INTERNATIONAL ISLAMIC UNIVERSITY CHITTAGONG

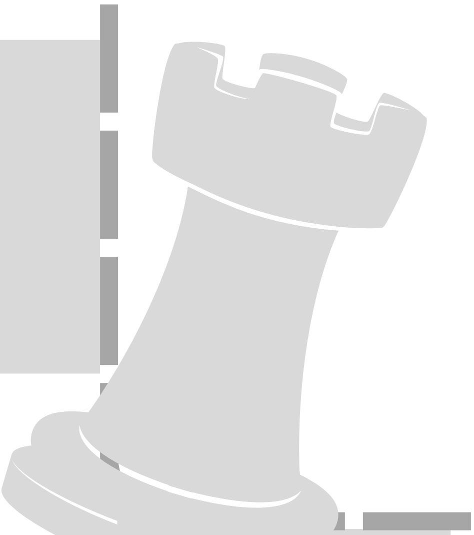
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Project Report
on
Chess - Mastering The Game

Course Title : Artificial Intelligence Lab
Course Code : CSE-3636

Submitted by

- Nahian Subah Ishma_C223286
- Rehnuma Tasneem_C223288
- Saima Kawsar_C223297



Submitted to

Sara Karim
Adjunct Lecturer, IIUC

TABLE OF CONTENTS

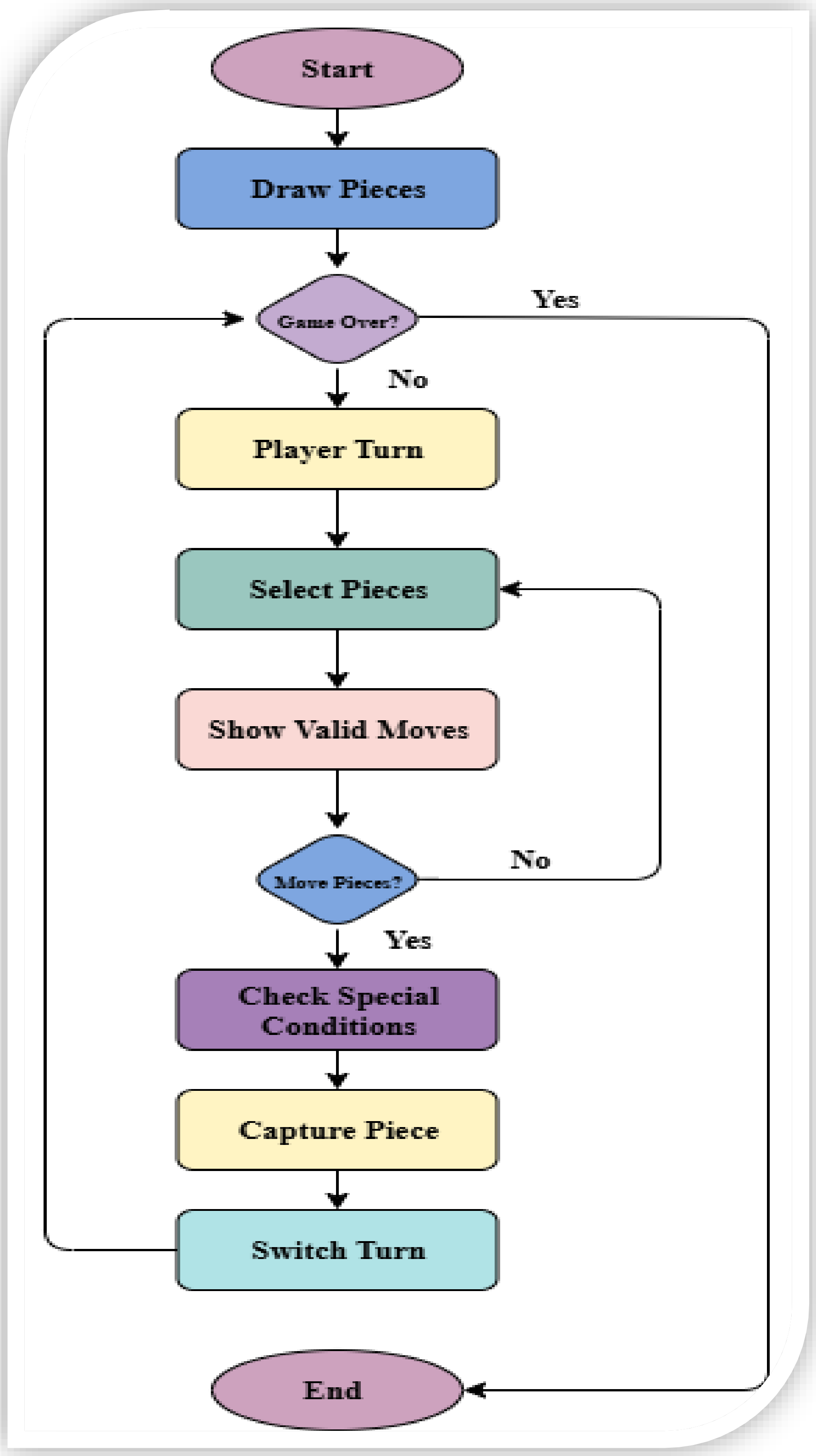
01. Introduction:

The game of Chess is a classic two-player strategy board game that has been played for centuries across cultures. It involves critical thinking, strategic planning, and tactical moves to outmaneuver the opponent. The game consists of a 64-square chessboard and two sets of pieces—white and black. Each player controls 16 pieces, including a king, queen, rooks, bishops, knights, and pawns, all of which move in distinct ways. The ultimate goal of the game is to checkmate the opponent's king, placing it under direct attack with no legal moves to escape. In this project, we aim to develop a digital version of the traditional Chess game for two players. This project emphasizes creating a user-friendly interface, accurate implementation of Chess rules, and an engaging gameplay experience. The game will provide players with real-time feedback, move validations, and an interactive environment to play Chess virtually.

02. Objective:

The primary objective of this project is to design and develop a fully functional two-player Chess game that strictly adheres to the official rules of Chess. The game aims to provide an interactive and user-friendly interface, allowing players to enjoy seamless gameplay. A crucial aspect of the development includes implementing accurate move validation and checkmate detection mechanisms to ensure fair and strategic play. The game will support real-time interaction and enable turn-based moves between two players, enhancing the overall gaming experience. Additionally, by simulating a competitive and strategic environment, the game is intended to improve players' logical thinking and problem-solving skills. To maintain a smooth and uninterrupted gaming experience, error handling mechanisms will also be integrated throughout the application.

