NATIONAL PUBLIC SCHOOL HSR LAYOUT

ChessExcel

2023-24

Made By:

Mervyn Simon Panicker (11B)

Vansh Aggarwal (11B)

Aarush Reddy (11B)

Index:

INDEX

Certificate

ACKNOWLEDGEMENT:

The successful development of this Python chess game would not have been possible without the invaluable support and guidance of several individuals and institutions.

Firstly, we would like to express our sincere gratitude to our school, National Public School HSR Layout. The stimulating academic environment and resources provided by the school greatly contributed to our ability to undertake this project. We are especially thankful to the principal, Ms. Shefali Tyagi, for her continuous encouragement and support throughout our learning journey.

Our deepest appreciation goes to our dedicated teacher, Ms. Nimi Kumar. Her insightful guidance, insightful feedback, and unwavering belief in our capabilities were instrumental in helping us navigate the challenges of this project and achieve its completion. Her expertise and passion for programming have fostered our interest in the field and equipped us with valuable skills.

We are also grateful to our friends and fellow classmates for their camaraderie and support. Their helpful discussions, brainstorming sessions, and willingness to offer assistance whenever needed proved invaluable. The collaborative learning environment and friendly competition made the process enjoyable and motivated us to strive for excellence.

Finally, we would like to acknowledge the countless online resources and tutorials that provided valuable knowledge and insights throughout the development process. The open-source community and the wealth of information available online played a significant role in helping us overcome challenges and implement various functionalities in this game.

We are truly grateful for the contributions of all these individuals and institutions, and we dedicate this project to them in recognition of their unwavering support.

Vansh Aggarwal, Mervyn Simon & Aarush Reddy

An Overview of Python

Python is a high-level, general-purpose programming language known for its emphasis on code readability and its use of significant indentation. It was created by Guido van Rossum and first released in 1991. Python's open-source community is vast, diverse, and continuously growing. The language has a wide range of libraries and frameworks, making it versatile and popular due to its straightforwardness and uncomplicated syntax.

History:

Python was conceived in the late 1980s, and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands. The development of the language commenced in December 1989, and the first public release, version 0.9.0, was made available in February 1991. Python 1.0 was introduced in 1994, incorporating new functionalities such as lambda, map, filter, and reduce.

Features:

Python is a versatile programming language known for its simplicity, readability, and wide range of applications. Its usability is attributed to its simple syntax, support for multiple programming paradigms, and extensive libraries and frameworks, making it accessible to engineers with various backgrounds. Python's functions, including user-defined functions, enable the division of programs into modules, making the code easier to manage, debug, and scale, and promoting code reuse.

The language's straightforwardness and uncomplicated syntax have contributed to its popularity, as it allows for easy onboarding of engineers with existing expertise in different programming paradigms. Python's usability extends to various applications, including web programming, data analysis, game development, and the creation of system utilities.

In summary, Python's usability is derived from its simple and versatile nature, making it suitable for a wide range of applications and accessible to engineers with diverse backgrounds. Its support for code reuse and program organization further enhances its usability in various development scenarios.

Open Source Community:

Python's open-source community is known for its vastness, diversity, and continuous growth. It provides extensive support through various channels such as newsletters, Slack teams, Discord, and LinkedIn groups. The community offers curated news, articles, new releases, jobs, and more. Engaging in open-source projects is seen as an optimal way for individuals to contribute to and learn from the community.

In summary, Python is a versatile programming language with a rich history, a strong emphasis on community, and a wide range of features and capabilities that have contributed to its widespread adoption and continued growth.

PROJECT SYNOPSIS

Introduction:

The provided Python script presents a sophisticated implementation of a chess game utilizing the Pygame library, offering players a visually appealing and interactive gaming experience. The program is meticulously designed to encompass various functionalities, ensuring adherence to the standard rules of chess while providing users with intuitive gameplay mechanics and a polished graphical interface.

Modules Used:

Central to the program's functionality is the Pygame library, which serves as the foundation for handling graphical rendering, event management, and user input. Additionally, standard Python modules such as `os` are leveraged for file operations, further augmenting the program's versatility and functionality.

Key Features:

Graphical User Interface:

Pygame's robust graphical capabilities enable the creation of a visually appealing game interface, complete with intricate board designs and detailed chess piece representations.

Piece Movement:

Through meticulously crafted logic components, the program facilitates the validation and execution of legal piece movements, adhering to the established rules and constraints of chess gameplay.

Capturing Pieces:

Players can strategically capture opponent pieces by maneuvering their own pieces to occupy occupied squares, effectively simulating the dynamic interplay of chess strategy.

Check Detection:

The program diligently monitors the game state to detect instances of check, ensuring that players remain cognizant of potential threats to their kings and prompting strategic decision-making.

Captured Pieces Display:

A visual representation of captured pieces is prominently displayed alongside the main game board, offering players insight into the progression of the game and the strategic maneuvers undertaken by both sides.

Game Over Handling:

Upon detecting a checkmate condition, the program gracefully concludes the game, declaring the victorious player and affording users the opportunity to restart or exit the game as desired.

Working of the Code:

Initialization:

The script initializes the Pygame module, configures the game window, and loads essential resources such as fonts and piece images, laying the groundwork for subsequent gameplay interactions.

Board and Piece Setup:

Initial chess piece positions are predefined, and corresponding graphical representations are loaded onto the game board, establishing the foundation for subsequent gameplay interactions.

Event Handling:

The program continuously monitors user input events, primarily mouse clicks, to facilitate player interactions with the game, including piece selection, movement execution, and game navigation.

Piece Movement Validation:

Upon user input, the program rigorously validates the legality of requested piece movements, leveraging specialized functions tailored to the unique movement characteristics of each chess piece.

Check Detection:

Through intricate algorithms, the program dynamically evaluates the game state to detect instances of check, intelligently assessing potential threats to the kings of both players and conveying pertinent game state information to the players.

Game Over Handling:

Upon identifying a checkmate condition, the program gracefully concludes the game, declaring the victorious player and presenting users with options to restart or exit the game, ensuring a seamless and satisfying gameplay experience.

Logic Components:

Piece Movement Validation:

Specialized functions are employed to validate the legality of piece movements, accounting for factors such as piece type, current position, and board state to determine valid move options for each piece.

Check Detection:

Through sophisticated algorithms, the program dynamically evaluates the game state to detect instances of check, systematically analyzing potential threats to the kings of both players and triggering appropriate game state updates accordingly.

Game Over Handling:

Upon detecting a checkmate condition, the program initiates the game over sequence, gracefully concluding the game and providing players with pertinent information regarding the victor and available game options.

Prospects:

While the current iteration of the program offers a robust and immersive chess gameplay experience, there exist numerous avenues for future enhancements and expansions:

User Interface Enhancements:

Further refinement of the graphical interface, including additional animations, sound effects, and visual cues, could heighten player immersion and elevate the overall gaming experience.

AI Integration:

Integration of AI opponents with varying difficulty levels could introduce single-player gameplay options, allowing users to test their skills against computer-controlled adversaries.

Multiplayer Support:

Implementation of multiplayer functionality, whether locally or online, would enable players to engage in competitive chess matches with friends or opponents from around the world, fostering a vibrant community of chess enthusiasts.

Game Variant Support:

The program could be extended to support various chess variants or custom rule sets, catering to diverse player preferences and expanding the scope of gameplay possibilities.

In essence, the provided Python script represents a commendable effort in bringing the timeless game of chess to life in the digital realm. Through meticulous design, thoughtful implementation, and a commitment to user engagement, the program succeeds in delivering an immersive and enjoyable chess-playing experience, with ample potential for future growth and enhancement.

