פרויקט גמר - אלגוריתמים מתקדמים למערכות נבונות
Enhanced Checkers

שם המרצה: ד"ר רינה צביאל גירשין
שמות המגישים: עלאא יחיא
מוחמד אבו פול
מוחמד לחאם

לינקים:

מצגת

סרט הדגמה של המצגת – Google Photos

סרט הדגמה של המצגת – Youtube

סרט הדגמה של המשחק – Google Photos

סרט הדגמה של המשחק – Youtube

GitHub

סיכום כללי

הרקע והבעיה

בעיית הבסיס היא פיתוח משחק דמקה אינטראקטיבי שיירחיב את חוקי הדמקה הקלאסית, יאפשר גודל לוחות משתנים, מערכת כוחות מיוחדים חד-פעמיים ויתמוך במשחק מול בינה מלאכותית חכמה.

הפתרון שבנינו

- ממשק גרפי: באמצעות Pygame, עם בחירת לוח, אנימציות תנועה, הדגשת מהלכים וחלון מידע.
- לוח דינמי.12×12 / 10×10 / 8×8 :
- מצבי משחק: שחקן-נגד-שחקן (PvP) ושחקן-נגד-AI כלומר (PVAI).
- כוחות מיוחדים (פעם אחת לכל שחקן):
  - **DOUBLE_JUMP** – קפיצה של שתי משבצות
  - **IMMUNITY** – חסינות לתור היריב הבא
  - **MOVE_TWICE** – שני מהלכים רצופים
  - **FORCE_SWAP** – החלפת בעלות על כלי יריב
- בינה מלאכותית: מימוש שני אלגוריתמי חיפוש – Minimax רגיל ו-Alpha-Beta Pruning, עם גמישות בעומק החיפוש (1–8).

טכנולוגיות ושפות

- שפת תכנות Python :
- ספרייה גרפית Pygame :
- יתרונות: פשטות, מהירות פיתוח, ניידות בין פלטפורמות.

 # הסבר על המשחק עצמו נמצא במצגת, סרט הדגמה למצגת וסרט ההדגמה למשחק.

אלגוריתמים וניהול AI

Minimax רגיל

פונקציה:

```
def minimax_plain(board, depth, maximizing_player, current_player):
    if depth == 0 or board.is_terminal(): …
    moves = board.get_all_moves(current_player)
    …
    # maximize or minimize
```

- מסע עץ מלא לעומק depth; סיבוכיות $O(b^d)$.

Alpha-Beta Pruning

פונקציה:

```
def minimax_ab(board, depth, alpha, beta, maximizing_player,
current_player):
    if depth == 0 or board.is_terminal(): …
    for mv in moves:
        …
        alpha = max(alpha, eval_score)
        if beta <= alpha: break
```

- חיתוך ענפים חסרי תועלת (prune) בעזרת α ו-β.

הסיבוכיות

הפרויקט משתמש באלגוריתמים **Minimax** ו־ **Alpha-Beta Pruning** לקבלת החלטות עבור שחקן ה־AI.
הסיבוכיות של האלגוריתמים האלו נבחנת לפי שני פרמטרים:

- **b** - מספר האפשרויות הממוצע לתור אחד (branching factor)
- **d** - עומק החיפוש של האלגוריתם

Minimax – 🔹 סיבוכיות תאורטית: O (b^d)
כלומר, ככל שמתרחבים מספר האפשרויות בתור והעומק בו מחפש ה־ AI, כמות המצבים הנבדקים גדלה בצורה אקספוננציאלית.

Alpha-Beta Pruning – 🔹 סיבוכיות במקרה הטוב: O(b^{d/2})
כאשר מבוצע חיתוך ענפים אפקטיבי. כלומר, האלגוריתם מדלג על מצבים מיותרים שברור שלא ישפרו את התוצאה.

הפונקציה ההיריסטית (Evaluation)

- **מרכיבים:**

1. **חומר:** Man=1, King=2, Super-King=3 →
   `Σ(שחקן) − Σ(יריב) = score`

2. **משקל כוחות פעילים:**
   - DOUBLE_JUMP → בונוס לטווח תנועה
   - IMMUNITY → בונוס לחסינות מדירוג אכילה
   - MOVE_TWICE → תור נוסף ↔ בונוס אסטרטגי
   - FORCE_SWAP → ערך לפי חשיבות הכלי המוחלף

3. **אכילה לאחור:** יתרון לחייל רגיל → מגדילה את מגוון המהלכים החוקיים

4. **התאמות לגודל לוח** (לוחות גדולים → משקל גבוה יותר על יתרון חומרי).