03. Flowchart:



04. Code Implementation:

```
import pygame
pygame.init()

# Screen dimensions
WIDTH = 800
HEIGHT = 700
screen = pygame.display.set_mode([WIDTH, HEIGHT])
pygame.display.set_caption('Two-Player Pygame Chess!')

# Cell size and board dimensions
cell_size = 75
board_width = 8 * cell_size
board_height = 8 * cell_size

# Fonts
font = pygame.font.Font('freesansbold.ttf', 20)
medium_font = pygame.font.Font('freesansbold.ttf', 30)
big_font = pygame.font.Font('freesansbold.ttf', 40)

# Clock and FPS
timer = pygame.time.Clock()
fps = 60

# Game variables and images
white_pieces = ['rook', 'knight', 'bishop', 'queen', 'king', 'bishop', 'knight', 'rook',
                'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn']
white_locations = [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0),
                   (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1)]
black_pieces = ['rook', 'knight', 'bishop', 'queen', 'king', 'bishop', 'knight', 'rook',
                'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn']
black_locations = [(0, 7), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7),
                   (0, 6), (1, 6), (2, 6), (3, 6), (4, 6), (5, 6), (6, 6), (7, 6)]

captured_pieces_white = []
captured_pieces_black = []

# Turn step (0: white selects, 1: white moves, 2: black selects, 3: black moves)
turn_step = 0
selection = 100  # Currently selected piece
counter = 0
winner = ''
game_over = False
```

```
# load in game piece images (queen, king, rook, bishop, knight, pawn) x 2
black_queen = pygame.image.load('assets/images/black queen.png')
black_queen = pygame.transform.scale(black_queen, (60, 60))
black_queen_small = pygame.transform.scale(black_queen, (40, 40))
black_king = pygame.image.load('assets/images/black king.png')
black_king = pygame.transform.scale(black_king, (60, 60))
black_king_small = pygame.transform.scale(black_king, (40, 40))
black_rook = pygame.image.load('assets/images/black rook.png')
black_rook = pygame.transform.scale(black_rook, (60, 60))
black_rook_small = pygame.transform.scale(black_rook, (40, 40))
black_bishop = pygame.image.load('assets/images/black bishop.png')
black_bishop = pygame.transform.scale(black_bishop, (60, 60))
black_bishop_small = pygame.transform.scale(black_bishop, (40, 40))
black_knight = pygame.image.load('assets/images/black knight.png')
black_knight = pygame.transform.scale(black_knight, (60, 60))
black_knight_small = pygame.transform.scale(black_knight, (40, 40))
black_pawn = pygame.image.load('assets/images/black pawn.png')
black_pawn = pygame.transform.scale(black_pawn, (45, 45))
black_pawn_small = pygame.transform.scale(black_pawn, (40, 40))
white_queen = pygame.image.load('assets/images/white queen.png')
white_queen = pygame.transform.scale(white_queen, (60, 60))
white_queen_small = pygame.transform.scale(white_queen, (40, 40))
white_king = pygame.image.load('assets/images/white king.png')
white_king = pygame.transform.scale(white_king, (60, 60))
white_king_small = pygame.transform.scale(white_king, (40, 40))
white_rook = pygame.image.load('assets/images/white rook.png')
white_rook = pygame.transform.scale(white_rook, (60, 60))
white_rook_small = pygame.transform.scale(white_rook, (40, 40))
white_bishop = pygame.image.load('assets/images/white bishop.png')
white_bishop = pygame.transform.scale(white_bishop, (60, 60))
white_bishop_small = pygame.transform.scale(white_bishop, (40, 40))
white_knight = pygame.image.load('assets/images/white knight.png')
white_knight = pygame.transform.scale(white_knight, (60, 60))
white_knight_small = pygame.transform.scale(white_knight, (40, 40))
white_pawn = pygame.image.load('assets/images/white pawn.png')
white_pawn = pygame.transform.scale(white_pawn, (45, 45))
white_pawn_small = pygame.transform.scale(white_pawn, (40, 40))
white_images = [white_pawn, white_queen, white_king, white_knight, white_rook, white_bishop]
small_white_images = [white_pawn_small, white_queen_small, white_king_small,
white_knight_small,
                      white_rook_small, white_bishop_small]
black_images = [black_pawn, black_queen, black_king, black_knight, black_rook, black_bishop]
small_black_images = [black_pawn_small, black_queen_small, black_king_small,
black_knight_small,
                      black_rook_small, black_bishop_small]

piece_list = ['pawn', 'queen', 'king', 'knight', 'rook', 'bishop']
```

```
# check variables/ flashing counter
counter = 0
winner = ''
game_over = False

# Draw the chessboard
def draw_board():
    light_color = (245, 222, 179)
    dark_color = (139, 69, 19)
    for row in range(8):
        for col in range(8):
            color = light_color if (row + col) % 2 == 0 else dark_color
            pygame.draw.rect(screen, color, [col * cell_size, row * cell_size, cell_size, cell_size])

# Draw status area
  pygame.draw.rect(screen, (139, 69, 19), [0, board_height, WIDTH, HEIGHT - board_height])  #
  pygame.draw.rect(screen, (255, 215, 0), [0, board_height, WIDTH, HEIGHT - board_height], 3)  #
  pygame.draw.rect(screen, (255, 215, 0), [board_width, 0, WIDTH - board_width, HEIGHT], 3)  #

    # Status text
    status_text = ['White: Select a Piece to Move!', 'White: Select a Destination!',
                   'Black: Select a Piece to Move!', 'Black: Select a Destination!']
    text_surface = big_font.render(status_text[turn_step], True, (255, 255, 255))  # White text
    screen.blit(text_surface, (10, board_height + 20))
    for i in range(9):
        pygame.draw.line(screen, (0, 0, 0), (0, cell_size * i), (board_width, cell_size * i), 2)  # Black
        pygame.draw.line(screen, (0, 0, 0), (cell_size * i, 0), (cell_size * i, board_height), 2)  # Black
    screen.blit(medium_font.render('FORFEIT', True, 'white'), (620, 630))

# Draw pieces
def draw_pieces():
    for i in range(len(white_pieces)):
        index = piece_list.index(white_pieces[i])
        if white_pieces[i] == 'pawn':
screen.blit(white_pawn, (white_locations[i][0] * cell_size + 12, white_locations[i][1] * cell_size +
20))
        else: screen.blit(white_images[index], (white_locations[i][0] * cell_size + 10,
white_locations[i][1] * cell_size + 10))
        if turn_step < 2 and selection == i:
pygame.draw.rect(screen, 'red', [white_locations[i][0] * cell_size + 1, white_locations[i][1] * cell_size
+ 1,
cell_size, cell_size], 2)
    for i in range(len(black_pieces)):
        index = piece_list.index(black_pieces[i])
        if black_pieces[i] == 'pawn':
screen.blit(black_pawn, (black_locations[i][0] * cell_size + 12, black_locations[i][1] * cell_size + 20))
        else: screen.blit(black_images[index], (black_locations[i][0] * cell_size + 10,
black_locations[i][1] * cell_size + 10))
        if turn_step >= 2 and selection == i:
            pygame.draw.rect(screen, 'blue', [black_locations[i][0] * cell_size + 1, black_locations[i][1] *
cell_size + 1,
 cell_size, cell_size], 2)
```