SYSTEM REQUIREMENT

Detailed System Requirements for Running the Chess Game Program:

Operating System:

- Windows: Windows 7 or later versions

- macOS: macOS 10.11 (El Capitan) or later versions

- Linux: Any modern distribution with X11 support

Python Interpreter:

- Python 3.x: Ensure that Python 3.x is installed on your system. The program is compatible with Python 3.6, 3.7, 3.8, or later versions.

Pygame Library:

- Pygame: The Pygame library is a prerequisite for running the chess game program. Install Pygame using pip, the Python package manager:

```
pip install Pygame
```

Resource Files:

- **Font Files:** The program utilizes font files for text rendering. Ensure that the font files referenced in the script (`freesansbold.ttf`) are either present in the same directory as the Python script or correctly specified with their file paths.

- **Image Assets:** The program relies on image assets for displaying chess piece graphics. Ensure that the image files (`bq.png`, `bk.png`, `br.png`, `bb.png`, `bn.png`, `bp.png`, `wq.png`, `wk.png`, `wr.png`, `wb.png`, `wn.png`, `wp.png`) are accessible to the program and located in the same directory as the Python script or correctly specified with their file paths.

Hardware Requirements:

- **Processor**: A dual-core CPU or higher is recommended for smooth execution of the program.

- **RAM:** A minimum of 2 GB of RAM is required for optimal performance.

- **Graphics**: The program utilizes basic 2D graphics. An integrated or dedicated GPU capable of rendering 2D graphics is sufficient.

Display Resolution:

- **Minimum Resolution**: The program's graphical interface is optimized for a minimum display resolution of 800x600 pixels. Ensure that your monitor supports this resolution or higher for proper rendering of the game interface.

Input Devices:

- **Mouse**: The program primarily relies on mouse input for player interactions, including piece selection and movement. Ensure that a functional mouse or compatible pointing device is connected to the system for optimal gameplay experience.

Internet Connection (Optional):

- Multiplayer Functionality: If multiplayer functionality or online features are implemented in future updates, an internet connection may be required for accessing online resources and engaging with other players.

By meeting these detailed system requirements, you can ensure smooth execution and optimal performance of the chess game program on your system.

User Manual:

User Manual for Chess Game Program:

1. Introduction:

Welcome to ChessExcel, a Python-based chess game developed by Vansh, Mervyn, and Aarush. This user manual provides a comprehensive guide on how to install, run, and play the chess game on your computer.

2. Installation:

Before running the chess game program, ensure that you have Python and the Pygame library installed on your system. Follow these steps to install the necessary components:

- Install Python: If Python is not already installed on your system, download and install the latest version of Python 3.x from the official Python website (https://www.python.org/).

- Install Pygame: Open a command prompt or terminal and enter the following command to install Pygame using pip, the Python package manager:

```
pip install Pygame
```

3. Running the Program:

Once Python and Pygame are installed, follow these steps to run the ChessExcel program:

- Download the Code: Download the Python script (`chess_game.py`) provided by the developers.

- Open Terminal or Command Prompt: Navigate to the directory where the Python script is saved using the terminal or command prompt.

- Run the Program: Enter the following command to execute the Python script:

```
python chess_game.py
```

4. Game Interface:

Upon running the program, you will be greeted with the ChessExcel game interface. Here's an overview of the different elements on the interface:

- Game Board: The main chessboard occupies the central area of the screen, consisting of 8x8 squares.

- Captured Pieces Area: Located on the right side of the screen, this area displays the pieces captured by each player during the game.

- Status Bar: The status bar at the bottom of the screen provides information about the current turn and game status.

5. Playing the Game:

ChessExcel follows the standard rules of chess. Here's a step-by-step guide on how to play the game:

- Selecting a Piece: To move a piece, click on it with the left mouse button. The selected piece will be highlighted.

- Valid Moves: After selecting a piece, valid move options will be displayed as highlighted squares on the board. Move the cursor to one of these squares to see where the selected piece can be moved.

- Moving a Piece: Click on one of the highlighted squares to move the selected piece to that position. The piece will be relocated, and it will now be the opponent's turn.

- Capturing Pieces: If a piece is moved to a square occupied by an opponent's piece, the opponent's piece will be captured and displayed in the captured pieces area.

- Winning the Game: The game ends when one player successfully places the opponent's king in checkmate, indicating victory.

6. Restarting the Game:

If you wish to restart the game at any point, simply press the "Enter" key on your keyboard after the game is over. This will reset the board and allow you to start a new game.

7. Exiting the Program:

To exit the ChessExcel program, close the window or press the "X" button at the top right corner of the window.

8. Troubleshooting:

If you encounter any issues while running the program or playing the game, ensure that you have met all the system requirements and followed the installation steps correctly. Additionally, refer to any error messages displayed in the terminal or command prompt for troubleshooting assistance.

9. Feedback and Support:

If you have any feedback, suggestions, or encounter any technical issues while using ChessExcel, feel free to reach out to the developers via their contact information provided with the game. They will be happy to assist you and improve the game based on user feedback.

10. Enjoy Playing ChessExcel!

Now that you're familiar with the game interface and how to play ChessExcel, dive in and enjoy the immersive experience of playing chess against your friends or the computer. Have fun strategizing, capturing pieces, and ultimately achieving victory in this classic game of skill and strategy!

MODULES AND FUCNTIONS

1. Pygame Module:

- Purpose: Pygame is a cross-platform set of Python modules designed for writing video games. It provides functionality for handling graphics, sound, and user input, making it ideal for developing interactive applications like games.

- Usage in the Program: The `Pygame` module is used extensively throughout the ChessExcel program for creating the game window, handling user input, rendering graphics, and managing game events.

- Future Applications: Pygame can be utilized for developing various types of games, simulations, educational software, and multimedia applications. Its versatility and ease of use make it suitable for both hobbyist and professional game development projects.

2. os Module:

- Purpose: The `os` module provides a portable way of using operating system-dependent functionality. It allows Python programs to interact with the operating system, including file operations, process management, and environment variables.

- Usage in the Program: In the ChessExcel program, the `os` module may be used for tasks such as file path manipulation, checking file existence, or executing system commands. For example, it may be used to load image files or fonts from the filesystem.

- Future Applications: The `os` module is essential for developing platform-independent applications that need to perform file and system-related operations. It can be utilized in a wide range of software development projects beyond gaming, including system utilities, automation scripts, and web applications.

3. sys Module:

- Purpose: The `sys` module provides access to some variables used or maintained by the Python interpreter and functions that interact with the Python

runtime environment. It allows manipulation of the Python runtime environment, command-line arguments, and system-specific parameters.

- Usage in the Program: In the ChessExcel program, the `sys` module may be used to handle command-line arguments, access system-specific parameters, or manipulate the Python runtime environment. For example, it may be used to exit the program gracefully or retrieve command-line arguments.

- Future Applications: The `sys` module is useful for developing Python scripts that require interaction with the runtime environment or system-specific configurations. It can be utilized in various applications, including system administration tools, command-line utilities, and software development frameworks.

4. Python Standard Library:

- Purpose: The Python Standard Library is a comprehensive set of modules and packages that come with the Python programming language. It provides a wide range of functionality for performing common tasks such as file I/O, networking, data manipulation, and more.

- Usage in the Program: Various modules from the Python Standard Library may be used in the ChessExcel program, depending on the specific requirements. For example, the `os` module for file operations, `random` module for generating random numbers, and `time` module for time-related functionality.

- Future Applications: The Python Standard Library is an invaluable resource for Python developers, offering a rich collection of modules and packages for building diverse applications. It serves as a foundation for developing robust and feature-rich software across different domains, including web development, data science, automation, and more.

5. Font Files:

- Purpose: Font files (e.g., `freesansbold.ttf`) are used to render text and graphical elements with specific fonts and styles in the game interface. They provide visual consistency and enhance the readability of text displayed to the user.

