Chess Game for Two Players using Python

Manas Pandya
Basics of Computing - IITM Zanzibar

February 2024

Introduction

This report outlines the development process of a two-player chess game designed and implemented in Python using the Pygame library. The game features basic chess functionalities, allowing two players to compete on a single PC with visual and interactive elements to facilitate the gameplay.

1. Game Logic and Core Functionalities

Our chess game is designed with a focus on enabling two players to engage in a classic game of chess on a computer. The game is developed using Python and Pygame, leveraging the latter's capabilities for creating an interactive graphical user interface. Here, we outline the key components and functionalities implemented in our game, as highlighted by the core functions within our codebase.

1.1. Game Setup and Mechanics

Our chess game combines **Game Initialization**, **Board Visualization**, and **Piece Movement** into a seamless experience:

- **Game Initialization**: The process begins with setting up the Pygame window and loading chess piece images. The `initialize_board()` function arranges pieces on an 8x8 grid, using uppercase for white and lowercase for black pieces, ensuring an authentic setup.
- **Board and Pieces Visualization**:
  - `draw_board()`: Renders the chessboard, clearly marking the squares.
  - `draw_pieces()`: Places each chess piece image according to the game state, with special highlighting for the king when in check.
- **Piece Movement Mechanics**:
  - The `get_legal_moves()` function identifies valid moves by:
    * Evaluating the piece's type and board position.
    * Navigating obstacles and potential captures.
  - In case of a check:
    * Movements are restricted to only those that can resolve the check, safeguarding the king's position.

This streamlined approach ensures that players are engaged from the outset, offering a visually appealing board and interactive pieces, alongside movement mechanics that adhere to traditional chess rules, enhancing strategic gameplay depth.

1.2. Game State Management

Efficient game state management is essential for a dynamic and responsive chess game. Our implementation focuses on accurately tracking and updating the game state based on player actions and game rules. Here's a brief overview:

- **Board and Piece States:** The core of game state management lies in maintaining the current positions of all pieces on the board. This is achieved through a two-dimensional array representing the chessboard, where each element corresponds to a square and contains information about the piece occupying it, if any.

- **Player Turns:** The game alternates between players' turns, allowing each player to make one move per turn. This is managed by toggling the `current_player` variable between 'white' and 'black', ensuring fair and orderly gameplay.
- **Move Validation and Execution:** Upon a player's action (selecting and moving a piece), the game validates the move using the `get_legal_moves()` function to ensure it complies with chess rules and the piece's movement capabilities. Valid moves are then executed, updating the board state accordingly.
- **Check and Checkmate Handling:** The game constantly checks for check and checkmate conditions to adjust the game state as needed. If a king is in check, the game restricts the player's moves to those that resolve the check. Checkmate or stalemate conditions trigger the end of the game, with appropriate messages displayed to the players.
- **Timer Integration:** Timers for each player add an additional layer to game state management. The timers decrement in real-time, and if a player's timer expires, the game state updates to end the game, declaring the other player as the winner due to time out.

This approach to game state management ensures a smooth and rule-consistent chess experience, providing players with a clear understanding of the game's progress, turn order, and the outcomes of their actions.

1.3. Check and Checkmate Detection Mechanisms

The detection of check and checkmate situations is fundamental to replicating the strategic depth of chess. Our implementation focuses on accurately determining these states to enforce the rules of chess and provide a realistic game experience.

- **Check Detection:** The game continuously monitors the safety of the kings. The function `is_in_check(king_pos, board_state, opponent)` plays a crucial role in this process. It scans the board to identify if the current player's king is under threat from any of the opponent's pieces. This involves calculating potential moves for all opponent pieces and checking if any could capture the king in their next move. If the king's position is found in the list of legal moves for any opposing piece, the game state is updated to reflect that the king is in check.
- **Legal Move Filtering:** When a piece is selected, the `get_legal_moves()` function generates all possible moves based on the piece's type and position. However, not all these moves may be viable if the king is in check. The `filter_legal_moves_for_check()` function then filters out any moves that would leave the king in check or expose him to check. This ensures that the player must address the check by either moving the king to safety, capturing the threatening piece, or blocking the threat with another piece.
- **Checkmate Detection:** The `is_checkmate()` function is invoked to determine if the player has any legal moves left that would remove the check on their king. This function simulates each possible move from the current board state to see if the check can be resolved. If no legal moves can alleviate the check, the function concludes that the player is in checkmate, ending the game. This rigorous approach ensures that all possibilities are explored before declaring a checkmate, aligning with the game's strategic nature.
- **Game Flow Management:** The detection of check and checkmate significantly influences the game flow. Upon detecting a check, the game alerts the player, requiring immediate action to prevent checkmate. If a checkmate is detected, the game concludes, and the victory is awarded to the player delivering the checkmate. These mechanisms are integrated into the main game loop, ensuring that the game accurately follows chess rules and responds to the players' actions in real time.

