Chess

Session 2023 – 2027

Submitted by:

Ayesha Wasim	2023-CS-66
Khadija Saeed	2023-CS-74
Ufaq Hafeez	2023-CS-75
Hira Sohail	2023-CS-76

Supervised by:

Sir Nazeef ul Haq

Course:

Data Structures and Algorithms

Department of Computer Science

University of Engineering and Technology
Lahore Pakistan

Table of Contents:

Table of Figures:

1. Introduction:

This document provides a comprehensive report on the implementation of a chess game using Python, with a focus on BFS-based movement validation. The game supports standard chess rules, including special moves such as castling, en passant, and pawn promotion. This report details the algorithms, code structure, and functionality of the game, highlighting its robust and modular design. Additionally, the report delves into the historical evolution of chess, its cultural significance, and its intersection with modern technology. The chess game project is designed to provide a digital representation of the classic board game, leveraging Python's capabilities for graphical user interfaces and logic implementation. The program integrates fundamental chess rules with advanced features like BFS-based move validation to ensure a seamless and engaging user experience.

2. Software Requirements

- Python 3.8+
- Required Libraries: Pygame, Sys

Setting up the Environment:

Install Python:

- Download and install Python 3.8 or later from the official Python website.
- During installation, make sure to check the option "Add Python to PATH" to make Python accessible from the command line.

Install Required Libraries:

- Open a terminal or command prompt and run the following command to install the pygame library: pip install pygame.
- The sys library is part of Python's standard library and is included by default, so you don't need to install it separately.

Troubleshoot Installation Issues:

- If you face issues, ensure Python is added to your system's PATH. To check, type python --version in the terminal. If it doesn't show the version, reinstall Python and check the *"Add to PATH"* option during setup.
- If pip isn't recognized, refer to the pip documentation for help.

3. System Design and Architecture:

The game is designed with a modular architecture, where different components of the chess game are organized into separate classes. The core functionality is managed by the ChessGraph class, which handles the board state, piece management, and game logic. The Player class manages player-specific information, including turn management. The Node class represents the individual squares on the chessboard, and the Stack class is used to store the history of moves, enabling undo functionality. Additionally, the ChessAI class is responsible for controlling the AI player. These components interact seamlessly to ensure smooth gameplay.

3.1 Instructions:

- Play Human vs Human or Human vs AI.
- In every mode player 1will get the first turn.
- Objective: Checkmate the opponent's King.
- Click a piece to select, then click a valid square to move.
- Press "U" or click Undo to revert the last move.
- In Rapid Mode, manage your time or lose when it runs out.
- Captured pieces are shown on the sides.
- Undo a Move: Press the **U** key to undo the last move. In AI mode, pressing U will undo both the AI's and your last move.
- The game ends if one player's king is captured or if the game is manually exited.

4. Game Features and Functionality:

4.1 Chess Board Initialization

The chessboard is initialized with the correct arrangement of pieces, and the game ensures that players take turns to move their pieces. Each piece has its own set of valid moves, which are calculated dynamically based on the piece type and current position.

4.2 Gameplay Logic

Special rules, such as castling and pawn promotion, are incorporated into the game logic to ensure the correct handling of these scenarios. The game also checks for conditions like check and checkmate to determine the outcome of the game.

4.3 Special Rules

- Handling castling (kingside and queenside).
- Checking for check and checkmate conditions.
- Detecting square attacks by opponent pieces.

4.4 Move Validation

Check if the Move is Valid:
Every time you make a move, the game checks if it follows the rules:

- Is it a legal move for the selected piece?
- Does the move put your own king in check? If yes, it's not allowed.

Reverting Invalid Moves:

- If a move is not valid, the game will notify you.
- The board will stay as it was before the invalid move, so you can try again.

5. Data Structures:

5.1 Nodes

Purpose: The nodes data structure represents the board's layout and stores information about each square.

Type: Self.nodes is a **dictionary** where keys are positions ((row, col) tuples), and values are objects representing each square on the chessboard.

Details of Each Node

Each node contains:

Piece*:* The piece present on the square. For example white rook, black pawn or none if the square is empty.

Color: The color of the piece occupying the square white or black.

Has_moved: This is a bool variable that tracks whether the piece on the square has moved during the game. It is useful for rules like castling or pawn movement.

Role in BFS Move Calculations

Nodes are essential for validating moves and generating legal ones. It determine if a square is empty or occupied. It distinguish between enemy and ally pieces based on color. It also support algorithms like BFS or DFS to calculate valid moves for pieces like Queens, Bishops, or Rooks.

5.2 King Positions

Purpose: Tracks the positions of both kings to verify game-ending conditions.

Type:

Dictionary: self.king_positions stores the current positions of the white and black kings.

Keys: white and black.

Values: Position tuples like (row, col).

Usage: Used in the is_game_over method to check if either king is missing from the board. It also helps identify game win conditions.

5.3 Move History:

Purpose: Tracks the history of moves for advanced mechanics like en passant.

Type

Stack: Self.move_history is implemented as a stack to store recent moves in a Last In, First Out (LIFO) manner.

Details of Move Data

Each move entry contains:

Source Position: Where the move originated.

Destination Position: Where the move ended.

Piece: The piece involved in the move.

Additional Metadata: Can include information like the captured piece.

Role in En Passant

During pawn moves, move_history is checked to validate whether the last move qualifies for en passant. En passant checks the source and destination positions of the last pawn move.

5.4 Directions (for BFS)

Purpose: Defines movement directions for pieces to traverse the board during BFS calculations.

Type

List of Tuples: Used to represent offsets in rows and columns for different pieces.

Examples by Piece

Rook: [(-1, 0), (1, 0), (0, -1), (0, 1)] (up, down, left, right).

Bishop: [(-1, -1), (-1, 1), (1, -1), (1, 1)] (diagonals).

Queen: Combines Rook and Bishop directions.

King: Similar to Queen but limited to one step.

Pawn: Forward-only movement, with diagonal captures.

Usage: It enables dynamic traversal of the board. It is also used to calculate moves until a boundary or obstacle is hit.

5.5 Rows and Columns

Purpose: Defines board boundaries.

Type:

Constants: ROWS and COLS represent the dimensions of the chessboard (8x8).

Usage: Validates moves to ensure they remain within the board's boundaries. Helps in BFS traversal logic by restricting moves outside [0, ROWS-1] and [0, COLS-1].