- Usage in the Program: In the ChessExcel program, font files are loaded using Pygame's font module (`pygame.font`) to render text elements such as

status messages, player names, and game prompts. Different fonts and sizes may be used to achieve desired visual effects.

- Future Applications: Font files are essential for graphical user interface (GUI) development in various software applications, including games, desktop applications, websites, and mobile apps. They allow developers to customize the appearance of text-based content and enhance the user experience.

6. Image Assets (Chess Piece Graphics):

- Purpose: Image assets (e.g., `bq.png`, `wk.png`) represent graphical representations of chess pieces (e.g., queen, king) used in the game interface. They provide visual cues to players and enhance the overall aesthetics of the game.

- Usage in the Program: In the ChessExcel program, image assets are loaded using Pygame's image module (`pygame. image`) to display chess pieces on the game board. Different images correspond to different types and colors of chess pieces, allowing players to identify and interact with them.

- Future Applications: Image assets are commonly used in graphic design, multimedia, and user interface development across various software applications. Beyond games, they can be utilized in web design, digital art, educational software, and more to enhance visual communication and engagement.

7. Functions (e.g., check_pawn, check_rook, draw_board):

- Purpose: Functions in the ChessExcel program encapsulate specific tasks or operations, providing modularization, code reuse, and maintainability. Each function serves a distinct purpose related to game logic, graphics rendering, or user interaction.

- Usage in the Program: Functions such as `check_pawn`, `check_rook`, and `draw_board` are called within the program to perform tasks such as validating piece movements, checking for checkmate conditions, and rendering the game board interface, respectively.

- Future Applications: Modular functions facilitate code organization, readability, and extensibility, making it easier to maintain and enhance the program over time. As the ChessExcel program evolves, additional functions

may be added, or existing functions modified to accommodate new features and improvements.