Implementing these features required a deep understanding of chess rules and careful programming to accurately evaluate the complex scenarios that arise during play. The check and checkmate detection algorithms are central to the game's logic, ensuring that players engage in a challenging and rule-compliant match.

1.4. Timer Settings Implementation

A crucial component added to enhance the gameplay and introduce a sense of urgency was the implementation of countdown timers for each player. This feature ensures that each player has a fixed amount of time to make their moves throughout the game, thus simulating a more competitive environment similar to official chess tournaments.

The timer functionality was realized through the following key steps:

- **Initialization:** At the start of the game, each player is allocated a predefined amount of time. This was implemented using a Python dictionary, `player_time`, with keys for 'white' and 'black' players, each set to 120,000 milliseconds (equivalent to 2 minutes).
- **Countdown Mechanism:** The game loop continuously monitors and updates the remaining time for the current player. Utilizing Pygame's clock functionality, we calculate the elapsed time (`dt`) between each frame and decrement the current player's timer accordingly.
- **Display Updates:** The updated timer values for both players are formatted into minutes and seconds using the `format_time()` function and displayed in the game window's caption. This provides real-time feedback to the players about their remaining time.
- **Timeout Handling:** If a player's timer reaches zero, the game immediately ends, and the opponent is declared the winner due to timeout. This condition is checked in each iteration of the game loop, ensuring prompt game termination upon timeout.

This timer system not only adds a strategic layer to the game, requiring players to manage their time efficiently but also prevents excessively long play sessions, making each game more engaging and dynamic.

2. Limitations of the Work

Our chess game has been a rewarding project, yet it embodies several limitations that provide opportunities for future enhancements:

1. **Nonrestrictive Play:** The game is designed for local play on a single PC, which limits the potential for broader accessibility and competitive play. Extending functionality to support online multiplayer modes through direct connections or matchmaking servers could vastly increase its appeal and usability.
2. **Artificial Intelligence (AI) Opposition:** Currently, our game does not support playing against an AI, limiting users to human opponents only. Introducing AI opponents with varying levels of difficulty would not only cater to solo players but also serve as a tool for improving one's chess skills.
3. **Save/Load Game Feature:** The absence of a feature to save and resume games means that players must complete each session in one sitting. Implementing save/load functionality would allow for more flexible play sessions.

UI/UX Considerations

The user interface (UI) and user experience (UX) of our chess game are areas with room for improvement:

- The game lacks advanced UI features such as undoing moves, viewing game history, and selecting from different difficulty levels for AI opponents, which could make the game more user-friendly and adaptable to various player needs.
- The current single-screen setup without menus or settings options restricts the ability to customize gameplay experiences, such as choosing different themes or adjusting sound levels.
- Navigation and interaction within the game are primarily mouse-driven, without support for keyboard shortcuts or touch input, which could enhance accessibility and ease of use.
- The mechanism for highlighting legal moves is basic, relying on simple visual cues that could be enhanced for better clarity and strategic planning.
- Currently, the game lacks audio feedback. Incorporating sound effects for moves, captures, and special notifications (e.g., check or checkmate) could significantly enrich the user experience.
- The aesthetic presentation of the board and pieces is minimalistic. It totally lacks the animation. A more detailed and refined graphical representation could make the game more engaging and visually appealing.

Addressing these limitations presents a pathway toward a more sophisticated chess application that is accessible, engaging, and enjoyable for a wider audience.

Thank you