import tkinter as tk  
import chess  
from PIL import Image, ImageTk  
  
# Initialize the chess board  
board = chess.Board()  
  
# Create a dictionary for piece image paths  
piece\_images = {  
 "K": "images/chess\_klt60.png", "Q": "images/chess\_qlt60.png", "R": "images/chess\_rlt60.png", "B": "images/chess\_blt60.png", "N": "images/chess\_nlt60.png", "P": "images/chess\_plt60.png",  
 "k": "images/chess\_kdt60.png", "q": "images/chess\_qdt60.png", "r": "images/chess\_rdt60.png", "b": "images/chess\_bdt60.png", "n": "images/chess\_ndt60.png", "p": "images/chess\_pdt60.png"  
}  
  
# Set up the window  
root = tk.Tk()  
root.title("TWO PLAYERS CHESS GAME")  
  
# Define a square size for the board  
SQUARE\_SIZE = 60  
canvas = tk.Canvas(root, width=8 \* SQUARE\_SIZE + 250, height=8 \* SQUARE\_SIZE + 50) # Extra space for labels and captured pieces  
canvas.pack(side=tk.LEFT)  
  
# Create a frame for captured pieces  
right\_frame = tk.Frame(root)  
right\_frame.pack(side=tk.RIGHT, padx=20)  
  
# Store the references to the images  
image\_references = {}  
  
# Move history stack for undo  
move\_history = []  
  
# Captured pieces dictionaries  
captured\_white = []  
captured\_black = []  
  
# Function to load the piece images  
def load\_piece\_image(piece):  
 if piece is None:  
 return None  
 image\_path = piece\_images.get(piece.symbol(), "")  
 if image\_path:  
 if image\_path not in image\_references:  
 image = Image.open(image\_path)  
 image = image.resize((SQUARE\_SIZE, SQUARE\_SIZE), Image.Resampling.LANCZOS)  
 image\_references[image\_path] = ImageTk.PhotoImage(image)  
 return image\_references[image\_path]  
 return None  
  
# Function to draw the chessboard  
def draw\_board():  
 canvas.delete("all")  
 colors = ["#f0d9b5", "#b58863"]  
 for row in range(8):  
 for col in range(8):  
 color = colors[(row + col) % 2]  
 canvas.create\_rectangle(col \* SQUARE\_SIZE, row \* SQUARE\_SIZE,  
 (col + 1) \* SQUARE\_SIZE, (row + 1) \* SQUARE\_SIZE,  
 fill=color, outline=color)  
 # Get the piece at the current square  
 square = chess.square(col, 7 - row) # Convert row,col to square index  
 piece = board.piece\_at(square) # Make sure `board` is the `chess.Board` object  
 if piece:  
 piece\_image = load\_piece\_image(piece)  
 if piece\_image:  
 canvas.create\_image(col \* SQUARE\_SIZE + SQUARE\_SIZE / 2,  
 row \* SQUARE\_SIZE + SQUARE\_SIZE / 2,  
 image=piece\_image)  
  
 # Draw the file (a-h) and rank (1-8) labels  
 draw\_labels()  
  
 # Draw captured pieces (right panel)  
 draw\_captured\_pieces()  
  
# Draw labels for file (a-h) and rank (1-8)  
def draw\_labels():  
 # Rank labels (1-8) on the right side of the board (vertical)  
 for i in range(8):  
 canvas.create\_text(8 \* SQUARE\_SIZE + 20, (i + 0.5) \* SQUARE\_SIZE, text=str(8 - i), font=("Arial", 12), anchor="center")  
  
 # File labels (a-h) below the board (horizontal)  
 for i in range(8):  
 canvas.create\_text((i + 0.5) \* SQUARE\_SIZE, 8 \* SQUARE\_SIZE + 20, text=chr(97 + i), font=("Arial", 12), anchor="center")  
  
# Draw captured pieces in two columns  
def draw\_captured\_pieces():  
 # Create frames for captured pieces if not already created  
 white\_frame = tk.Frame(right\_frame)  
 white\_frame.grid(row=0, column=0, padx=10, pady=10)  
 black\_frame = tk.Frame(right\_frame)  
 black\_frame.grid(row=0, column=1, padx=10, pady=10)  
  