5. **מלך על (Super King)**
   - תנועה "מעופפת" באלכסונים, אכילה מרחוק
   - בפונקציה: בונוס של 3 נקודות לחומר + תוספת משקל אסטרטגי (Mobility)

- **התוצאה:** הערכה מורכבת יותר שמשלבת חומר, כוחות, ויכולות תנועה.

מתי וְכיצד AI משתמש בכוח

פונקציה:

```
def should_ai_use_power(board, ai_player):
    if board.power_used[ai_player]: return False
    ai_pieces = …
    opp_pieces = …
    if ai_pieces < opp_pieces or (ai_pieces+opp_pieces)<8:
        return True
    return random.random() < 0.3
```

- שילוב קריטריונים (#pieces, שלב המשחק) ו־30% סיכוי אקראי.

קטעי קוד עיקריים

מימוש לוח ודפוסי תנועה

```
class Board:
    def __init__(self, size):
        self.size = size
        self.matrix = [[0]*size for _ in range(size)]
        self._init_pieces()
        self.power_used = {'R': False, 'G': False}
        self.active_powers = {'R': None, 'G': None}
        self.immunity_turns = {'R':0,'G':0}
        self.double_jump_active = {'R':False,'G':False}
        self.move_twice_active = {'R':False,'G':False}
    def _init_pieces(self):
        rows_each = self.size//2-1
        for r in range(self.size):
            for c in range(self.size):
                if (r+c)%2==0:
                    if r<rows_each: self.matrix[r][c]=1
                    elif r>=self.size-rows_each: self.matrix[r][c]=-1
```

חיפוש מהלכים ואכילה

```
    def _get_piece_moves(self, start_pos):
        # handle normal, king, super-king captures
        # check immunity: if self.immunity_turns[player]>0: continue
        # double_jump_active extends jump_range
```

הפעלה וזרימת כוח

```
   def generate_random_power(self, player):
        if not self.power_used[player]:
            p = random.choice(list(PowerType))
            self.active_powers[player] = p
            self.power_used[player] = True
            return p
```

```
        return None

    def activate_power(self, player, power_type, target=None):
        if power_type == PowerType.DOUBLE_JUMP:
            self.double_jump_active[player] = True
        elif power_type == PowerType.IMMUNITY:
            self.immunity_turns[player] = 1
        # etc.
```

Alpha-Beta-ו Minimax

```
def minimax_plain(board, depth, max_p, curr_p): …
def minimax_ab(board, depth, alpha, beta, max_p, curr_p): …
def find_best_move(board, ai_p, depth, alg):
    if alg=="Minimax": return minimax_plain(...)
    return minimax_ab(...)
```

שילוב AI בלולאת המשחק

```
if mode=='AI' and curr_player=='R':
    if should_ai_use_power(board,'R'):
        p = board.generate_random_power('R')
        board.activate_power('R', p)
    ai_move = find_best_move(board,'R',ai_depth,ai_algorithm)
    animate_move(...)
    board.update_turn_effects('R')
    curr_player='G'
```

אנימציית מהלך ואכילה מרובת קפיצות

```
def animate_move(...):
    for each segment:
steps-הפקת אנימציה חלקה ב #
        if capture: remove piece immediately; pause 0.5s
capture_count בסיום: קידום לפי #
```