5.6. En Passant Logic

Purpose: Special pawn capture rule requiring validation of the last move.

Type: It is derived from Move History. En passant uses the move_history stack to check the most recent move.

Usage: Validates if the previous move was a pawn moving two squares forward. It checks if the target position aligns with en passant conditions.

5.7. Potential Moves (for King and Pawns)

Purpose: Defines pre-calculated possible moves for pieces with limited mobility like King and Pawn.

Type: List of Tuples: Represents all possible moves relative to the current position.

Usage:

King: [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)] for one-step moves.

Straight Movement: One or two steps forward based on the pawn's position.

Diagonal Movement: For captures.

5.8. Combining Moves for Complex Pieces

Purpose: Implements advanced move generation for composite pieces (Queen).

Type: Combination of Lists: Queen's moves combine Rook and Bishop move lists.

Usage: Leverages helper functions like get_valid_rook_moves_bfs and get_valid_bishop_moves_bfs.

5.9. Piece Metadata

Purpose: Represents individual chess pieces.

Type: Stored in self.nodes for each square.

Details:

piece: Name of the piece (e.g., "Pawn_white").

color: Color of the piece ("white" or "black").

Usage: Validates moves by differentiating between ally and enemy pieces. It helps detect captures, blocks, and empty squares.

5.10. Game State Validation

Purpose: Checks endgame conditions.

Type: Based on king positions and nodes.

Usage: Validates if a king is missing to determine the game's outcome. It can be extended to include stalemate, checkmate, or insufficient material checks.

6. Flow Chart:

```
Start
 |
 Is Source Square Valid? (Contains a piece?)
 |
 |--No--> Invalid Move (End)
 |
 |--Yes--> Is Destination Valid?
            |
            |--No--> Invalid Move (End)
            |
            |--Yes--> Does the Move Put King in Check?
                          |
                          |--Yes--> Invalid Move (End)
                          |
                          |--No--> Update Board and Toggle Turn
```

7. Table: Piece Movement Capabilities

Piece	Movement Type	Range of Movement
Pawn	Forward + Capture	1 step or 2 steps
Rook	Straight	Unlimited
Knight	L-Shape	Fixed (2+1)
Bishop	Diagonal	Unlimited
Queen	Straight/Diagonal	Unlimited
King	All Directions	1 step

Table:1

8. Class and Methods:

1. ChessGraph Class

Responsibilities:

The ChessGraph class is the core of the chess game, managing the chessboard, game logic, and interactions between players and pieces.

Attributes:

nodes: A dictionary where keys are positions ((row, col)) and values are Node objects representing each square on the chessboard.

king_positions: A dictionary storing the current positions of both kings ('white' and 'black').

players: A dictionary of Player objects for managing player-specific information.

move_history: A stack to keep track of all moves for undo functionality.

ai: An instance of ChessAI for handling AI-based moves.

captured_white_pieces: A list of white pieces captured during the game.

captured_black_pieces: A list of black pieces captured during the game.

Key Methods:

➢ **__init__()**

- **Purpose:** Initializes the chessboard, pieces, and game state.
- **Parameters:** None.
- **Returns:** None.
- **Logic:** Calls helper methods to set up the chessboard and pieces.

➢ **create_chessboard()**

- **Purpose:** Sets up an 8x8 chessboard by populating nodes with Node objects.
- **Parameters:** None.
- **Returns:** None.
- **Logic:** Uses nested loops to create rectangles representing squares on the board.

➢ setup_pieces()

- **Purpose:** Initializes the pieces in their starting positions.
- **Parameters:** None.
- **Returns:** None.
- **Logic:** Assigns images and positions to pawns, rooks, knights, bishops, queens, and kings.

➢ can_castle(player_color, side)

- **Purpose:** Checks whether castling is allowed for a given player and side (kingside or queenside).
- **Parameters:** player_color (str): The player's color ('white' or 'black'), side (str): Either kingside or queenside.
- Returns: True if castling is valid, False otherwise.

➢ move_piece(source, destination)

- **Purpose:** Handles piece movement and updates the game state.
- **Parameters:** source (tuple): Starting position of the piece. destination (tuple): Ending position of the piece.
- Returns: None.

2. Node Class

Responsibilities:

Represents a single square on the chessboard, managing its state and the piece it holds.

Attributes:

position: A tuple representing the position (row, col) of the square.

rect: A Pygame rectangle object for rendering.

piece: The piece on the square (e.g., 'Pawn_white').

image: The image of the piece for rendering.

color: The color of the piece ('white' or 'black').

has_moved: A boolean indicating if the piece has moved.

Key Methods:

set_piece(piece, image, color)

- **Purpose:** Assigns a piece to the node.
- **Parameters:**

piece (str): The piece name.

image (pygame.Surface): The image of the piece.

color (str): The piece color.

Returns: None.

remove_piece()

- **Purpose:** Clears the square by removing the piece.

- **Parameters:** None.
- **Returns:** None.

3. Player Class

Responsibilities:

Manages player-specific information, such as color and turn.

Attributes:

color: The player's color ('white' or 'black').

turn: A boolean indicating whether it is the player's turn.

Key Methods:

toggle_turn()

- **Purpose:** Switches the player's turn.
- **Parameters:** None.
- **Returns:** None.

set_turn(turn)

- **Purpose***:* Explicitly sets the player's turn state.
- **Parameters:**
- **turn (bool):** The desired turn state.
- **Returns:** None.

9. UML Diagram:

The diagram represents class relationships in a chess game.

Classes interact through composition and aggregation relationships, showcasing their modular design.

```
+----------------+          +----------------+       +----------------+       +----------------+
|  ChessGraph    |           |     Node       |        |    Player      |        |   ChessAI      |
+----------------+           +----------------+       +----------------+       +----------------+
|    - nodes     |<>------->| - position      |        | - color        |        | - ai_color     |
| - players      |           | - piece        |        | - turn         |        | - difficulty   |
| - king_pos     |           | - color        |       +----------------+       +----------------+
| - moveHistory |            +---------------+        | + toggle_turn  |        | + get_best_move|
+----------------+          | + set_piece    |        | + set_turn     |        |                |
| + create_board |          |+ removePiece |       +----------------+       +--------------------+
| + move_piece  |
| + can_castle    |
+------------------+
```

10. AI Functionality:

In the chess game, the AI plays the role of a computer opponent that analyzes the game state and makes decisions based on the Minimax algorithm with Alpha-Beta pruning. The goal of the AI is to make the best move by evaluating possible outcomes using these techniques.