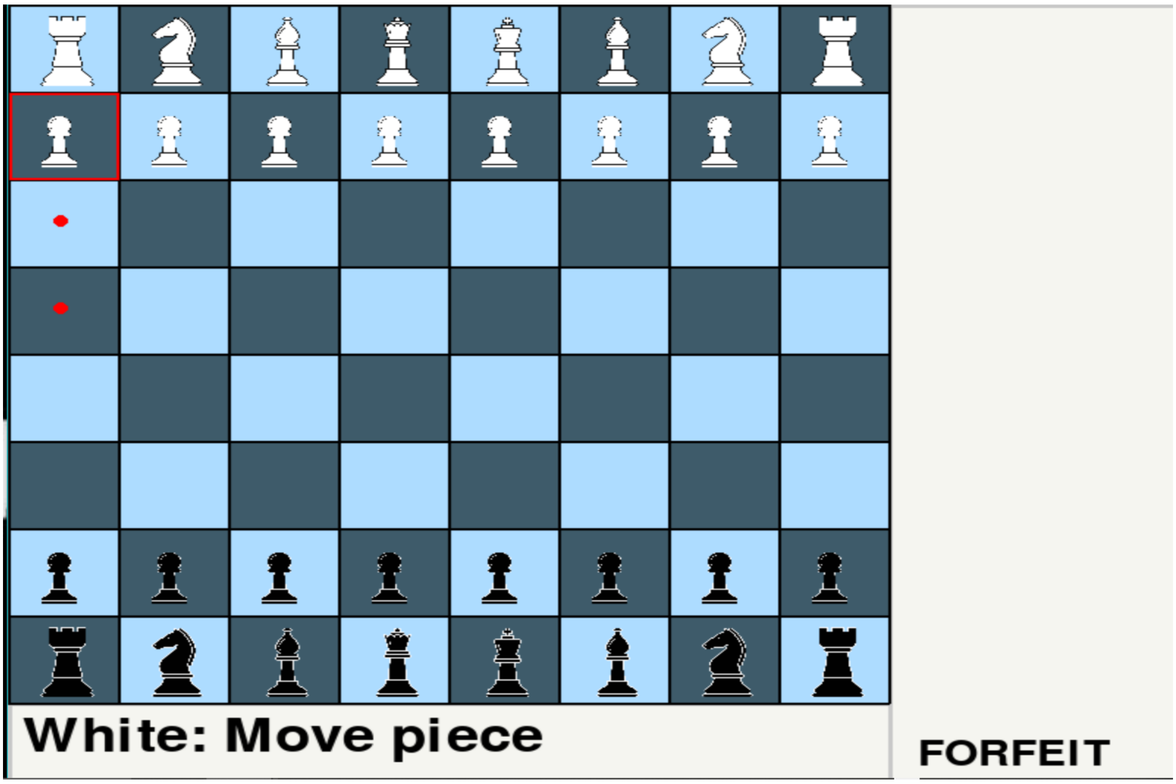
```
# function to check all pieces valid options on board
def check_options(pieces, locations, turn):
    moves_list = []
    all_moves_list = []
    for i in range((len(pieces))):
        location = locations[i]
        piece = pieces[i]
        if piece == 'pawn':
            moves_list = check_pawn(location, turn)
        elif piece == 'rook':
            moves_list = check_rook(location, turn)
        elif piece == 'knight':
            moves_list = check_knight(location, turn)
        elif piece == 'bishop':
            moves_list = check_bishop(location, turn)
        elif piece == 'queen':
            moves_list = check_queen(location, turn)
        elif piece == 'king':
            moves_list = check_king(location, turn)
        all_moves_list.append(moves_list)
    return all_moves_list

# check king valid moves
def check_king(position, color):
    moves_list = []
    if color == 'white':
        enemies_list = black_locations
        friends_list = white_locations
    else:
        friends_list = black_locations
        enemies_list = white_locations

  # 8 squares to check for kings, they can go one square any direction
    targets = [(1, 0), (1, 1), (1, -1), (-1, 0), (-1, 1), (-1, -1), (0, 1), (0, -1)]
    for i in range(8):
        target = (position[0] + targets[i][0], position[1] + targets[i][1])
        if target not in friends_list and 0 <= target[0] <= 7 and 0 <= target[1] <= 7:
            moves_list.append(target)
    return moves_list

# check queen valid moves
def check_queen(position, color):
    moves_list = check_bishop(position, color)
    second_list = check_rook(position, color)
    for i in range(len(second_list)):
        moves_list.append(second_list[i])
    return moves_list

# check bishop moves
def check_bishop(position, color):
    moves_list = []
    if color == 'white':
        enemies_list = black_locations
        friends_list = white_locations
    else:
        friends_list = black_locations
        enemies_list = white_locations
    for i in range(4):  # up-right, up-left, down-right, down-left
        path = True
        chain = 1
        if i == 0:
            x = 1
            y = -1
        elif i == 1:
            x = -1
            y = -1
        elif i == 2:
            x = 1
            y = 1
        else:
            x = -1
            y = 1
        while path:
            if (position[0] + (chain * x), position[1] + (chain * y)) not in friends_list and \
                    0 <= position[0] + (chain * x) <= 7 and 0 <= position[1] + (chain * y) <= 7:
                moves_list.append((position[0] + (chain * x), position[1] + (chain * y)))
                if (position[0] + (chain * x), position[1] + (chain * y)) in enemies_list:
                    path = False
                chain += 1
            else:
                path = False
    return moves_list