קוד מלא

מצורף בקובץ מקורי (checkers.py)

```
import pygame
import sys
import math
import random
from enum import Enum

# Power-up types
class PowerType(Enum):
    DOUBLE_JUMP = "🔥 Double Jump"
    IMMUNITY = "🛡️ Immunity"
    MOVE_TWICE = "⏱️ Move Twice"
    FORCE_SWAP = "🧲 Force Swap"

# Descriptions for each power, shown in the pop-out alert
POWER_DESCRIPTIONS = {
    PowerType.DOUBLE_JUMP: "Allows jumping over two spaces in a single move segment.",
    PowerType.IMMUNITY: "Your pieces cannot be captured during your opponent's next turn.",
    PowerType.MOVE_TWICE: "Take two separate moves in this turn.",
    PowerType.FORCE_SWAP: "Swap ownership of an opponent's piece at a chosen position.",
}

# ----------------------------
# Configuration
# ----------------------------
TIME_PER_TURN = 40     # seconds per turn

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
BLUE = (76, 252, 241)
RED_COLOR = (200, 50, 50)
GREEN_COLOR = (50, 200, 50)
PANEL_BG = (40, 40, 40)
OVERLAY_COLOR = (0, 0, 0, 180)
POWER_BUTTON_COLOR = (100, 100, 255)
POWER_BUTTON_HOVER = (150, 150, 255)
POWER_BUTTON_USED = (80, 80, 80)
IMMUNITY_GLOW = (255, 215, 0)  # Golden glow for immunity
BUTTON_TEXT_COLOR = WHITE

# Initialize Pygame
pygame.init()

# Query display size to choose board size dynamically
display_info = pygame.display.Info()
SCREEN_W, SCREEN_H = display_info.current_w, display_info.current_h
```

```
# Fonts
FONT = pygame.font.SysFont('Arial', 24)
BIG_FONT = pygame.font.SysFont('Arial', 32)
CLOCK = pygame.time.Clock()
FPS = 60

# Globals to set after board size selection
ROWS = None
SQUARE_SIZE = None
BOARD_PIXELS = None
# Increase panel height to ensure enough space
PANEL_HEIGHT = 160  # px for info & buttons panel

# Images to load after SQUARE_SIZE known
RED_IMG = None
GREEN_IMG = None
RED_KING_IMG = None
GREEN_KING_IMG = None
STAR_IMG = None

# Pygame window placeholder; will set later
WIN = None

# ------------------------------
# Board & Game Logic
# ------------------------------
class Board:
    """
    Dynamic-size board representation.
    """
    def __init__(self, size):
        self.size = size
        self.matrix = [[0 for _ in range(size)] for _ in range(size)]
        self._init_pieces()

        # Power-up tracking
        self.power_used = {'R': False, 'G': False}
        self.active_powers = {'R': None, 'G': None}
        # immunity_turns[player] > 0 means this player's pieces are immune for the upcoming opponent
turn
        self.immunity_turns = {'R': 0, 'G': 0}
        self.double_jump_active = {'R': False, 'G': False}
        self.move_twice_active = {'R': False, 'G': False}

    def _init_pieces(self):
        rows_each = self.size // 2 - 1
        for row in range(self.size):
            for col in range(self.size):
                if (row + col) % 2 == 0:
                    if row < rows_each:
                        self.matrix[row][col] = 1   # Red man
                    elif row >= self.size - rows_each:
```

```
                self.matrix[row][col] = -1  # Green man

    def clone(self):
        b = Board(self.size)
        b.matrix = [r.copy() for r in self.matrix]
        b.power_used = self.power_used.copy()
        b.active_powers = self.active_powers.copy()
        b.immunity_turns = self.immunity_turns.copy()
        b.double_jump_active = self.double_jump_active.copy()
        b.move_twice_active = self.move_twice_active.copy()
        return b

    def in_bounds(self, row, col):
        return 0 <= row < self.size and 0 <= col < self.size

    def get_all_moves(self, player):
        moves = []
        for r in range(self.size):
            for c in range(self.size):
                if player == 'R' and self.matrix[r][c] > 0:
                    seqs = self._get_piece_moves((r, c))
                    moves.extend(seqs)
                if player == 'G' and self.matrix[r][c] < 0:
                    seqs = self._get_piece_moves((r, c))
                    moves.extend(seqs)
        return moves

    def _get_piece_moves(self, start_pos):
        r0, c0 = start_pos
        piece = self.matrix[r0][c0]
        if piece == 0:
            return []
        player = 'R' if piece > 0 else 'G'
        absval = abs(piece)

        capture_seqs = []

        # DFS for normal captures
        def dfs_normal_captures(r, c, board_mat, path, visited):
            found = False
            for dr, dc in [(-1, -1), (-1, 1), (1, -1), (1, 1)]:
                mid_r, mid_c = r + dr, c + dc
                end_r, end_c = r + 2*dr, c + 2*dc
                if self.in_bounds(mid_r, mid_c) and self.in_bounds(end_r, end_c):
                    mid_val = board_mat[mid_r][mid_c]