 # Captured white pieces (on the white side)  
 for i, piece in enumerate(captured\_white):  
 piece\_image = load\_piece\_image(piece)  
 if piece\_image:  
 tk.Label(white\_frame, image=piece\_image).grid(row=i, column=0, pady=5) # Display captured white pieces in white\_frame  
  
 # Captured black pieces (on the black side)  
 for i, piece in enumerate(captured\_black):  
 piece\_image = load\_piece\_image(piece)  
 if piece\_image:  
 tk.Label(black\_frame, image=piece\_image).grid(row=i, column=0, pady=5) # Display captured black pieces in black\_frame  
  
# Function to handle clicks on the board  
selected\_square = None  
  
def on\_square\_click(event):  
 global selected\_square  
 col = event.x // SQUARE\_SIZE  
 row = event.y // SQUARE\_SIZE  
 square = chess.square(col, 7 - row)  
  
 # Ensure the player is trying to move their own piece  
 if selected\_square is None:  
 piece = board.piece\_at(square)  
 if piece and ((board.turn == chess.WHITE and piece.color == chess.WHITE) or (board.turn == chess.BLACK and piece.color == chess.BLACK)):  
 selected\_square = square  
 else:  
 move = chess.Move(selected\_square, square)  
 if move in board.legal\_moves:  
 piece = board.piece\_at(selected\_square)  
 if piece: # If a piece is moved, track captured pieces  
 captured\_piece = board.piece\_at(square)  
 if captured\_piece:  
 if captured\_piece.color == chess.WHITE:  
 captured\_white.append(captured\_piece)  
 else:  
 captured\_black.append(captured\_piece)  
  
 move\_history.append(move) # Save the move to history for undo  
 board.push(move)  
 draw\_board() # Redraw the board after the move  
 update\_status() # Update turn after the move  
  
 selected\_square = None  
  
# Set up the event handler for clicks  
canvas.bind("<Button-1>", on\_square\_click)  
  
# Function to update the game status (turn info)  
def update\_status():  
 if board.is\_checkmate():  
 status\_label.config(text="Checkmate! " + ("White" if board.turn == chess.BLACK else "Black") + " wins!")  
 restart\_button.pack() # Show restart button on checkmate  
 elif board.is\_stalemate():  
 status\_label.config(text="Stalemate!")  
 restart\_button.pack() # Show restart button on stalemate  
 elif board.is\_check():  
 status\_label.config(text="Check! " + ("White" if board.turn == chess.WHITE else "Black") + " in check!")  
 elif board.is\_insufficient\_material():  
 status\_label.config(text="Draw! Insufficient material.")  
 restart\_button.pack() # Show restart button on draw  
 elif board.is\_fifty\_moves():  
 status\_label.config(text="Draw! Fifty-move rule.")  
 restart\_button.pack() # Show restart button on draw  
 elif board.is\_variant\_draw():  
 status\_label.config(text="Draw! Variant draw.")  
 restart\_button.pack() # Show restart button on variant draw  
 else:  
 status\_label.config(text="White's Turn" if board.turn == chess.WHITE else "Black's Turn")  
  
# Create an undo button  
def undo\_move():  
 if move\_history:  
 last\_move = move\_history.pop() # Get the last move from the history  
 board.pop() # Undo the move on the board  
 draw\_board() # Redraw the board after the undo  
 update\_status() # Update the status after the undo  
  
# Create a restart game button  
def restart\_game():  
 global captured\_white, captured\_black, move\_history, board  
 captured\_white = []  
 captured\_black = []  
 move\_history = []  
 board = chess.Board() # Reset the board to its initial state  
 draw\_board() # Redraw the board  
 update\_status() # Update the status  
 restart\_button.pack\_forget() # Hide the restart button after the game restarts  
  
# Create a status label at the bottom  
status\_label = tk.Label(root, text="White's Turn", bg='green', fg='black', font=("Arial", 14))  
status\_label.pack()  
  
# Create an undo button below the status label  
undo\_button = tk.Button(root, text="Undo Move", bg='blue', fg='pink', font=("Arial", 12), command=undo\_move)  
undo\_button.pack()  
  
# Create a restart button (initially hidden)  
restart\_button = tk.Button(root, text="Restart Game", bg='yellow', fg='black', font=("Arial", 12), command=restart\_game)  
restart\_button.pack\_forget() # Hide the button initially  
  
# Initialize the board and start the GUI loop  
draw\_board()  
root.mainloop()