# check rook moves
def check_rook(position, color):
    moves_list = []
    if color == 'white':
        enemies_list = black_locations
        friends_list = white_locations
    else:
        friends_list = black_locations
        enemies_list = white_locations
    for i in range(4):  # down, up, right, left
        path = True
        chain = 1
        if i == 0:
            x = 0
            y = 1
        elif i == 1:
            x = 0
            y = -1
        elif i == 2:
            x = 1
            y = 0
        else:
            x = -1
            y = 0
```

```
while path:
        if (position[0] + (chain * x), position[1] + (chain * y)) not in friends_list and \
                0 <= position[0] + (chain * x) <= 7 and 0 <= position[1] + (chain * y) <= 7:
            moves_list.append((position[0] + (chain * x), position[1] + (chain * y)))
            if (position[0] + (chain * x), position[1] + (chain * y)) in enemies_list:
                path = False
            chain += 1
        else:
            path = False
    return moves_list

# check valid pawn moves
def check_pawn(position, color):
    moves_list = []
    if color == 'white':
        if (position[0], position[1] + 1) not in white_locations and \
                (position[0], position[1] + 1) not in black_locations and position[1] < 7:
            moves_list.append((position[0], position[1] + 1))
        if (position[0], position[1] + 2) not in white_locations and \
                (position[0], position[1] + 2) not in black_locations and position[1] == 1:
            moves_list.append((position[0], position[1] + 2))
        if (position[0] + 1, position[1] + 1) in black_locations:
            moves_list.append((position[0] + 1, position[1] + 1))
        if (position[0] - 1, position[1] + 1) in black_locations:
            moves_list.append((position[0] - 1, position[1] + 1))
    else:
        if (position[0], position[1] - 1) not in white_locations and \
                (position[0], position[1] - 1) not in black_locations and position[1] > 0:
            moves_list.append((position[0], position[1] - 1))
        if (position[0], position[1] - 2) not in white_locations and \
                (position[0], position[1] - 2) not in black_locations and position[1] == 6:
            moves_list.append((position[0], position[1] - 2))
        if (position[0] + 1, position[1] - 1) in white_locations:
            moves_list.append((position[0] + 1, position[1] - 1))
        if (position[0] - 1, position[1] - 1) in white_locations:
            moves_list.append((position[0] - 1, position[1] - 1))
    return moves_list

# check valid knight moves
def check_knight(position, color):
    moves_list = []
    if color == 'white':
        enemies_list = black_locations
        friends_list = white_locations
    else:
        friends_list = black_locations
        enemies_list = white_locations
```