PROGRAM CODE

```
import pygame as pyg
#how to make screen?

pyg.init()

WIDTH = 800
HEIGHT = 800

screen = pyg.display.set_mode([WIDTH, HEIGHT])
pyg.display.set_caption('ChessExcel by Vansh, Mervyn, Aarush')

wpieces = ['rook', 'knight', 'bishop', 'king', 'queen', 'bishop', 'knight', 'rook',
           'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn']
wlocs = [(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)]
bpieces = ['rook', 'knight', 'bishop', 'king', 'queen', 'bishop', 'knight', 'rook',
           'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn']
blocs = [(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)]

cap_pieces_w = []
cap_pieces_b = []

which_turn = 0
selection = 100
chalega_kya = []
```

```
# load in images
bqueen = pyg.image.load(r'bq.png')
bqueen = pyg.transform.scale(bqueen, (80, 80))
bking = pyg.image.load(r'bk.png')
bking = pyg.transform.scale(bking, (80, 80))
brook = pyg.image.load(r'br.png')
brook = pyg.transform.scale(brook, (80, 80))
bbishop = pyg.image.load(r'bb.png')
bbishop = pyg.transform.scale(bbishop, (80, 80))
bknight = pyg.image.load(r'bn.png')
bknight = pyg.transform.scale(bknight, (80, 80))
bpawn = pyg.image.load(r'bp.png')
bpawn = pyg.transform.scale(bpawn, (65, 65))
wqueen = pyg.image.load(r'wq.png')
wqueen = pyg.transform.scale(wqueen, (80, 80))
wking = pyg.image.load(r'wk.png')
wking = pyg.transform.scale(wking, (80, 80))
wrook = pyg.image.load(r'wr.png')
wrook = pyg.transform.scale(wrook, (80, 80))
wbishop = pyg.image.load(r'wb.png')
wbishop = pyg.transform.scale(wbishop, (80, 80))
wknight = pyg.image.load(r'wn.png')
wknight = pyg.transform.scale(wknight, (80, 80))
wpawn = pyg.image.load(r'wp.png')
wpawn = pyg.transform.scale(wpawn, (65, 65))
```

```
white_images = [wpawn, wqueen, wking,
                wknight, wrook, wbishop]

black_images = [bpawn, bqueen, bking,
                bknight, brook, bbishop]

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

counter = 0
winner = ''
game_over = False

def draw_board():
    for i in range(32):
        column = i % 4
        row = i // 4
        if row % 2 == 0:
            pyg.draw.rect(screen, 'blue', [600 - (column * 200), row * 100, 100, 100])
        else:
            pyg.draw.rect(screen, 'blue', [700 - (column * 200), row * 100, 100, 100])
        pyg.draw.rect(screen, 'white', [0, 800, WIDTH, 75])
```

```
77  def draw_pieces():
78      for i in range(len(wpieces)):
79          index = piece_list.index(wpieces[i])
80          if wpieces[i] == 'pawn':
81              screen.blit(
82                  wpawn, (wlocs[i][0] * 100 + 22, wlocs[i][1] * 100 + 30))
83          else:
84              screen.blit(white_images[index], (wlocs[i]
85                          [0] * 100 + 10, wlocs[i][1] * 100 + 10))
86
87      for i in range(len(bpieces)):
88          index = piece_list.index(bpieces[i])
89          if bpieces[i] == 'pawn':
90              screen.blit(
91                  bpawn, (blocs[i][0] * 100 + 22, blocs[i][1] * 100 + 30))
92          else:
93              screen.blit(black_images[index], (blocs[i]
94                          [0] * 100 + 10, blocs[i][1] * 100 + 10))
95
```

```
97  def check_options(pieces, locations, turn):
98      moves_list = []
99      all_moves_list = []
100     for i in range((len(pieces))):
101         location = locations[i]
102         piece = pieces[i]
103         if piece == 'pawn':
104             moves_list = check_pawn(location, turn)
105         elif piece == 'rook':
106             moves_list = check_rook(location, turn)
107         elif piece == 'knight':
108             moves_list = check_knight(location, turn)
109         elif piece == 'bishop':
110             moves_list = check_bishop(location, turn)
111         elif piece == 'queen':
112             moves_list = check_queen(location, turn)
113         elif piece == 'king':
114             moves_list = check_king(location, turn)
115         all_moves_list.append(moves_list)
116     return all_moves_list
117
```

```
117
118
119  def check_king(position, color):
120      moves_list = []
121      if color == 'white':
122          enemies_list = blocs
123          friends_list = wlocs
124      else:
125          friends_list = blocs
126          enemies_list = wlocs
127      targets = [(1, 0), (1, 1), (1, -1), (-1, 0),
128                 (-1, 1), (-1, -1), (0, 1), (0, -1)]
129      for i in range(8):
130          target = (position[0] + targets[i][0], position[1] + targets[i][1])
131          if target not in friends_list and 0 <= target[0] <= 7 and 0 <= target[1] <= 7:
132              moves_list.append(target)
133      return moves_list
134
```

```
135
136  def check_queen(position, color):
137      moves_list = check_bishop(position, color)
138      second_list = check_rook(position, color)
139      for i in range(len(second_list)):
140          moves_list.append(second_list[i])
141      return moves_list
142
143
144  def check_bishop(position, color):
145      moves_list = []
146      if color == 'white':
147          enemies_list = blocs
148          friends_list = wlocs
149      else:
150          friends_list = blocs
151          enemies_list = wlocs
152      for i in range(4):
153          path = True
154          chain = 1
155          if i == 0:
156              x = 1
157              y = -1
158          elif i == 1:
159              x = -1
160              y = -1
161          elif i == 2:
162              x = 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
```

```
180    def check_rook(position, color):
181        moves_list = []
182        if color == 'white':
183            enemies_list = blocs
184            friends_list = wlocs
185        else:
186            friends_list = blocs
187            enemies_list = wlocs
188        for i in range(4):
189            path = True
190            chain = 1
191            if i == 0:
192                x = 0
193                y = 1
194            elif i == 1:
195                x = 0
196                y = -1
197            elif i == 2:
198                x = 1
199                y = 0
200            else:
201                x = -1
202                y = 0
```

```
202                y = 0
203            while path:
204                if (position[0] + (chain * x), position[1] + (chain * y)) not in friends_list and \
205                        0 <= position[0] + (chain * x) <= 7 and 0 <= position[1] + (chain * y) <= 7:
206                    moves_list.append(
207                        (position[0] + (chain * x), position[1] + (chain * y)))
208                    if (position[0] + (chain * x), position[1] + (chain * y)) in enemies_list:
209                        path = False
210                    chain += 1
211                else:
212                    path = False
213        return moves_list
214
```

```
def check_pawn(position, color):
    moves_list = []
    if color == 'white':
        if (position[0], position[1] + 1) not in wlocs and \
                (position[0], position[1] + 1) not in blocs and position[1] < 7:
            moves_list.append((position[0], position[1] + 1))
        if (position[0], position[1] + 2) not in wlocs and \
                (position[0], position[1] + 2) not in blocs and position[1] == 1:
            moves_list.append((position[0], position[1] + 2))
        if (position[0] + 1, position[1] + 1) in blocs:
            moves_list.append((position[0] + 1, position[1] + 1))
        if (position[0] - 1, position[1] + 1) in blocs:
            moves_list.append((position[0] - 1, position[1] + 1))
    else:
        if (position[0], position[1] - 1) not in wlocs and \
                (position[0], position[1] - 1) not in blocs and position[1] > 0:
            moves_list.append((position[0], position[1] - 1))
        if (position[0], position[1] - 2) not in wlocs and \
                (position[0], position[1] - 2) not in blocs and position[1] == 6:
            moves_list.append((position[0], position[1] - 2))
        if (position[0] + 1, position[1] - 1) in wlocs:
            moves_list.append((position[0] + 1, position[1] - 1))
        if (position[0] - 1, position[1] - 1) in wlocs:
            moves_list.append((position[0] - 1, position[1] - 1))
    return moves_list
```

```
243 def check_knight(position, color):
244     moves_list = []
245     if color == 'white':
246         enemies_list = blocs
247         friends_list = wlocs
248     else:
249         friends_list = blocs
250         enemies_list = wlocs
251     targets = [(1, 2), (1, -2), (2, 1), (2, -1),
252                (-1, 2), (-1, -2), (-2, 1), (-2, -1)]
253     for i in range(8):
254         target = (position[0] + targets[i][0], position[1] + targets[i][1])
255         if target not in friends_list and 0 <= target[0] <= 7 and 0 <= target[1] <= 7:
256             moves_list.append(target)
257     return moves_list
258
```

```
260  def check_valid_moves():
261      if which_turn < 2:
262          options_list = white_options
263      else:
264          options_list = black_options
265      valid_options = options_list[selection]
266      return valid_options
267
268
```

```
# main game loop
black_options = check_options(bpieces, blocs, 'black')
print ("Black options are :", black_options)
white_options = check_options(wpieces, wlocs, 'white')
print ("White options are ", white_options)
run = True
while run:


    screen.fill('dark gray')
    draw_board()
    draw_pieces()
```

```
if selection != 100:
    chalega_kya = check_valid_moves()
for event in pyg.event.get():
    if event.type == pyg.QUIT:
        print (" END : quit detected")
        run = False
    if event.type == pyg.MOUSEBUTTONDOWN and event.button == 1 and not game_over:
        print ("Mouse event, and game not ended")
        x_coord = event.pos[0] // 100
        y_coord = event.pos[1] // 100
        click_coords = (x_coord, y_coord)
        print (" Detected the cordinates of click" , click_coords)
        if which_turn <= 1:
            if click_coords == (8, 8) or click_coords == (9, 8):
                winner = 'black'
            if click_coords in wlocs:
                selection = wlocs.index(click_coords)
                if which_turn == 0:
                    which_turn = 1
            if click_coords in chalega_kya and selection != 100:
                wlocs[selection] = click_coords
                if click_coords in blocs:
                    black_piece = blocs.index(click_coords)
                    cap_pieces_w.append(bpieces[black_piece])
                    if bpieces[black_piece] == 'king':
                        winner = 'white'
```

```
                        winner = 'white'
                    bpieces.pop(black_piece)
                    blocs.pop(black_piece)
                black_options = check_options(
                    bpieces, blocs, 'black')
                white_options = check_options(
                    wpieces, wlocs, 'white')
                which_turn = 2
                selection = 100
                chalega_kya = []
        if which_turn > 1:
            if click_coords == (8, 8) or click_coords == (9, 8):
                winner = 'white'
            if click_coords in blocs:
                selection = blocs.index(click_coords)
                if which_turn == 2:
                    which_turn = 3
            if click_coords in chalega_kya and selection != 100:
                blocs[selection] = click_coords
                if click_coords in wlocs:
                    white_piece = wlocs.index(click_coords)
                    cap_pieces_b.append(wpieces[white_piece])
                    if wpieces[white_piece] == 'king':
                        winner = 'black'
                    wpieces.pop(white_piece)
```

```
331                     wpieces.pop(white_piece)
332                     wlocs.pop(white_piece)
333                 black_options = check_options(
334                     bpieces, blocs, 'black')
335                 white_options = check_options(
336                     wpieces, wlocs, 'white')
337                 which_turn = 0
338                 selection = 100
339                 chalega_kya = []
340     if event.type == pyg.KEYDOWN and game_over:
341         print (" Game is over. Lets declare");
342         if event.key == pyg.K_RETURN:
343             game_over = False
344             winner = ''
345             wpieces = ['rook', 'knight', 'bishop', 'king', 'queen', 'bishop', 'knight', 'rook',
346                        'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn']
347             wlocs = [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0),
348                      (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1)]
349             bpieces = ['rook', 'knight', 'bishop', 'king', 'queen', 'bishop', 'knight', 'rook',
350                        'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn', 'pawn']
351             blocs = [(0, 7), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7),
352                      (0, 6), (1, 6), (2, 6), (3, 6), (4, 6), (5, 6), (6, 6), (7, 6)]
353             cap_pieces_w = []
```

```
                cap_pieces_w = []
                cap_pieces_b = []
                which_turn = 0
                selection = 100
                chalega_kya = []
                black_options = check_options(
                    bpieces, blocs, 'black')
                white_options = check_options(
                    wpieces, wlocs, 'white')

    if winner != '':
        game_over = True
        print("The winner is: " , winner)
        break

    pyg.display.flip()

pyg.quit()
```

```
# Previous Version of the code:

"""
x = True
y = True
r = 0
master_board = {
    "a1": "wr1",  "b1":"wn1",  "c1":"wb1",  "d1":"wq",  "e1":"wk",  "f1":"wb2",  "g1":"wn2",  "h1":"wr2",
    "a2": "wp1",  "b2":"wp2",  "c2":"wp3",  "d2":"wp4",  "e2":"wp5",  "f2":"wp6",  "g2":"wp7",  "h2":"wp8",
    "a3": "", "b3":"", "c3":"", "d3":"", "e3":"", "f3":"", "g3":"", "h3":"",
    "a4": "", "b4":"", "c4":"", "d4":"", "e4":"", "f4":"", "g4":"", "h4":"",
    "a5": "", "b5":"", "c5":"", "d5":"", "e5":"", "f5":"", "g5":"", "h5":"",
    "a6": "", "b6":"", "c6":"", "d6":"", "e6":"", "f6":"", "g6":"", "h6":"",
    "a7": "bp1",   "b7":"bp2",   "c7":"bp3",   "d7":"bp4",   "e7":"bp5",   "f7":"bp6",   "g7":"bp7",   "h7":"bp8",
    "a8": "br1",   "b8":"bn1",   "c8":"bb1",   "d8":"bq",   "e8":"bk",   "f8":"bb2",   "g8":"bn2",   "h8":"br2"
}

def w_move_pawn(master_board, piece_position, w_moved_position):
```

```
def w_move_pawn(master_board, piece_position, w_moved_position):
    piece_colour = "w"
    row_difference = ord(w_moved_position[1]) - ord(piece_position[1])
    if row_difference == 1 and w_moved_position[0] == piece_position[0]:
        return True
    elif row_difference == 2 and piece_position[1] == "2" and w_moved_position[0] == piece_position[0] and master_board[chr(ord(piece_position
        return True
    elif row_difference == 1 and abs(ord(w_moved_position[0]) - ord(piece_position[0])) == 1 and master_board[w_moved_position][0] != piece_co
        return True
    else:
        return False
```

```
def w_move_rook(master_board, piece_position, w_moved_position):
    piece_colour = "w"
    if piece_position[0] == w_moved_position[0]:
        return True
    elif piece_position[1]== w_moved_position[1]:
        return True
    else:
        return False