10.1 Introduction

The ChessAI class is responsible for handling the AI logic in the chess game. It uses a depth-first search algorithm, namely **Minimax**, to evaluate the game state and decide on the best move. The AI analyzes all possible moves for a given depth and chooses the move that maximizes its chances of winning (or minimizes the opponent's chances, depending on the turn).

10.2 Purpose of the AI

The purpose of the AI is to make strategic decisions in the game based on a set of moves, ensuring that it plays optimally or as intelligently as the set depth allows. The AI logic considers various pieces' movements, evaluates the board, and predicts outcomes based on the current game state.

11. Components of AI:

11.1 Constructor

- The **__init__** method initializes the AI with:
- ***Color:*** The color of the AI (either 'white' or 'black'). This determines which player's moves the AI will make.
- ***Depth:*** The depth of the search in the Minimax algorithm. A higher depth allows for a more thorough evaluation but increases computation time.

11.2Best Move Decision (get_best_move)

- The **get_best_move** function evaluates all possible moves using the Minimax algorithm. It iterates through every possible move for the AI's color and applies the Minimax evaluation to determine the best possible move.
- For each move, it calculates its score by calling the minimax method.
- It selects the move with the best score based on the AI's color (maximizing for white and minimizing for black).

12. Move Generation (get_all_possible_moves):

The **get_all_possible_moves** function generates all possible valid moves for the given color. It looks at all the nodes (squares) on the chessboard, checks the piece on each square, and determines the valid moves for that piece using predefined movement functions.

- The chess graph (chess_graph.nodes) holds information about the state of each square on the board.
- For each piece type like Pawn, Knight, Rook, Bishop, Queen, King, it calls the respective function to get valid moves.

13. How Move Generation Works:

Pawn: Moves are generated using Breadth-First Search (BFS) to account for both single square and double square forward movement, as well as capturing moves.

Knight: The Knight moves in an 'L' shape, and the valid moves are identified using BFS as well.

Rook, Bishop, Queen, King: For each of these pieces, valid moves are determined based on their movement rules (straight lines for Rook, diagonals for Bishop, and a combination of both for Queen). BFS is used to get the valid moves within the constraints of the chessboard.

14. Minimax Algorithm with Alpha-Beta Pruning:

14.1 Minimax Algorithm

The minimax function is a recursive algorithm that explores all possible future moves in the game up to a specified depth. It evaluates each move by assuming both players play optimally, with the AI trying to maximize its score and the opponent trying to minimize it.

- ➢ Maximizing Player: The AI tries to maximize the evaluation score (for white).
- ➢ Minimizing Player: The opponent (black) tries to minimize the evaluation score.

14.2 Alpha-Beta Pruning

Alpha-Beta pruning is an optimization technique used in Minimax to avoid unnecessary evaluations of moves. The algorithm keeps track of two values:

- **Alpha**: The best value that the maximizing player can guarantee so far.
- **Beta**: The best value that the minimizing player can guarantee so far.

If at any point, the value of a node becomes worse than an already evaluated move, further exploration of that node is stopped (pruned). Pruning occurs if the score is worse than alpha (for minimizing player) or beta (for maximizing player), then the current move is discarded.

14.3 Evaluation of Board (evaluate_board)

The evaluate_board function assigns a numerical value to the board state. This is based on the pieces remaining on the board:

- **Positive value** for white pieces (maximizing player).
- **Negative value** for black pieces (minimizing player).
- The value of each piece is assigned according to standard chess piece values (Pawn = 1, Knight/Bishop = 3, Rook = 5, Queen = 9, King = 0).

14.4 Simulation of Move

Before evaluating the potential outcomes, the function **simulates** a move to see its effect on the board. This simulation ensures that the AI is considering the real impact of each move.

- **Move is simulated** using move_piece with a simulated=True flag to avoid actually changing the game state.
- **Undoing the move**: After evaluating the move, the state is restored using the undo_simulated_move method to backtrack and evaluate other possible moves.

15. Table: Move Types for Each Piece

This table describes the types of moves available to each piece and how the AI generates them in the get_all_possible_moves method using the BFS (Breadth-First Search) approach.

Piece	Movement Description	Movement Function
Pawn	Moves forward one square, two squares on first move, captures diagonally.	get_valid_pawn_moves_bfs
Knight	Moves in an "L" shape (two squares in one direction and one square perpendicular).	get_valid_knight_moves_bfs
Rook	Moves any number of squares horizontally or vertically.	get_valid_rook_moves_bfs
Bishop	Moves any number of squares diagonally.	get_valid_bishop_moves_bfs
Queen	Combines the movement of both rook and bishop (any number of squares in any direction).	get_valid_queen_moves_bfs
King	Moves one square in any direction.	get_valid_king_moves_bfs

Table:2

16. Chess Game User Interface (UI):

Introduction

16.1 Overview

This section describes the UI part of the chess game, implemented using the Pygame library. The code manages the graphical display of the chessboard, piece movements, capturing of pieces, and interaction between players in different game modes.

16.2 Purpose

The primary goal of this code is to create a user interface for the chess game that supports:

- Player interaction via mouse clicks for piece movement.
- Real-time updates of the chessboard after each move.
- Displaying captured pieces.
- Handling both "Human vs Human" and "Human vs AI" modes.

Key Libraries and Dependencies

16.3 Pygame

Pygame is used to create the game window and handle graphical operations, such as drawing the chessboard, moving pieces, and handling user input.

16.4 Custom Modules

- **ChessAI**: Manages the AI's move logic and decision-making.
- **ChessGraph**: Represents the chessboard structure, keeps track of pieces' positions, and handles move generation and validation.
- **Constants**: Contains constants such as piece images, square size, board size, and colors.

Code Structure

16.5 Initial Setup

- **pygame**: Main library for rendering the UI.
- **argparse**: Used for parsing command-line arguments for player names and game mode.
- **ChessAI**: Importing the AI module for handling AI moves.
- **ChessGraph**: Importing the chessboard logic that manages the state and behavior of the game.
- **constants**: Contains pre-defined constants used for board layout, colors, and piece images.

16.6 draw_board Function

Purpose: Draws the chessboard and all pieces on the screen, highlighting valid moves if provided.