                    end_val = board_mat[end_r][end_c]
                    if mid_val != 0 and (mid_val * piece) < 0 and end_val == 0:
                        # Check immunity of the target piece's owner
                        target_player = 'R' if mid_val > 0 else 'G'
                        if self.immunity_turns[target_player] > 0:
                            continue
                        if (mid_r, mid_c) in visited:
```

```
                continue
            new_board = [row.copy() for row in board_mat]
            new_board[r][c] = 0
            new_board[mid_r][mid_c] = 0
            new_board[end_r][end_c] = piece
            new_path = path + [(end_r, end_c)]
            new_visited = visited | {(mid_r, mid_c)}
            deeper = dfs_normal_captures(end_r, end_c, new_board, new_path, new_visited)
            if not deeper:
                capture_seqs.append(new_path)
            found = True
    return found

# DFS for super king captures (flying captures)
def dfs_super_captures(r, c, board_mat, path, visited):
    found = False
    for dr, dc in [(-1, -1), (-1, 1), (1, -1), (1, 1)]:
        i = 1
        while True:
            mid_r = r + dr*i
            mid_c = c + dc*i
            if not self.in_bounds(mid_r, mid_c):
                break
            mid_val = board_mat[mid_r][mid_c]
            if mid_val == 0:
                i += 1
                continue
            if mid_val * piece > 0:
                break
            if mid_val * piece < 0 and (mid_r, mid_c) not in visited:
                # Check immunity
                target_player = 'R' if mid_val > 0 else 'G'
                if self.immunity_turns[target_player] > 0:
                    break
                j = 1
                while True:
                    land_r = mid_r + dr*j
                    land_c = mid_c + dc*j
                    if not self.in_bounds(land_r, land_c):
                        break
                    if board_mat[land_r][land_c] != 0:
                        break
                    new_board = [row.copy() for row in board_mat]
                    new_board[r][c] = 0
                    new_board[mid_r][mid_c] = 0
                    new_board[land_r][land_c] = piece
                    new_path = path + [(land_r, land_c)]
                    new_visited = visited | {(mid_r, mid_c)}
                    deeper = dfs_super_captures(land_r, land_c, new_board, new_path, new_visited)
                    if not deeper:
                        capture_seqs.append(new_path)
                    found = True
```

```
                j += 1
            break
        else:
            break
    # next direction
return found

# First, capture search
if absval == 3:
    dfs_super_captures(r0, c0, self.matrix, [(r0, c0)], set())
else:
    dfs_normal_captures(r0, c0, self.matrix, [(r0, c0)], set())

# Simple moves:
moves = []
jump_range = 2 if self.double_jump_active[player] else 1

if absval == 3:
    for dr, dc in [(-1, -1), (-1, 1), (1, -1), (1, 1)]:
        i = 1
        while True:
            nr = r0 + dr*i
            nc = c0 + dc*i
            if not self.in_bounds(nr, nc):
                break
            if self.matrix[nr][nc] == 0:
                moves.append([(r0, c0), (nr, nc)])
                i += 1
            else:
                break
else:
    if absval == 2:
        dirs = [(-1, -1), (-1, 1), (1, -1), (1, 1)]
    else:
        if player == 'R':
            dirs = [(1, -1), (1, 1)]
        else:
            dirs = [(-1, -1), (-1, 1)]
    for dr, dc in dirs:
        for distance in range(1, jump_range + 1):
            nr = r0 + dr * distance
            nc = c0 + dc * distance
            if self.in_bounds(nr, nc) and self.matrix[nr][nc] == 0:
                if distance == 2:
                    mid_r = r0 + dr
                    mid_c = c0 + dc
                    if self.matrix[mid_r][mid_c] != 0:
                        break
                moves.append([(r0, c0), (nr, nc)])
            else:
                break
```

```
    # Combine capture_seqs and simple moves
    all_seqs = []
    for seq in capture_seqs:
        if seq not in all_seqs:
            all_seqs.append(seq)
    for seq in moves:
        if seq not in all_seqs:
            all_seqs.append(seq)
    return all_seqs

def apply_move_simple(self, move_seq):
    if not move_seq or len(move_seq) < 2:
        return
    sr, sc = move_seq[0]
    piece = self.matrix[sr][sc]
    initial_piece = piece
    self.matrix[sr][sc] = 0
    curr_r, curr_c = sr, sc
    capture_count = 0

    for idx in range(1, len(move_seq)):
        nr, nc = move_seq[idx]
        dr = nr - curr_r
        dc = nc - curr_c
        # super king
