token import Token
import color.colors as colors
import color.fcolors as fcolors
from typing import List
from utils import are_neighbours


4 usages
class Board:


    def __init__(self, board_size, tile_size, current_player):
        self.board = {}
        self.board_size = board_size
        self.tile_size = tile_size
        self.playable_tiles = []
        self.board_light = colors.BEIGE
        self.board_dark = colors.BROWN
        self.selected_tokens: List[Token] = []
        self.current_player = current_player

    4 usages
    def change_selected_tokens_status(self):
        [selected_token.change_selected_status() for selected_token in self.selected_tokens]
```

```
1 usage
def initialize_board(self):
    token_width = int(self.tile_size * 0.8)
    token_height = self.tile_size // 8
    self.board = {
        (row, column): [] if row in (0, self.board_size - 1) else [
            Token(row, column, row % 2, token_width, token_height, level: 1),
            # Token(row, column, row % 2, token_width, token_height, 2),
            # Token(row, column, row % 2, token_width, token_height, 3),
        ]
        for row in range(self.board_size)
        for column in range(self.board_size)
        if (row % 2 == column % 2)
    }
    self.playable_tiles = list(self.board.keys())
```

```
def highlight_clicked_token(self, x, y):
    token_width = int(self.tile_size * 0.8)
    token_height = self.tile_size // 8
    tile_padding = (self.tile_size - token_width) / 2

    for stack in self.board.values():
        for i in range(len(stack)):
            token = stack[i]
            token_x = token.column * self.tile_size + tile_padding
            token_y = (token.row+1) * self.tile_size - token_height * token.level

            token_x_min = token_x
            token_x_max = token_x + token_width
            token_y_min = token_y
            token_y_max = token_y + token_height

            if token_x_min <= x <= token_x_max and token_y_min <= y <= token_y_max:
                # Abort if opposite player token has been attempted to select
                if self.current_player in ['h', 'H'] and token.color == colors.BLACK:
                    return
                if self.current_player in ['c', 'C'] and token.color == colors.WHITE:
                    return

                if self.selected_tokens and token == self.selected_tokens[0]:
                    self.change_selected_tokens_status()
                    self.selected_tokens = []
                    return
                self.change_selected_tokens_status()
                self.selected_tokens = []
                self.selected_tokens = [stack[j] for j in range(i, len(stack))]
                self.change_selected_tokens_status()
```

```python
def move_stack(self, row, column):
    # Check if token was selected
    selected_tokens_count = len(self.selected_tokens)
    if selected_tokens_count == 0:
        print(f'{fcolors.WARNING}No token was selected{fcolors.ENDC}')
        return

    # Check if row, column are playable tiles
    if (row, column) not in self.playable_tiles:
        print(f'{fcolors.FAIL}Tile is not playable{fcolors.ENDC}')
        return

    current_row = self.selected_tokens[0].row
    current_column = self.selected_tokens[0].column

    # Check if destination tile is the same as current tile
    if row == current_row and column == current_column:
        print(f'{fcolors.WARNING}Source and destination tiles are same{fcolors.ENDC}')
        return

    # Check if tiles are in neighbourhood
    if not are_neighbours( source_tile: (current_row, current_column),  destination_tile: (row, column)):
        print(f'{fcolors.FAIL}Destination tile is too far away{fcolors.ENDC}')
        return

    current_tile_tokens_count = len(self.board[(current_row, current_column)])
    destination_tile_tokens_count = len(self.board[(row, column)])
```

```python
# Check if resulting level is higher than the starting level
if self.selected_tokens[0].level >= destination_tile_tokens_count + 1:
    print(f'{fcolors.FAIL}You are attempting to move token to lower or equal level{fcolors.ENDC}')
    return

# Check if resulting stack would have more than 8 tokens
resulting_stack_size = selected_tokens_count + destination_tile_tokens_count
if resulting_stack_size > 8:
    print(f'{fcolors.FAIL}You are attempting to make stack of size {resulting_stack_size}{fcolors.ENDC}')
    return

# Update old tile in board dictionary
self.board[(current_row, current_column)] = self.board[(current_row, current_column)][:current_tile_tokens_count-selected_tokens_count]

# Update token objects
for token in self.selected_tokens:
    token.move(row, column, destination_tile_tokens_count + 1)
    destination_tile_tokens_count += 1