Parameters:

- screen: The game window where everything is drawn.
- chess_graph: An object that holds the state of the chessboard and pieces.
- valid_moves: A list of valid moves to be highlighted on the board.

Steps:

1. **Drawing the Border**: The border is drawn using a rectangle around the chessboard area.
2. **Drawing the Chessboard**:
    1. Iterates through the rows and columns of the chessboard.
    2. Draws squares using alternating colors (White and Brown).
    3. Places the piece images in their respective squares.
    4. Checks if a piece is in check and highlights the king in red if so.
3. **Highlighting Valid Moves**: If valid moves are provided, they are highlighted in green.
4. **Displaying Coordinates**: Ranks (1-8) and files (a-h) are drawn on the sides of the board to help players identify squares.

16.7 draw_captured_pieces Function

- **Purpose**: Displays the captured pieces of both players at the edges of the screen.
- **Parameters**:
    - screen: The game window.
    - captured_white_pieces: List of pieces captured by the white player.
    - captured_black_pieces: List of pieces captured by the black player.

Steps:

1. **Displaying White Captured Pieces**: Captured white pieces are shown at the bottom-right of the screen.
2. **Displaying Black Captured Pieces**: Captured black pieces are shown at the top-right.
3. **Scaling Images**: Captured pieces are scaled to fit within the available space.

16.8 Game Modes:

- **Human vs Human (mode = 0)**: Both players take turns making moves manually.
- **Human vs AI (mode = 1)**: The human player plays against the AI.

Game Flow:

1. **Initialization**: Sets up the game screen, chess graph, and AI (if playing against the AI).
2. **Event Handling**: The game listens for events like mouse clicks, key presses (for undo), and quitting the game.
3. **Player Movement**:

  - In Human vs Human mode, players select pieces and move them by clicking on squares.
  - In Human vs AI mode, after the human player makes a move, the AI makes its move after a short delay.

4. **Undo Move**: If the 'u' key is pressed, the last move is undone for both players (AI first, then human).
5. **Drawing the UI**: After each event, the screen is updated by drawing the chessboard and captured pieces.

16.9 Handling Player and AI Turns

- **Player's Turn**:
  - Players click on the piece they want to move, then on the destination square.
  - The selected piece's valid moves are highlighted.
  - If the move is valid, the piece is moved and the turn passes to the opponent.
- **AI's Turn**:
  - After the human player's move, the AI calculates the best move using the ChessAI class and makes its move.

17. User Interaction and Gameplay:

17.1 Player Movement

- Players select a piece by clicking on it, and then click on a valid square to move it. Valid moves are highlighted as green squares.

17.2 AI Movement

- The AI uses the ChessAI class to calculate the best move after the human player's turn. It does this by evaluating the board using the Minimax algorithm with Alpha-Beta pruning.

17.3 Undo Functionality

- Players can press the 'u' key to undo the last move. The AI's move is undone first, followed by the human player's move.

18. Visual Components:

18.1 Chessboard

- The chessboard consists of 8x8 squares, alternating in color between white and brown. Each square is 64x64 pixels.

`18.2 Pieces

- Each chess piece is represented by an image, which is drawn on the corresponding square. The images are loaded from the PIECE_IMAGES dictionary.

18.3 Captured Pieces

- Captured pieces are displayed at the edges of the board: white pieces at the bottom-right and black pieces at the top-right. The pieces are scaled to fit in a small area.

19. Wireframes:

The following wireframes focus on various core aspects of the chess game, including the main menu, gameplay interface, settings, move history, and more. Each wireframe is designed with a user-centric approach, ensuring simplicity, accessibility, and intuitive navigation for players of all levels.