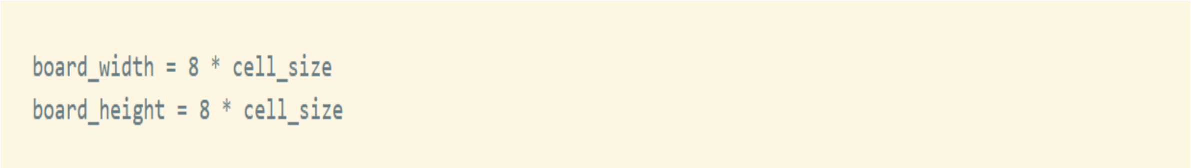
```
 # 8 squares to check for knights, they can go two squares in one direction and one in another
    targets = [(1, 2), (1, -2), (2, 1), (2, -1), (-1, 2), (-1, -2), (-2, 1), (-2, -1)]
    for i in range(8):
        target = (position[0] + targets[i][0], position[1] + targets[i][1])
        if target not in friends_list and 0 <= target[0] <= 7 and 0 <= target[1] <= 7:
            moves_list.append(target)
    return moves_list
def check_valid_moves():
    if turn_step < 2:
        options_list = white_options
    else:
        options_list = black_options
    valid_options = options_list[selection]
    return valid_options

# check for valid moves for just selected piece
def check_valid_moves():
    if turn_step < 2:
        options_list = white_options
    else:
        options_list = black_options
    valid_options = options_list[selection]
    return valid_options

# draw valid moves on screen
def draw_valid(moves):
    if turn_step < 2:
        color = 'red'
    else: color = 'blue'
    for i in range(len(moves)):
        pygame.draw.circle(screen, color, (moves[i][0] * cell_size + 36, moves[i][1] * cell_size + 36), 5)
valid_moves = []

# draw captured pieces on side of screen
def draw_captured():
    for i in range(len(captured_pieces_white)):
        captured_piece = captured_pieces_white[i]
        index = piece_list.index(captured_piece)
        screen.blit(small_black_images[index], (825, 5 + 50 * i))
    for i in range(len(captured_pieces_black)):
        captured_piece = captured_pieces_black[i]
        index = piece_list.index(captured_piece)
        screen.blit(small_white_images[index], (925, 5 + 50 * i))

# draw a flashing square around king if in check
def draw_check():
    global counter, white_options, black_options, cell_size, board_width, board_height
    if turn_step < 2:
        if 'king' in white_pieces:
            king_index = white_pieces.index('king')
            king_location = white_locations[king_index]
            for i in range(len(black_options)):
                if king_location in black_options[i]:
                    if counter < 15:
```

```
            pygame.draw.rect(screen, 'dark red', [white_locations[king_index][0] * cell_size + 1,
white_locations[king_index][1] * cell_size + 1, cell_size, cell_size], 5)
else:
    if 'king' in black_pieces:
        king_index = black_pieces.index('king')
        king_location = black_locations[king_index]
        for i in range(len(white_options)):
            if king_location in white_options[i]:
                if counter < 15:
                pygame.draw.rect(screen, 'dark blue', [black_locations[king_index][0] * cell_size + 1,
  black_locations[king_index][1] * cell_size + 1, cell_size, cell_size], 5)

# draw captured pieces on side of screen
def draw_captured():
    for i in range(len(captured_pieces_white)):
        captured_piece = captured_pieces_white[i]
        index = piece_list.index(captured_piece)
        screen.blit(small_black_images[index], (625, 5 + 50 * i))
    for i in range(len(captured_pieces_black)):
        captured_piece = captured_pieces_black[i]
        index = piece_list.index(captured_piece)
        screen.blit(small_white_images[index], (725, 5 + 50 * i))
def draw_game_over():
    pygame.draw.rect(screen, 'black', [200, 200, 400, 70])
    screen.blit(font.render(f'{winner} won the game!', True, 'white'), (210, 210))
    screen.blit(font.render(f'Press ENTER to Restart!', True, 'white'), (210, 240))

# Main game loop
black_options = check_options(black_pieces, black_locations, 'black')
white_options = check_options(white_pieces, white_locations, 'white')
run = True
while run:
    timer.tick(fps)
    if counter < 30:
        counter += 1e:
        counter = 0
    screen.fill((139, 69, 19))  # SaddleBrown background
    draw_board()
    draw_pieces()
    draw_captured()
    draw_check()
    if selection != 100:
        valid_moves = check_valid_moves()
        draw_valid(valid_moves)

# Event handling
    for event in pygame.event.get():
        if event.type == pygame.QUIT:      run = False
        if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1 and not game_over:
            x_coord = event.pos[0] // cell_size
            y_coord = event.pos[1] // cell_size
            click_coords = (x_coord, y_coord)
            if turn_step <= 1:
                if click_coords == (8, 8) or click_coords == (9, 8):
                    winner = 'black'
                if click_coords in white_locations:
                    selection = white_locations.index(click_coords)
                    if turn_step == 0:       turn_step = 1
                if click_coords in valid_moves and selection != 100:
                    white_locations[selection] = click_coords
                    if click_coords in black_locations:
                        black_piece = black_locations.index(click_coords)
                        captured_pieces_white.append(black_pieces[black_piece])
                        if black_pieces[black_piece] == 'king':
                            winner = 'white'
                        black_pieces.pop(black_piece)
                        black_locations.pop(black_piece)
                    black_options = check_options(black_pieces, black_locations, 'black')
                    white_options = check_options(white_pieces, white_locations, 'white')
                    turn_step = 2
                    selection = 100
                    valid_moves = []
            if turn_step > 1:
                if click_coords == (8, 8) or click_coords == (9, 8):
                    winner = 'white'
                if click_coords in black_locations:
                    selection = black_locations.index(click_coords)
                    if turn_step == 2:
                        turn_step = 3
                if click_coords in valid_moves and selection != 100:
                    black_locations[selection] = click_coords
                    if click_coords in white_locations:
                        white_piece = white_locations.index(click_coords)
                        captured_pieces_black.append(white_pieces[white_piece])
                        if white_pieces[white_piece] == 'king':
                            winner = 'black'
                        white_pieces.pop(white_piece)
                        white_locations.pop(white_piece)
                    black_options = check_options(black_pieces, black_locations, 'black')
                    white_options = check_options(white_pieces, white_locations, 'white')
                    turn_step = 0
                    selection = 100
                    valid_moves = []
        if event.type == pygame.KEYDOWN and game_over:
            if event.key == pygame.K_RETURN:
                game_over = False
                winner = ''
white_pieces = ['rook', 'knight', 'bishop', 'king', 'queen', 'bishop', 'knight', 'rook','pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn']
white_locations = [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1)]
black_pieces = ['rook', 'knight', 'bishop', 'king', 'queen', 'bishop', 'knight', 'rook', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn']
black_locations = [(0, 7), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7), (0, 6), (1, 6), (2, 6), (3, 6), (4, 6), (5, 6), (6, 6), (7, 6)]
                captured_pieces_white = []
                captured_pieces_black = []
                turn_step = 0
                selection = 100
                valid_moves = []
                black_options = check_options(black_pieces, black_locations, 'black')
                white_options = check_options(white_pieces, white_locations, 'white')
    if winner != '':
        game_over = True
        draw_game_over()
    pygame.display.flip()
pygame.quit()
```