# Update new tile in board dictionary
self.board[(row, column)] = [*self.board[(row, column)], *self.selected_tokens]

# Deselect tokens
self.change_selected_tokens_status()
self.selected_tokens = []

# Check if stack of size 8 has been created
if len(self.board[(row, column)]) == 8:
    print(f'{fcolors.OKGREEN}Stack with size 8 was created{fcolors.ENDC}')
    # Delete the tokens
    self.board[(row, column)] = []

# Change current player
self.current_player = 'h' if self.current_player in ['c', 'C'] else 'c'
```

GUI.py

Klasa GUI je osmišljena da bude srce vizuelnog prikaza igre, koristeći Pygame za kreiranje i održavanje grafičkog interfejsa. Ova klasa upravlja prikazom igračke table, uključujući iscrtavanje pojedinačnih polja i tokena. Naslov prozora igre se ažurira da prikaže čiji je trenutno potez. Funkcije za crtanje tabele i tokena su posebno dizajnirane da vizualno predstave stanje igre u svakom trenutku, uključujući prikaz selektovanih tokena i njihovih pozicija na tabli. Ova klasa igra ključnu ulogu u omogućavanju igračima da na efikasan način prate tok igre i planiraju svoje poteze.

```
class GUI:


    def __init__(self, screen):
        self.screen = screen

    1 usage
    def draw_board(self, board: Board):
        # Define who's turn it is
        if board.current_player in ['h', 'H']:
            pygame.display.set_caption('Byte - HUMAN (white) TURN')
        else:
            pygame.display.set_caption('Byte - COMPUTER (black) TURN')

        # Draw board
        for row in range(board.board_size):
            for column in range(board.board_size):

                color = board.board_dark if (row + column) % 2 == 0 else board.board_light
                tile_rect = (column*board.tile_size, row*board.tile_size, board.tile_size, board.tile_size)
                pygame.draw.rect(self.screen, color, tile_rect)

                if (row, column) in board.playable_tiles:
                    stack = board.board[(row, column)]
                    self.draw_stack(stack, board.tile_size)
```

```
def draw_stack(self, stack, tile_size):
    for token in stack:
        self.draw_token(token, tile_size)

1 usage
def draw_token(self, token: Token, tile_size):
    tile_padding = (tile_size - token.width) / 2
    token_color = token.color
    token_size = (token.width, token.height)
    token_thickness = 3 if token.selected else token.border_thickness
    token_position_x = token.column*tile_size + tile_padding
    token_position_y = (token.row+1)*tile_size - token.height*(token.level)
    token_position = (token_position_x, token_position_y)

    border_rect = [
        token_position[0] - token_thickness,
        token_position[1] - token_thickness,
        token_size[0] + token_thickness * 2,
        token_size[1] + token_thickness * 2
    ]
    border_color = colors.GREEN if token.selected else colors.BLACK if token_color == colors.WHITE else colors.WHITE

    pygame.draw.rect(self.screen, border_color, border_rect)
    pygame.draw.rect(self.screen, token_color, rect: (token_position, token_size))
```

Utils.py

Funkcija get_clicked_tile_position omogućava precizno određivanje pozicije kliknutog polja na tabli, koristeći x i y koordinate klika miša. Korišćenjem granica svakog polja, funkcija uspešno identifikuje red i kolonu kliknutog polja, omogućavajući igri da odgovori na igračeve akcije. Druga funkcija, are_neighbours, proverava da li su dva polja susedna. Ovo je ključno za validaciju poteza u igri, jer osigurava da igrači mogu premeštati tokene samo na dozvoljena polja, održavajući igru fer i u skladu sa njenim pravilima.

```python
def get_clicked_tile_position(x, y, board_size, tile_size):
    row, column = [
        (row, column)
        for row in range(board_size)
        for column in range(board_size)
        if column*tile_size <= x <= (column+1)*tile_size and row*tile_size <= y <= (row+1)*tile_size
    ][0]
    return row, column


2 usages
def are_neighbours(source_tile, destination_tile):
    x_distance = abs(destination_tile[1] - source_tile[1])
    y_distance = abs(destination_tile[0] - source_tile[0])
    if x_distance > 1 or y_distance > 1:
        return False
    return True
```