❖ Main menu

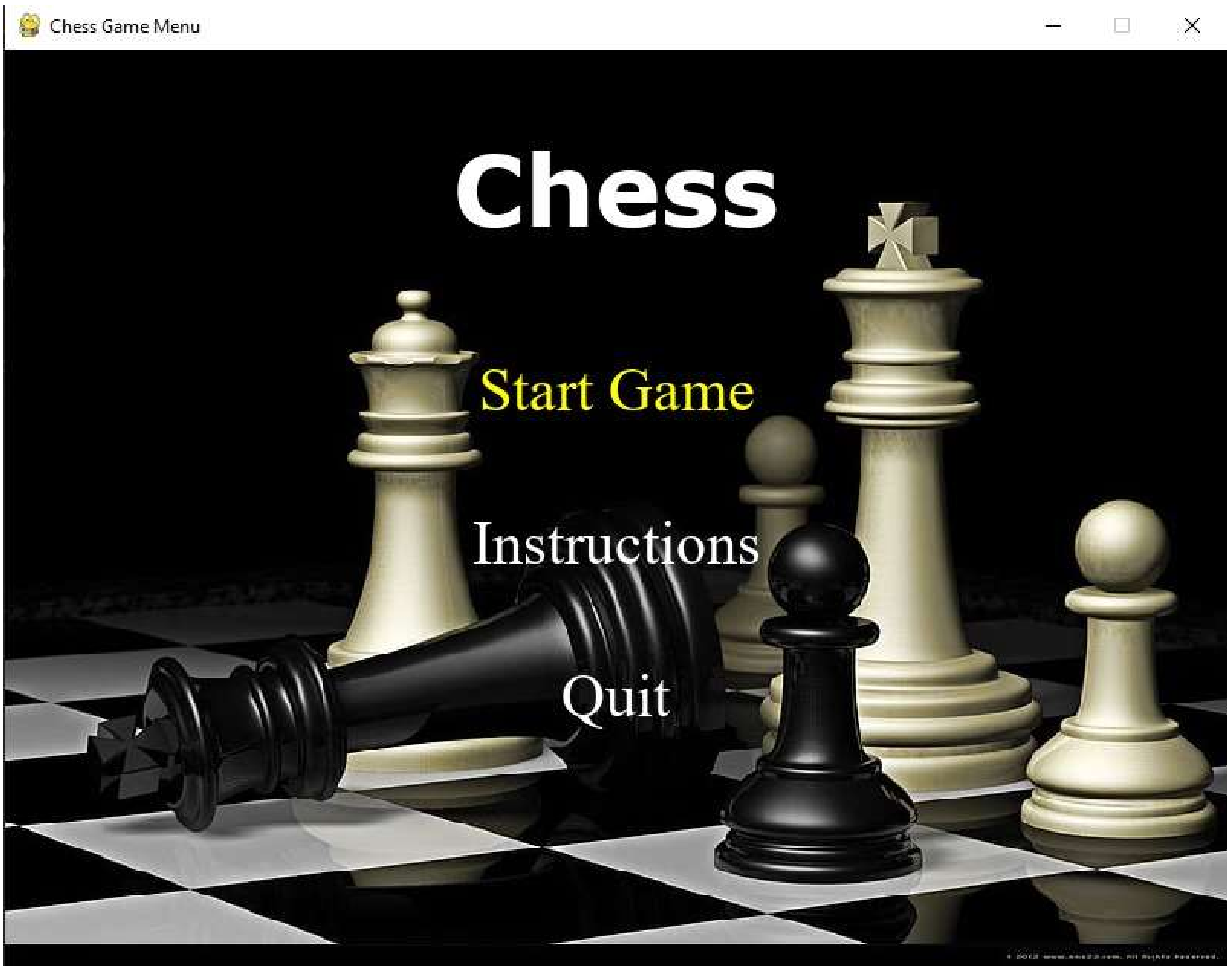
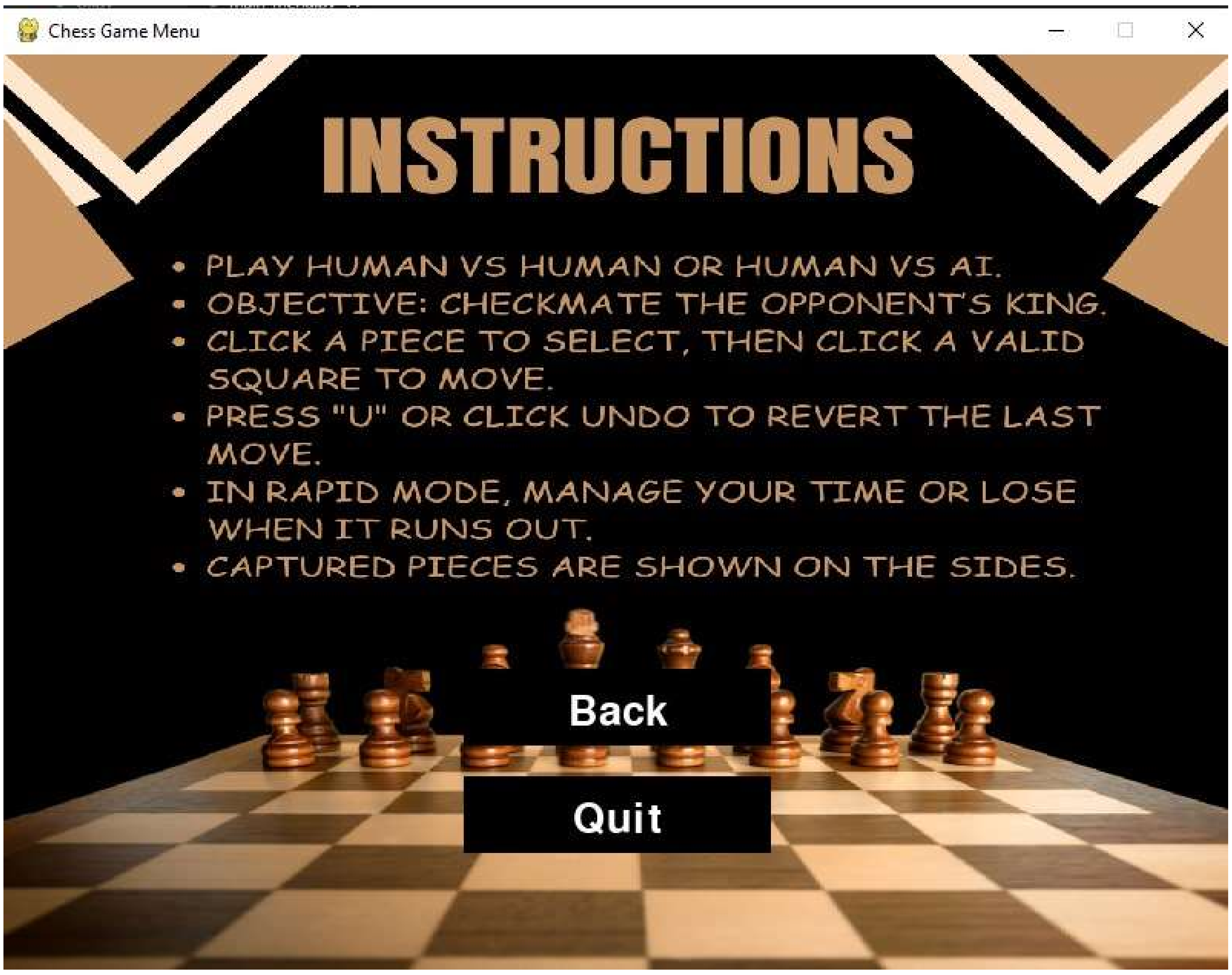


Figure:1

❖ Instructions:



Chess Game Menu

INSTRUCTIONS

- PLAY HUMAN VS HUMAN OR HUMAN VS AI.
- OBJECTIVE: CHECKMATE THE OPPONENT'S KING.
- CLICK A PIECE TO SELECT, THEN CLICK A VALID SQUARE TO MOVE.
- PRESS "U" OR CLICK UNDO TO REVERT THE LAST MOVE.
- IN RAPID MODE, MANAGE YOUR TIME OR LOSE WHEN IT RUNS OUT.
- CAPTURED PIECES ARE SHOWN ON THE SIDES.