        if abs(initial_piece) == 3:
            if abs(dr) > 1 or abs(dc) > 1:
                step_r = 1 if dr > 0 else -1
                step_c = 1 if dc > 0 else -1
                r_iter, c_iter = curr_r + step_r, curr_c + step_c
                while (r_iter, c_iter) != (nr, nc):
                    val = self.matrix[r_iter][c_iter]
                    if val != 0:
                        if val * initial_piece < 0:
                            self.matrix[r_iter][c_iter] = 0
                            capture_count += 1
                            break
                        else:
                            break
                    r_iter += step_r
                    c_iter += step_c
                curr_r, curr_c = nr, nc
            else:
                curr_r, curr_c = nr, nc
        else:
            if abs(dr) == 2 and abs(dc) == 2:
                mid_r = (curr_r + nr) // 2
                mid_c = (curr_c + nc) // 2
                self.matrix[mid_r][mid_c] = 0
                capture_count += 1
            curr_r, curr_c = nr, nc
```

```
        # Promotion / super king
        if abs(initial_piece) == 3:
            new_piece = initial_piece
        else:
            if capture_count >= 3:
                new_piece = 3 if initial_piece > 0 else -3
            else:
                if initial_piece == 1 and curr_r == self.size - 1:
                    new_piece = 2
                elif initial_piece == -1 and curr_r == 0:
                    new_piece = -2
                else:
                    new_piece = initial_piece
        self.matrix[curr_r][curr_c] = new_piece

    def is_terminal(self):
        green_exists = any(self.matrix[r][c] < 0 for r in range(self.size) for c in range(self.size))
        red_exists = any(self.matrix[r][c] > 0 for r in range(self.size) for c in range(self.size))
        if not green_exists or not red_exists:
            return True
        green_moves = self.get_all_moves('G')
        red_moves = self.get_all_moves('R')
        if not green_moves or not red_moves:
            return True
        return False

    def get_winner(self):
        green_exists = any(self.matrix[r][c] < 0 for r in range(self.size) for c in range(self.size))
        red_exists = any(self.matrix[r][c] > 0 for r in range(self.size) for c in range(self.size))
        green_moves = self.get_all_moves('G')
        red_moves = self.get_all_moves('R')

        if not green_exists and red_exists:
            return 'R'
        if not red_exists and green_exists:
            return 'G'
        if (not green_moves) and (not red_moves):
            return 'D'
        if not green_moves:
            return 'R'
        if not red_moves:
            return 'G'
        return None

    def evaluate(self, player):
        total = 0
        for r in range(self.size):
            for c in range(self.size):
                val = self.matrix[r][c]
                if val < 0:
                    total += (1 if val == -1 else 2 if val == -2 else 3)
                elif val > 0:
```

```
                total -= (1 if val == 1 else 2 if val == 2 else 3)
    if player == 'G':
        return total
    else:
        return -total

def generate_random_power(self, player):
    if not self.power_used[player]:
        powers = list(PowerType)
        selected_power = random.choice(powers)
        self.active_powers[player] = selected_power
        self.power_used[player] = True
        return selected_power
    return None

def activate_power(self, player, power_type, target_data=None):
    if power_type == PowerType.DOUBLE_JUMP:
        self.double_jump_active[player] = True
        return True

    elif power_type == PowerType.IMMUNITY:
        # Protect this player's pieces during the opponent's upcoming turn:
        self.immunity_turns[player] = 1
        return True

    elif power_type == PowerType.MOVE_TWICE:
        self.move_twice_active[player] = True
        return True

    elif power_type == PowerType.FORCE_SWAP and target_data:
        row, col = target_data
        piece = self.matrix[row][col]
        if piece != 0:
            # Check if it's opponent's piece
            if (player == 'R' and piece < 0) or (player == 'G' and piece > 0):
                # Swap the piece's ownership:
                new_piece = abs(piece) if player == 'R' else -abs(piece)
                # Check promotion if swapping lands in back row
                if player == 'R' and row == self.size - 1 and abs(new_piece) == 1:
                    new_piece = 2
                elif player == 'G' and row == 0 and abs(new_piece) == 1:
                    new_piece = -2
                self.matrix[row][col] = new_piece