05. Result:


White: Move piece

Error 1:

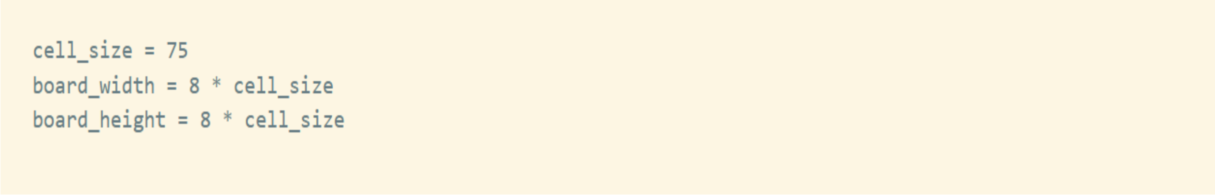


```
board_width = 8 * cell_size
board_height = 8 * cell_size
```

Cause:

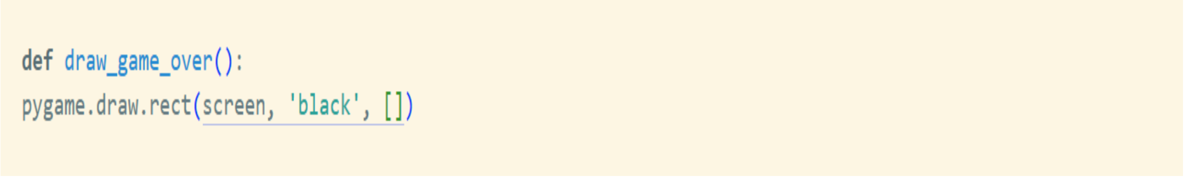
The cell size needs to be defined.

Solution:



```
cell_size = 75
board_width = 8 * cell_size
board_height = 8 * cell_size
```

Error 2:

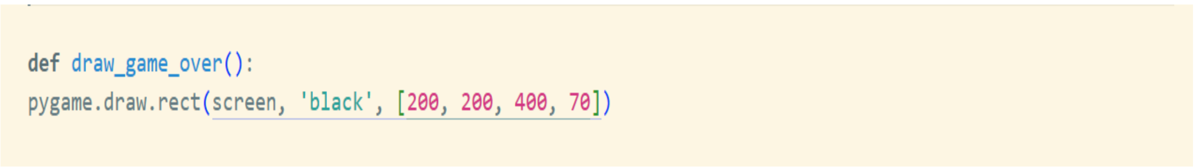


```
def draw_game_over():
pygame.draw.rect(screen, 'black', [])
```

Cause:

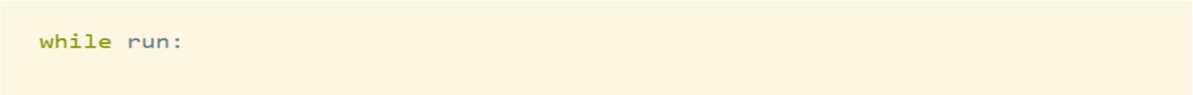
The position and dimension values are not specified.

Solution:



```
def draw_game_over():
pygame.draw.rect(screen, 'black', [200, 200, 400, 70])
```

Error 3:

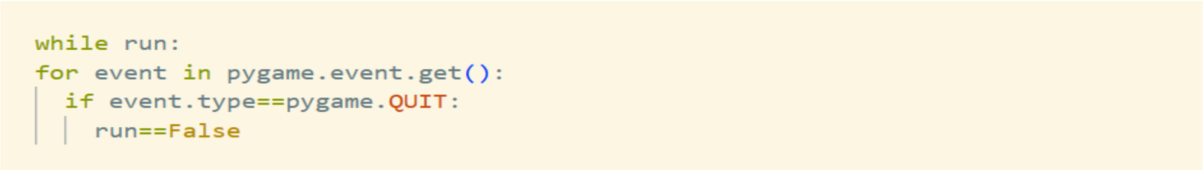


```
while run:
```

Cause:

There is no exit condition for the main loop, the game will run indefinitely.