Figure:2

- **Mode**

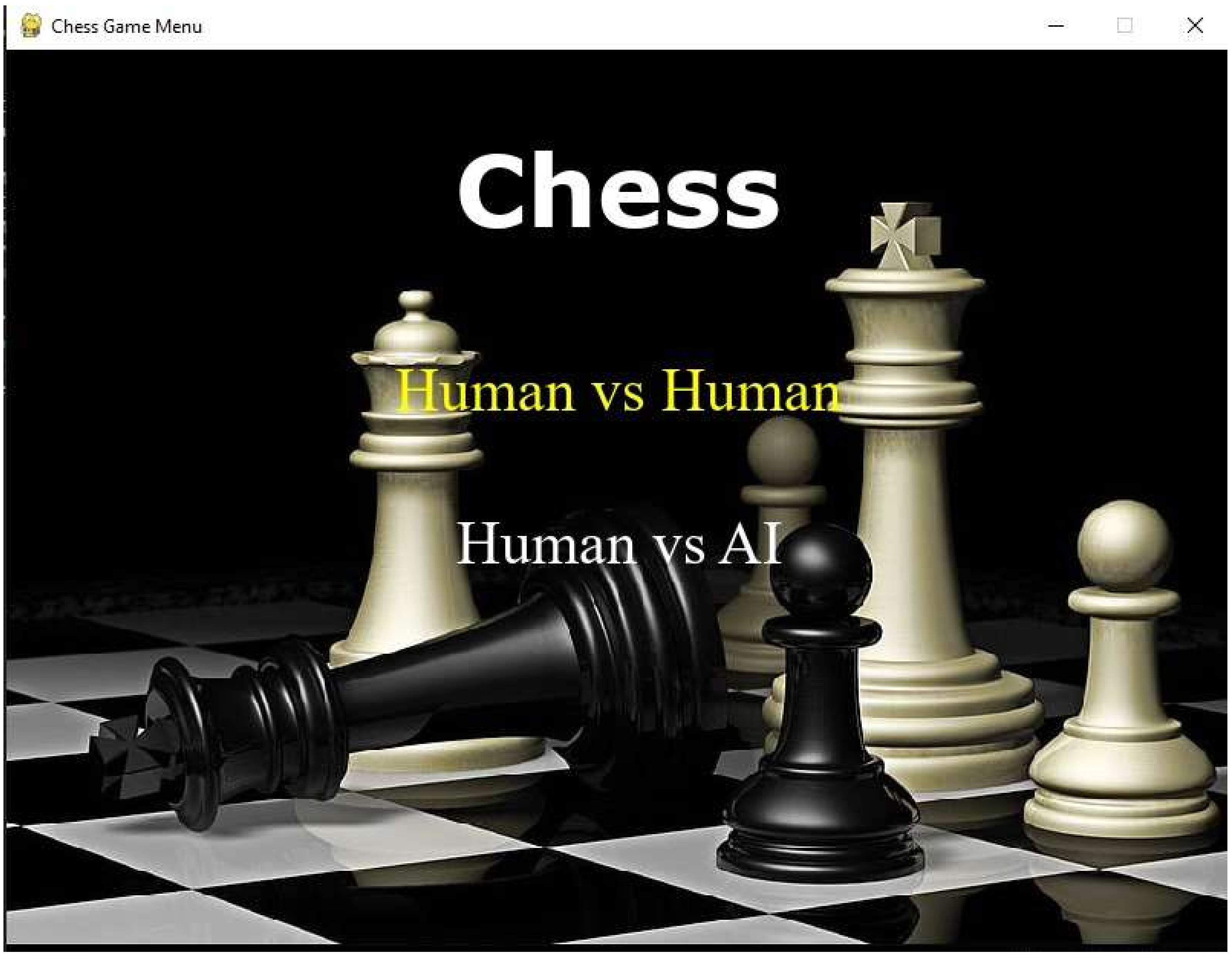
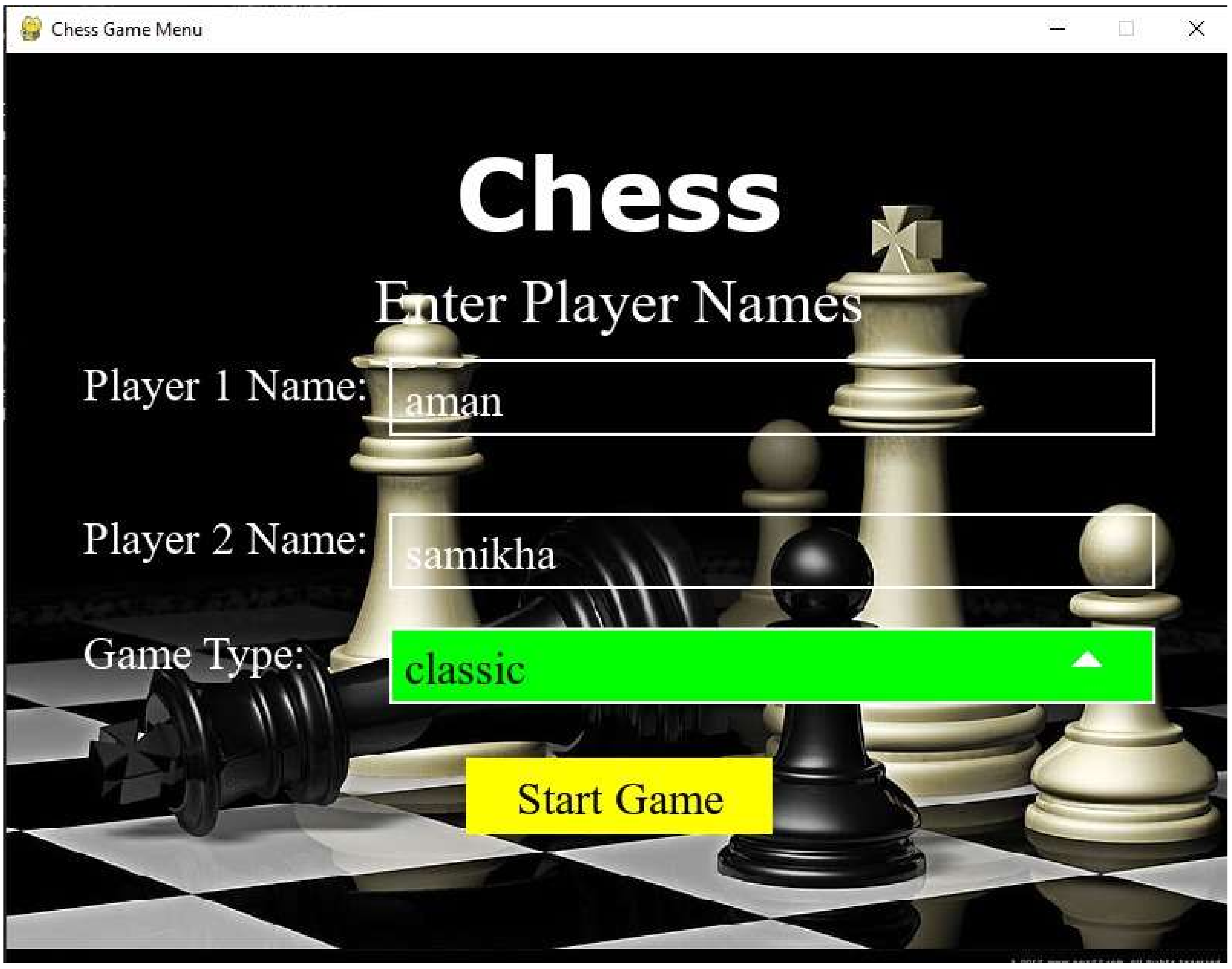