```

```
414
415  def w_move_knight(master_board, piece_position, w_moved_position):
416      piece_colour = "w"
417      row_difference = abs(ord(w_moved_position[1]) - ord(piece_position[1]))
418      column_difference = abs(ord(w_moved_position[0]) - ord(piece_position[0]))
419      if (row_difference == 2 and column_difference == 1) or (row_difference == 1 and column_difference == 2):
420          if master_board[w_moved_position] == "" or master_board[w_moved_position][0] != piece_colour:
421              return True  # Valid move
422      else:
423          return False  # Invalid move
424
```

```
425
426  def w_move_bishop(master_board, piece_position, w_moved_position):
427      piece_colour = "w"
428      row_difference = abs(ord(w_moved_position[1]) - ord(piece_position[1]))
429      column_difference = abs(ord(w_moved_position[0]) - ord(piece_position[0]))
430
431      if row_difference == column_difference:
432          row_increment = 1 if w_moved_position[1] > piece_position[1] else -1
433          col_increment = 1 if w_moved_position[0] > piece_position[0] else -1
434          current_row = ord(piece_position[1]) + row_increment
435          current_col = ord(piece_position[0]) + col_increment
436          while current_row != ord(w_moved_position[1]):
437              if master_board[chr(current_row) + chr(current_col)] != "":
438                  return False
439              current_row += row_increment
440              current_col += col_increment
441
442          if master_board[w_moved_position] == "" or master_board[w_moved_position][0] != piece_colour:
443              return True
444              # Valid move
445          else:
446              return False
447      else:
448          return False
449
```

```
450
451  def w_move_queen(master_board, piece_position, w_moved_position):
452      piece_colour = "w"
453
454  def w_move_king(master_board, piece_position, w_moved_position):
455      piece_colour = "w"
456
457  def b_move_pawn(master_board, piece_position, b_moved_position):
458      piece_colour = "b"
459
460      row_difference = ord(b_moved_position[1]) - ord(piece_position[1])
461
462      if row_difference == -1 and b_moved_position[0] == piece_position[0]:
463          return True
464      elif row_difference == -2 and piece_position[1] == "7" and b_moved_position[0] == piece_position[0] and master_board[chr(ord(piece_positio
465          return True
466      elif row_difference == -1 and abs(ord(b_moved_position[0]) - ord(piece_position[0])) == 1 and master_board[b_moved_position][0] != piece_c
467          return True
468      else:
469          return False
```

```
def b_move_rook(master_board, piece_position, b_moved_position):
    piece_colour = "b"
    if piece_position[0] == b_moved_position[0]:
        return True
    elif piece_position[1]== b_moved_position[1]:
        return True
    else:
        return False

def b_move_knight(master_board, piece_position, b_moved_position):
    piece_colour = "b"
    row_difference = abs(ord(b_moved_position[1]) - ord(piece_position[1]))
    column_difference = abs(ord(b_moved_position[0]) - ord(piece_position[0]))
    if (row_difference == 2 and column_difference == 1) or (row_difference == 1 and column_difference == 2):
        if master_board[b_moved_position] == "" or master_board[b_moved_position][0] != piece_colour:
            return True
    else:
        return False
```

```
def b_move_bishop(master_board, piece_position, b_moved_position):
    piece_colour = "b"
    row_difference = abs(ord(w_moved_position[1]) - ord(piece_position[1]))
    column_difference = abs(ord(w_moved_position[0]) - ord(piece_position[0]))

    if row_difference == column_difference:
        if w_moved_position[1] > piece_position[1]:
            row_increment = 1
        else:
            row_increment = -1
        col_increment = 1 if w_moved_position[0] > piece_position[0] else -1
        current_row = ord(piece_position[1]) + row_increment
        current_col = ord(piece_position[0]) + col_increment
        while current_row != ord(w_moved_position[1]):
            if master_board[chr(current_row) + chr(current_col)] != "":
                return False
            current_row += row_increment
            current_col += col_increment

        if master_board[w_moved_position] == "" or master_board[w_moved_position][0] != piece_colour:
            return True
            # Valid move
        else:
            return False
    else:
        return False
```

```
def b_move_queen(master_board, piece_position, b_moved_position):
    piece_colour = "b"

def b_move_king(master_board, piece_position, b_moved_position):
    piece_colour = "b"
```

```
while r != "End": #main game loop
    while x == True:
        w_moved_piece = input("Enter the white piece to be moved to: ")
        w_moved_position = input("Enter position to be moved to: ")

        for i in list(master_board.keys()):
            if master_board[i] == w_moved_piece:
                piece_position = i
        piece_type = w_moved_piece[1]  #piece type from the piece name

        if w_moved_position in list(master_board.keys()): # in the box check
            if w_moved_piece in list(master_board.values()):
                if master_board[w_moved_position][0] != "w":
                    if piece_type == "p":
                        validity = w_move_pawn(master_board, piece_position, w_moved_position)
                        if validity == True:
                            master_board[w_moved_position] = w_moved_piece
                            print(master_board)
                            break
                        else:
                            print("Invalid Move)")
                            continue

```

```
                elif piece_type == "r":
                    validity = w_move_rook(master_board, piece_position, w_moved_position)
                    if validity == True:
                        master_board[w_moved_position] = w_moved_piece
                        print(master_board)
                        break
                    else:
                        print("Invalid Move)")
                        continue
                elif piece_type == "n":
                    validity = w_move_knight(master_board, piece_position, w_moved_position)
                    if validity == True:
                        master_board[w_moved_position] = w_moved_piece
                        print(master_board)
                        break
                    else:
                        print("Invalid Move)")
                        continue
                elif piece_type == "k":
                    validity = w_move_king(master_board, piece_position, w_moved_position)
                    if validity == True:
                        master_board[w_moved_position] = w_moved_piece
                        print(master_board)
                        break
                    else:
                        print("Invalid Move)")
                        continue
                elif piece_type == "b":
                    validity = w_move_bishop(master_board, piece_position, w_moved_position)
                    if validity == True:
                        master_board[w_moved_position] = w_moved_piece
                        print(master_board)
                        break
                    else:
                        print("Invalid Move)")
                        continue
                elif piece_type == "q":
```

```
                        continue
                    elif piece_type == "q":
                        validity = w_move_queen(master_board, piece_position, w_moved_position)
                        if validity == True:
                            master_board[w_moved_position] = w_moved_piece
                            print(master_board)
                            break
                        else:
                            print("Invalid Move)")
                            continue
                else:
                    print ("Error")
                    print ("Please Enter again 1")
                    continue # figure out a way to send code back to line 20
            else:
                print ("Error")
                print ("Please Enter again ")
                continue # figure out a way to send code back to line 20
        else:
            print ("Error")
            print ("Please Enter again: ") # figure out a way to send code back to line 20
            continue


    while y == True:
        b_moved_piece = input ("Enter the white piece to be moved to: ")
        b_moved_position=input("Enter position to be moved to: ")

        for i in list(master_board.keys()):
            if master_board[i] == b_moved_piece:
                piece_position = i
        piece_type = b_moved_piece[1]  #piece type from the piece name

        if b_moved_position in list(master_board.keys()): # in the box check
            if b_moved_piece in list(master_board.values()):
                if master_board[b_moved_position][0] != "b":

                    if piece_type == "p":
                        validity = b_move_pawn(master_board, piece_position, b_moved_position)
```