Solution:



```
while run:
for event in pygame.event.get():
  if event.type==pygame.QUIT:
    run==False
```

06. Algorithm Used:

1. Brute Force Move Generation for Each Piece:

- The valid movement for each type of chess piece (Pawn, Rook, Knight, Bishop, Queen, King) is determined using basic brute-force techniques, which explore possible moves in all directions a piece can move.
- The Rook moves along horizontal and vertical lines, checking all squares in those directions until it encounters an obstacle (another piece).
- TheBishop moves diagonally, similarly checking for obstacles along its path.
- The Knight has an "L-shaped" movement, and this is handled by checking the set of squares that a Knight can jump to from its current position
- Pawn movement is handled by checking squares directly ahead and capturing diagonally, as well as including special moves such as en passant.
- The King moves one square in any direction and checks surrounding squares.
- This brute force approach does not involve any optimization algorithms, but ensures the correct legal moves are generated for each piece.

2. Turn-Based Control:

- The game alternates between two players (White and Black), and each player's turn consists of two phases: selecting a piece and moving it. The current turn is tracked using a variable (turn_step) to manage which player's turn it is.
- Turn management allows for the alternating sequence of moves between the two players but does not involve any complex decision-making or AI algorithms.

3. Piece Capture and Basic Game State Management:

- When aplayer moves a piece to a square occupied by an opposing piece, that opposing piece is captured and removed from the board. The board's state is updated accordingly.
- Although the game keeps track of the players' pieces and moves, there is no detailed board evaluation or AI strategy implemented in this version.

4. Visual Representation of Game State:

- The game's board and pieces are rendered visually, with each move and piece's position being updated after each player's turn. This rendering is done via a graphical user interface (GUI), ensuring that the game state is displayed clearly for the players.

5. Game Over Detection:

- Basic checks are performed to detect whether the game is over (i.e., whether one player has won). A player wins if the opponent's King is captured. However, checkmate and stalemate conditions are not evaluated in the current code.
- Theprogram also tracks whose turn it is and prompts the user accordingly.

07. Future Considerations:

- **AI Implementation:** If AI were to be implemented, algorithms such as Minimax or Alpha-Beta Pruning would be used to allow the computer to choose optimal moves based on evaluating potential future game states.
- **GameState Evaluation:** An evaluation function could be added to assess the strength of different board positions, considering factors such as material balance, piece activity, and King safety

08. Limitations:

1.No Advanced Chess Rules:

- **En sPassant:** The game lacks the implementation of special pawn capture rules like en passant, where a pawn can capture an opponent's pawn that moved two squares forward.
- **Castling:** Castling, a critical move for king safety, is not yet implemented.
- **Pawn Promotion:** While pawns can move forward, they don’t automatically promote to a queen (or other pieces) upon reaching the opposite end of the board.
- **Stalemate:** The game does not detect stalemate situations where a player cannot make a legal move, but their king is not in check.
- **Draw Conditions:** Rules like repetition (threefold repetition), insufficient material, or the 50-move rule for a draw are not implemented.

2.No Check or Checkmate Detection

- The game does not enforce check, checkmate, or king safety. Players can move their pieces in such a way that the king remains in check, which is not allowed in chess.

3.Lack of Undo/Redo Functionality

- The project does not allow players to undo or redo moves, which is useful for reviewing or correcting gameplay.

4.No AI or Single-Player Mode

- The project is limited to two-player mode. There is no AI opponent for single-player gameplay.

5.No Save/Load Feature

- Players cannot save the current game state and resume it later.

09. Conclusion:

The development of the two-player Chess game successfully combines strategic gameplay with a user-friendly digital interface, bringing the traditional board game into a virtual environment. By accurately implementing the rules of Chess, including move validations, turn-based gameplay, and checkmate detection, the project provides an engaging experience for players. This project not only demonstrates a deep understanding of programming logic, user interface design, and error handling but also serves as a platform for enhancing critical thinking and problem-solving skills. It highlights the importance of combining technical skills with creativity to replicate a real-world game in a digital format. Overall, the Chess game project fulfills its objectives of creating an interactive and enjoyable experience for two players while showcasing the potential of programming to develop classic games in modern applications.

10. Reference:

Python Documentation. "Python Programming Language Official Documentation". Available at: https://docs.python.org/3/

Pygame Documentation. "Pygame: Python Game Development".

Available at: https://www.pygame.org/docs/

Chess.com, "Chess Rules and Strategies".

Available at: https://www.chess.com/