Figure:3

- **Set Player Names**



Player 1 Name: aman

Player 2 Name: samikha

Game Type: classic

Figure:4

❖ **Human VS Human**

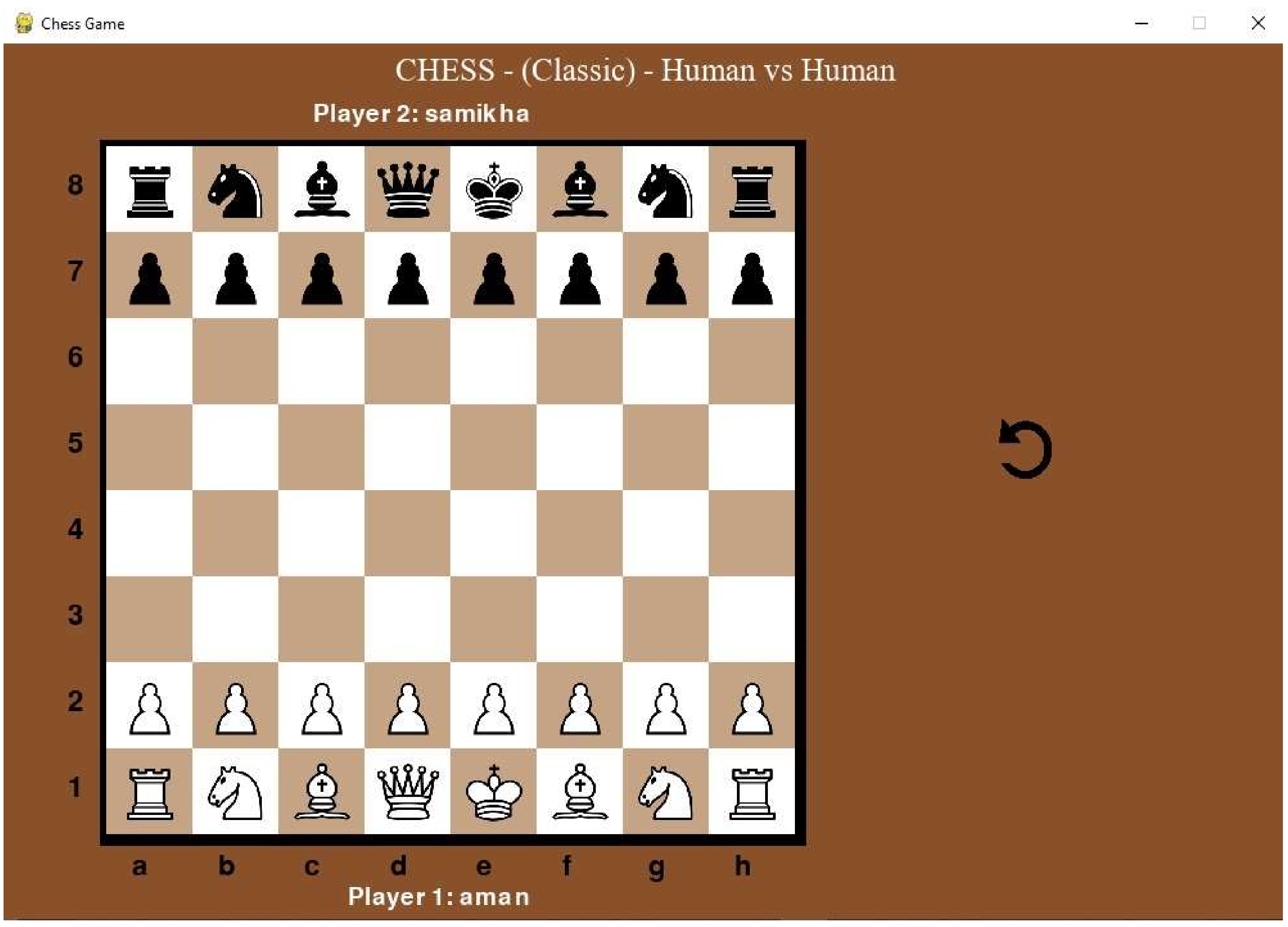


Figure:5

❖ Human VS AI

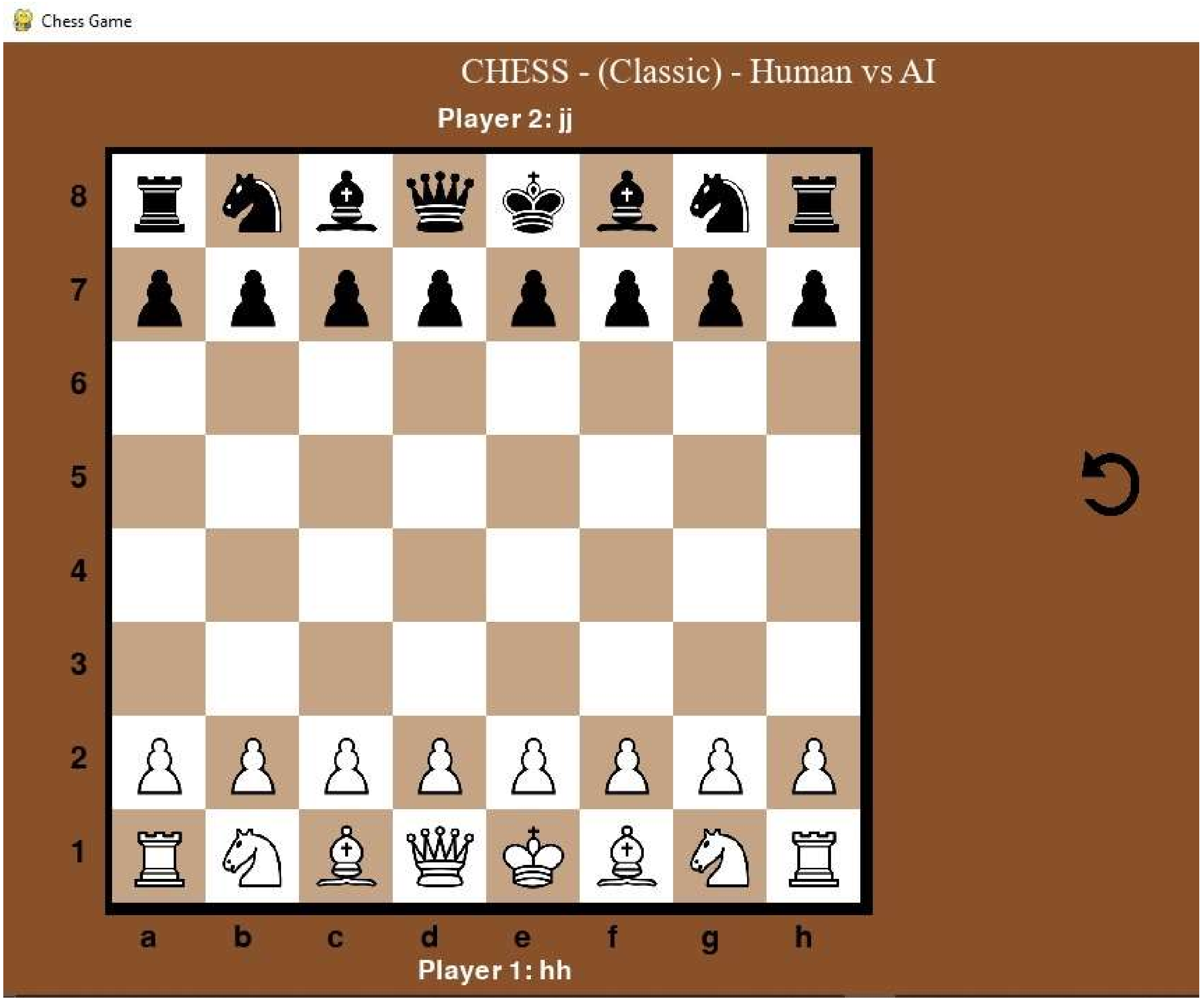
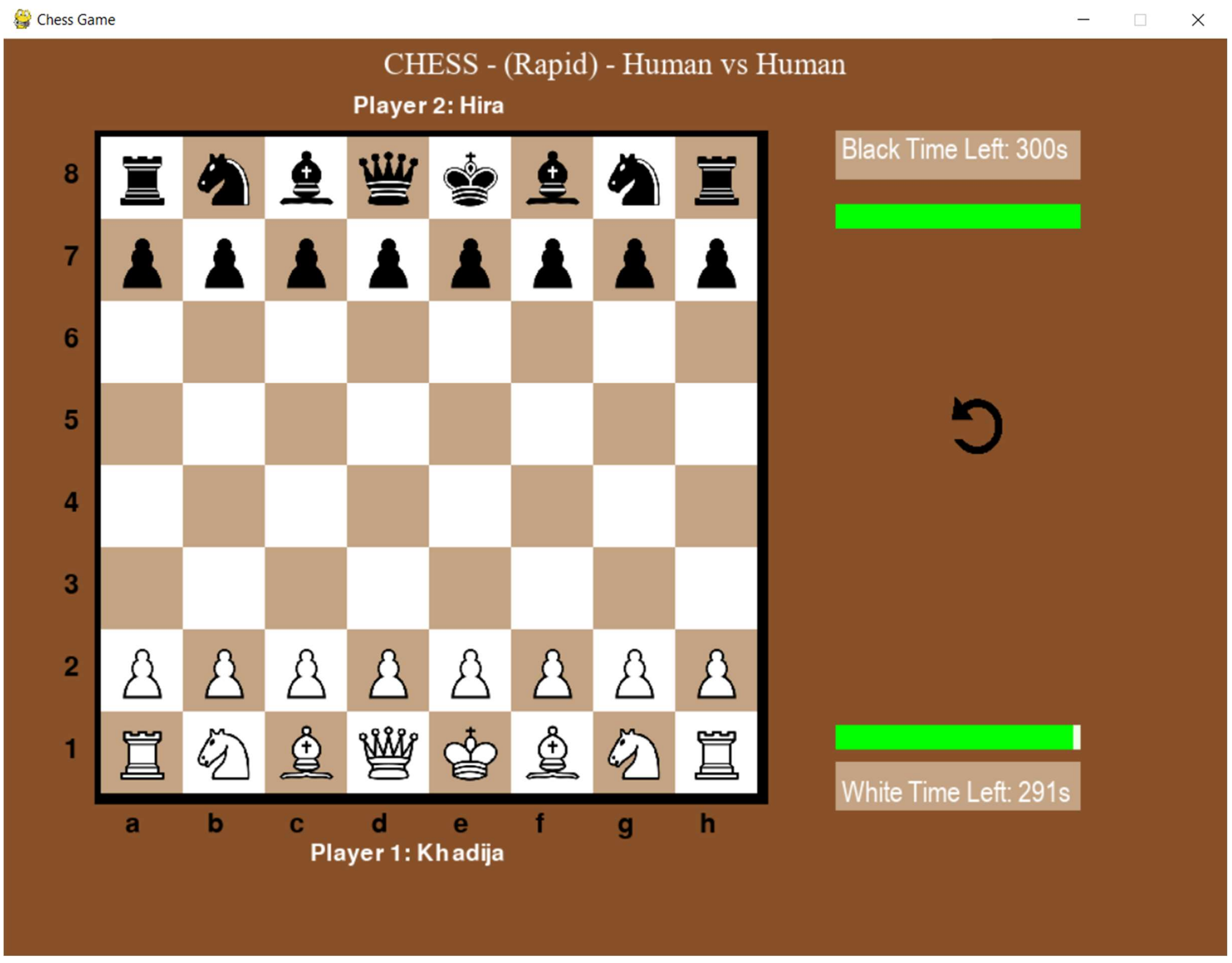

Chess Game
CHESS - (Classic) - Human vs AI
Player 2: jj
8
7
6
5
4
3
2
1
a
b
c
d
e
f
g
h
Player 1: hh

Figure:6

❖ Rapid Mode


Chess Game
CHESS - (Rapid) - Human vs Human
Player 2: Hira
Black Time Left: 300s
White Time Left: 291s
Player 1: Khadija
8
7
6
5
4
3
2
1
a
b
c
d
e
f
g
h