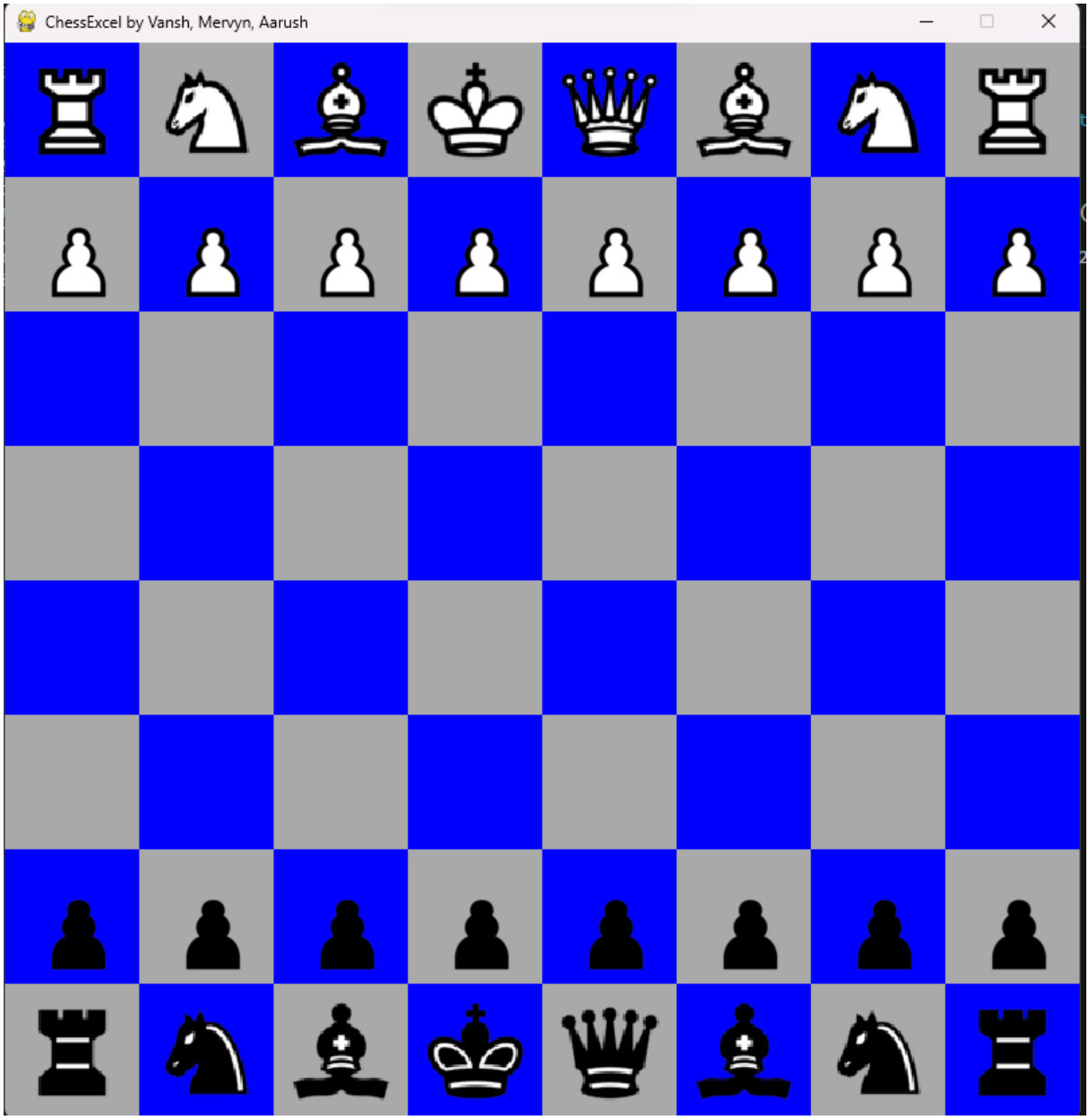
```
                if piece_type == "p":
                    validity = b_move_pawn(master_board, piece_position, b_moved_position)
                    if validity == True:
                        master_board[w_moved_position] = w_moved_piece
                        print(master_board)
                        break
                    else:
                        print("Invalid Move)")
                        continue
                elif piece_type == "r":
                    validity = b_move_rook(master_board, piece_position, b_moved_position)
                    if validity == True:
                        master_board[w_moved_position] = w_moved_piece
                        print(master_board)
                        break
                    else:
                        print("Invalid Move)")
                        continue
                elif piece_type == "n":
                    b_move_knight(master_board, piece_position, b_moved_position)
                    if validity == True:
                        master_board[w_moved_position] = w_moved_piece
                        print(master_board)
                        break
                    else:
                        print("Invalid Move)")
                        continue
                elif piece_type == "k":
                    b_move_king(master_board, piece_position, b_moved_position)
                    if validity == True:
                        master_board[w_moved_position] = w_moved_piece
                        print(master_board)
                        break
                    else:
                        print("Invalid Move)")
                        continue
                elif piece_type == "b":
                    b_move_bishop(master_board, piece_position, b_moved_position)
                    if validity == True:
```

```
                    elif piece_type == "b":
                        b_move_bishop(master_board, piece_position, b_moved_position)
                        if validity == True:
                            master_board[w_moved_position] = w_moved_piece
                            print(master_board)
                            break
                        else:
                            print("Invalid Move)")
                            continue
                    elif piece_type == "q":
                        b_move_queen(master_board, piece_position, b_moved_position)
                        if validity == True:
                            master_board[w_moved_position] = w_moved_piece
                            print(master_board)
                            break
                        else:
                            print("Invalid Move)")
                            continue
                else:
                    print ("Error")
                    print ("Please Enter again 1")
                    continue # figure out a way to send code back to line 20
            else:
                print ("Error")
                print ("Please Enter again ")
                continue # figure out a way to send code back to line 20
        else:
            print ("Error")
            print ("Please Enter again: ") # figure out a way to send code back to line 20
            continue


    """
```

PROGRAM OUTPUT

Start Screen:

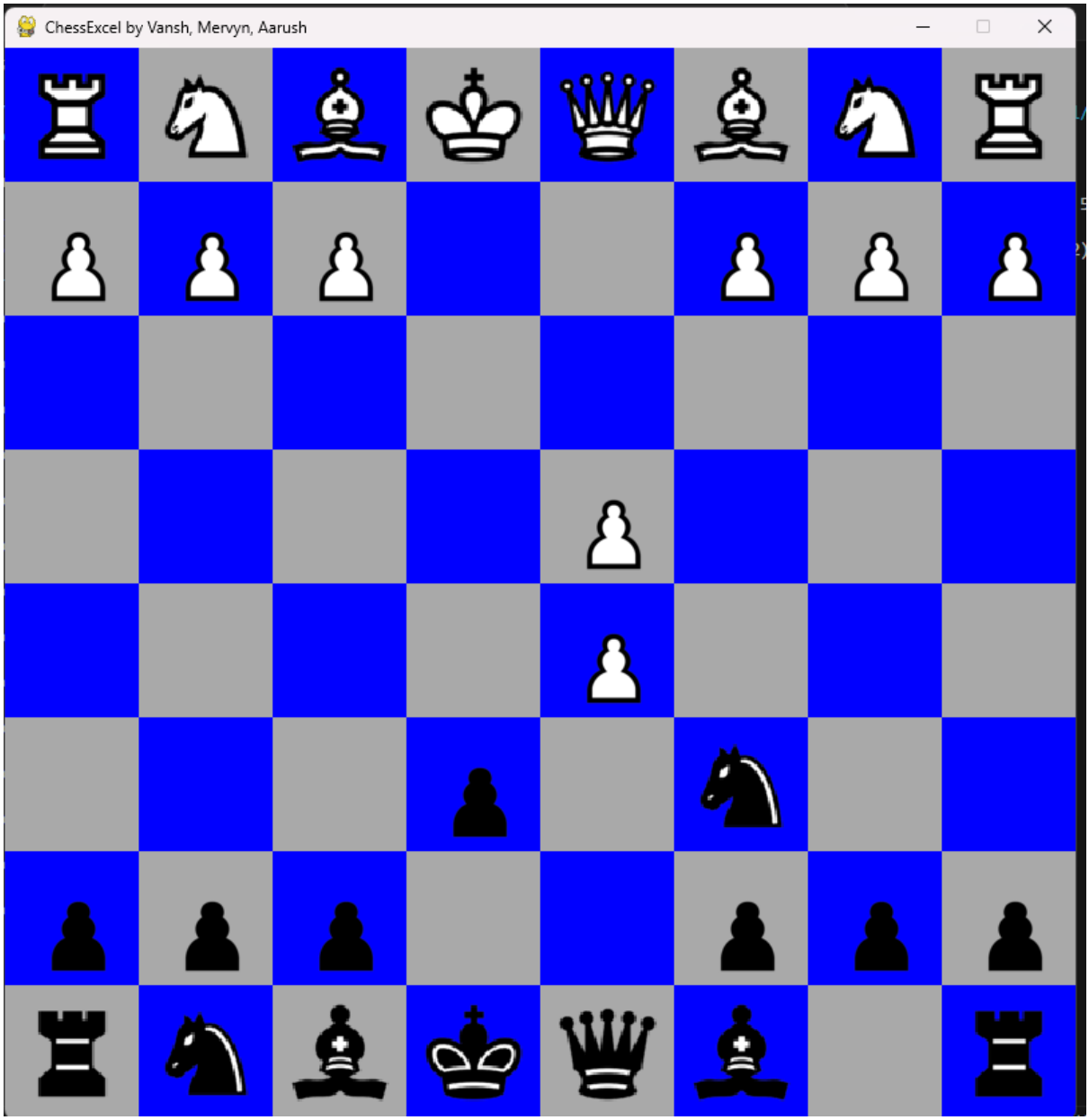


Logging:

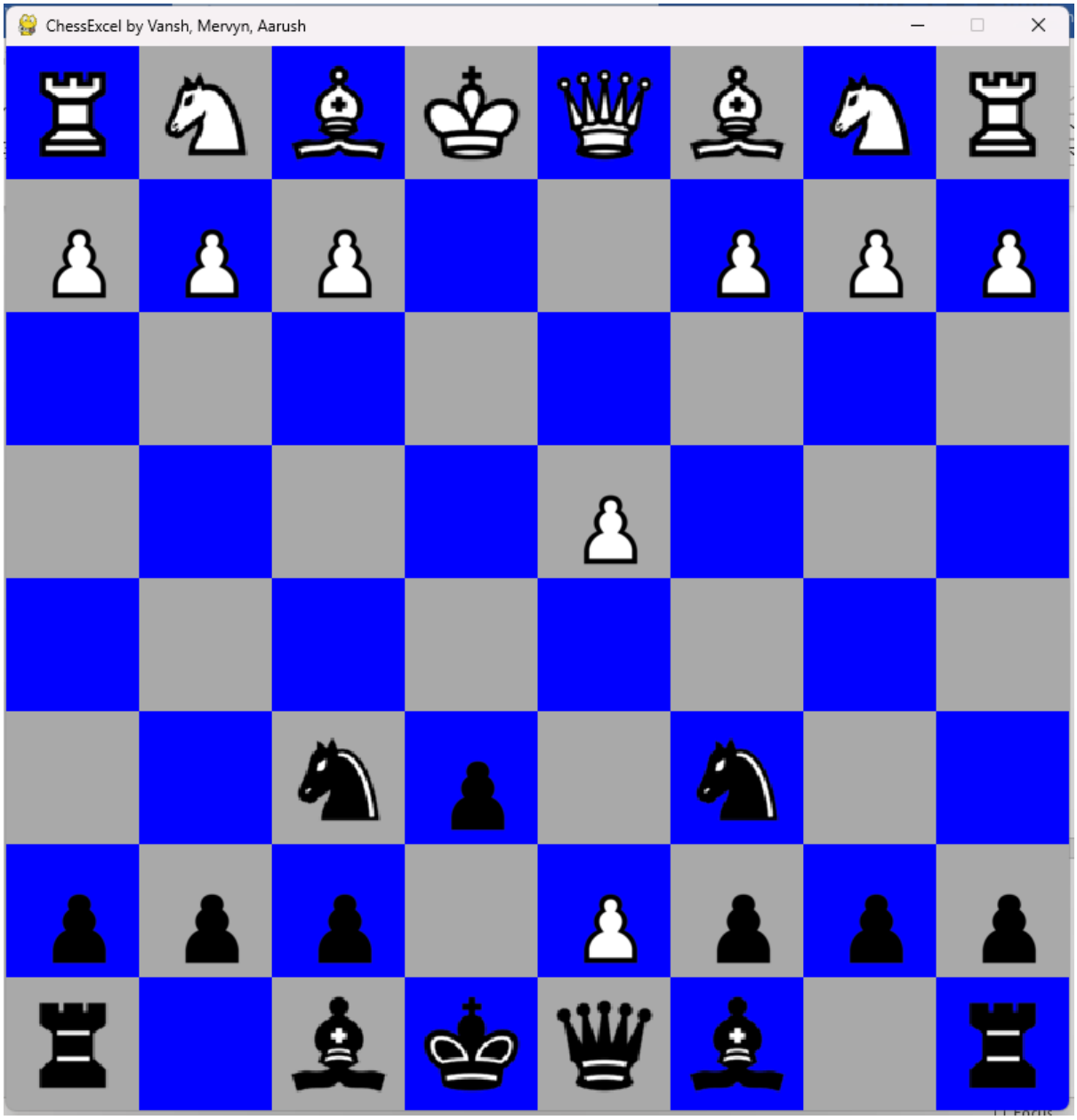
```
PS C:\Users\Manav\Desktop\chessexcel> & C:/Users/Manav/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/Manav/Desktop/chessexcel/chessexcel-Last-commit-for-grade-11/11
th grade project/Chess Excel game copy 2.py"
pygame 2.5.2 (SDL 2.28.3, Python 3.11.7)
Hello from the pygame community. https://www.pygame.org/contribute.html
Black options are : [[], [(2, 5), (0, 5)], [], [], [], [], [(7, 5), (5, 5)], [], [(0, 5), (0, 4)], [(1, 5), (1, 4)], [(2, 5), (2, 4)], [(3, 5), (3, 4)], [(4, 5), (4, 4)], [(5, 5),
 (5, 4)], [(6, 5), (6, 4)], [(7, 5), (7, 4)]]
White options are  [[], [(2, 2), (0, 2)], [], [], [], [], [(7, 2), (5, 2)], [], [(0, 2), (0, 3)], [(1, 2), (1, 3)], [(2, 2), (2, 3)], [(3, 2), (3, 3)], [(4, 2), (4, 3)], [(5, 2),
(5, 3)], [(6, 2), (6, 3)], [(7, 2), (7, 3)]]
Mouse event, and game not ended
 Detected the cordinates of click (4, 6)
Mouse event, and game not ended
 Detected the cordinates of click (3, 1)
Mouse event, and game not ended
 Detected the cordinates of click (3, 3)
Mouse event, and game not ended
 Detected the cordinates of click (4, 6)
Mouse event, and game not ended
 Detected the cordinates of click (4, 4)
Mouse event, and game not ended
 Detected the cordinates of click (3, 3)
Mouse event, and game not ended
 Detected the cordinates of click (4, 4)
Mouse event, and game not ended
 Detected the cordinates of click (5, 6)
Mouse event, and game not ended
 Detected the cordinates of click (5, 5)
Mouse event, and game not ended
 Detected the cordinates of click (4, 4)
Mouse event, and game not ended
 Detected the cordinates of click (5, 5)
Mouse event, and game not ended
 Detected the cordinates of click (5, 5)
Mouse event, and game not ended
 Detected the cordinates of click (5, 6)
Mouse event, and game not ended
 Detected the cordinates of click (3, 6)
Mouse event, and game not ended
 Detected the cordinates of click (3, 5)
```

```
Mouse event, and game not ended
 Detected the cordinates of click (3, 0)
Mouse event, and game not ended
 Detected the cordinates of click (3, 0)
Mouse event, and game not ended
 Detected the cordinates of click (4, 0)
Mouse event, and game not ended
 Detected the cordinates of click (3, 0)
Mouse event, and game not ended
 Detected the cordinates of click (4, 0)
Mouse event, and game not ended
 Detected the cordinates of click (3, 1)
Mouse event, and game not ended
 Detected the cordinates of click (3, 1)
Mouse event, and game not ended
 Detected the cordinates of click (5, 6)
Mouse event, and game not ended
 Detected the cordinates of click (4, 7)
Mouse event, and game not ended
 Detected the cordinates of click (6, 7)
Mouse event, and game not ended
 Detected the cordinates of click (5, 5)
Mouse event, and game not ended
 Detected the cordinates of click (3, 1)
Mouse event, and game not ended
 Detected the cordinates of click (5, 3)
Mouse event, and game not ended
 Detected the cordinates of click (4, 7)
Mouse event, and game not ended
 Detected the cordinates of click (4, 6)
Mouse event, and game not ended
 Detected the cordinates of click (5, 3)
Mouse event, and game not ended
 Detected the cordinates of click (3, 5)
Mouse event, and game not ended
 Detected the cordinates of click (1, 6)
Mouse event, and game not ended
 Detected the cordinates of click (1, 5)
Mouse event, and game not ended
 Detected the cordinates of click (3, 5)
Mouse event, and game not ended
 Detected the cordinates of click (3, 6)
The winner is:  white
PS C:\Users\Manav\Desktop\chessexcel>
```

Movement of Pieces:



Declaration of Winner:




```
Mouse event, and game not ended
 Detected the cordinates of click (3, 6)
Mouse event, and game not ended
 Detected the cordinates of click (4, 6)
Mouse event, and game not ended
 Detected the cordinates of click (3, 7)
The winner is:  white
```

LIMITATIONS AND FUTURE PLANS:

Limitations:

1. Single-Player Mode: The current version of ChessExcel only supports a single-player mode, where the player competes against an AI opponent. It lacks support for multiplayer functionality, such as online gameplay or local multiplayer.

2. Limited AI Complexity: The AI opponent implemented in ChessExcel has a basic level of intelligence and may not provide a challenging experience for experienced chess players. Improving the AI algorithm to enhance its strategic capabilities and decision-making process is necessary.

3. User Interface Complexity: The user interface of ChessExcel, while functional, may be considered simplistic compared to modern game interfaces. Enhancements such as improved graphics, animations, and user interactions could enhance the overall user experience.

4. Missing Features: Some essential chess features are missing in the current version of ChessExcel, such as castling, en-passant, and pawn promotion. Adding these features would make the game more comprehensive and enjoyable.

Future Plans:

1. Multiplayer Support: Introduce multiplayer support, allowing players to compete against each other online or locally. Implement features such as matchmaking, player rankings, and game lobbies to enhance the multiplayer experience.

2. AI Improvement: Enhance the AI algorithm to make it more challenging and adaptable. Implement advanced techniques such as minimax with alpha-beta pruning, machine learning, or neural networks to create a stronger AI opponent.

3. User Interface Enhancement: Improve the game's user interface by adding visually appealing graphics, animations, and sound effects. Enhance user interactions, such as drag-and-drop piece movement, tooltips, and customizable settings.

4. Additional Features: Implement missing chess features such as castling, en passant, pawn promotion, and undo/redo functionality. Introduce customizable game modes, difficulty levels, and game variants to cater to a broader audience.

Steps to Achieve Future Plans:

1. Research and Development: Conduct research on multiplayer networking protocols, AI algorithms, and user interface design principles. Experiment with different techniques and technologies to identify the most suitable solutions.

2. Prototyping and Testing: Create prototypes of new features and functionalities to validate their feasibility and effectiveness. Conduct extensive testing to identify and address any issues or limitations.

3. Community Feedback: Gather feedback from the ChessExcel community, including players, enthusiasts, and developers. Incorporate user suggestions and preferences into the development process to ensure that the game meets their expectations.

4. Continuous Improvement: Iteratively improve the game based on user feedback, technological advancements, and industry best practices. Release regular updates and patches to address bugs, introduce new features, and optimize performance.

Integration with a Database:

1. Database Selection: Choose a suitable database technology for storing game data, player profiles, leaderboard information, and other relevant information. Options may include relational databases (e.g., MySQL, PostgreSQL), NoSQL databases (e.g., MongoDB, Firebase), or cloud-based solutions.

2. Database Schema Design: Design the database schema to accommodate the required data structures and relationships. Define tables for storing game states, player profiles, match history, achievements, and other relevant entities.

3. Integration with Python: Use database libraries or frameworks compatible with Python (e.g., SQLAlchemy, Django ORM) to establish connections, execute queries, and perform CRUD (Create, Read, Update, Delete) operations on the database.

4. Data Persistence: Implement mechanisms for persisting game data, such as saving and loading game states, player progress, and configuration settings. Ensure data integrity, consistency, and security through proper error handling and validation.

5. Online Features: Integrate online features such as player authentication, account management, matchmaking, and leaderboards with the database. Implement APIs or web services for communication between the game client and server-side components.

6. Scalability and Performance: Optimize database performance and scalability to handle large volumes of concurrent users, game data, and transactions. Use techniques such as indexing, caching, sharding, and replication to improve responsiveness and reliability.

7. Security Considerations: Implement robust security measures to protect sensitive data, prevent unauthorized access, and mitigate common security

threats (e.g., SQL injection, cross-site scripting). Apply encryption, authentication, and access control mechanisms as needed.

8. Monitoring and Maintenance: Set up monitoring tools and procedures to monitor database performance, usage patterns, and potential issues. Perform regular maintenance tasks such as backups, updates, and optimizations to ensure the database remains reliable and efficient.

BIBLIOGRAPHY

1. 1. Adams, J. (2014). "Invent Your Own Computer Games with Python" - A free online book that teaches programming fundamentals using Python and Pygame. Available at: http://inventwithpython.com/invent4thed/

2. Swigert, A. (2017). "Pygame Documentation" - The official documentation for the Pygame library, providing comprehensive guides, tutorials, and references for game development with Python and Pygame. Available at: https://www.pygame.org/docs/

3. GeeksforGeeks. (n.d.). "GeeksforGeeks - Computer Science Tutorials" - An online platform providing tutorials, articles, and coding challenges covering various topics in computer science, algorithms, data structures, and programming languages, including Python. Retrieved from https://www.geeksforgeeks.org/


THANK
YOU!