                return True
    return False

def update_turn_effects(self, player):
    """
    Called at end of player 'player"s turn (before switching to opponent).
    Clear per-turn flags for this player, and decrement immunity for the opponent.
    """
    # Clear double-jump
```

```
        self.double_jump_active[player] = False
        # Decrement immunity for the opponent (whose pieces were protected during this player's turn)
        opponent = 'R' if player == 'G' else 'G'
        if self.immunity_turns[opponent] > 0:
            self.immunity_turns[opponent] -= 1
        # Clear single-use active power
        if self.active_powers[player] in [PowerType.DOUBLE_JUMP, PowerType.IMMUNITY,
PowerType.FORCE_SWAP]:
            self.active_powers[player] = None

# ------------------------------
# Minimax Implementations
# ------------------------------
def minimax_plain(board, depth, maximizing_player, current_player):
    if depth == 0 or board.is_terminal():
        return board.evaluate(maximizing_player), None

    moves = board.get_all_moves(current_player)
    if not moves:
        return board.evaluate(maximizing_player), None

    next_player = 'R' if current_player == 'G' else 'G'

    if current_player == maximizing_player:
        max_eval = -math.inf
        best_move = None
        for mv in moves:
            new_b = board.clone()
            new_b.apply_move_simple(mv)
            eval_score, _ = minimax_plain(new_b, depth - 1, maximizing_player, next_player)
            if eval_score > max_eval:
                max_eval = eval_score
                best_move = mv
        return max_eval, best_move
    else:
        min_eval = math.inf
        best_move = None
        for mv in moves:
            new_b = board.clone()
            new_b.apply_move_simple(mv)
            eval_score, _ = minimax_plain(new_b, depth - 1, maximizing_player, next_player)
            if eval_score < min_eval:
                min_eval = eval_score
                best_move = mv
        return min_eval, best_move

def minimax_ab(board, depth, alpha, beta, maximizing_player, current_player):
    if depth == 0 or board.is_terminal():
        return board.evaluate(maximizing_player), None

    moves = board.get_all_moves(current_player)
    if not moves:
```

```
        return board.evaluate(maximizing_player), None

    next_player = 'R' if current_player == 'G' else 'G'

    if current_player == maximizing_player:
        max_eval = -math.inf
        best_move = None
        for mv in moves:
            new_b = board.clone()
            new_b.apply_move_simple(mv)
            eval_score, _ = minimax_ab(new_b, depth - 1, alpha, beta, maximizing_player, next_player)
            if eval_score > max_eval:
                max_eval = eval_score
                best_move = mv
            alpha = max(alpha, eval_score)
            if beta <= alpha:
                break
        return max_eval, best_move
    else:
        min_eval = math.inf
        best_move = None
        for mv in moves:
            new_b = board.clone()
            new_b.apply_move_simple(mv)
            eval_score, _ = minimax_ab(new_b, depth - 1, alpha, beta, maximizing_player, next_player)
            if eval_score < min_eval:
                min_eval = eval_score
                best_move = mv
            beta = min(beta, eval_score)
            if beta <= alpha:
                break
        return min_eval, best_move

def should_ai_use_power(board, ai_player):
    if board.power_used[ai_player]:
        return False

    ai_pieces = sum(1 for r in range(board.size) for c in range(board.size)
                    if board.matrix[r][c] * (1 if ai_player == 'R' else -1) > 0)
    opponent_pieces = sum(1 for r in range(board.size) for c in range(board.size)
                          if board.matrix[r][c] * (-1 if ai_player == 'R' else 1) > 0)

    if ai_pieces < opponent_pieces or (ai_pieces + opponent_pieces) < 8:
        return True
    return random.random() < 0.3

def find_best_move(board, ai_player, depth, algorithm_name):
    if algorithm_name == "Minimax":
        _, mv = minimax_plain(board, depth, ai_player, ai_player)
        return mv
    else:
        _, mv = minimax_ab(board, depth, -math.inf, math.inf, ai_player, ai_player)
```

```
        return mv

# ----------------------------
# Drawing Functions
# ----------------------------
def draw_board(win, board: Board, highlighted_positions):
    for row in range(board.size):
        for col in range(board.size):
            rect = pygame.Rect(col * SQUARE_SIZE, row * SQUARE_SIZE, SQUARE_SIZE,
SQUARE_SIZE)
            color = BLACK if (row + col) % 2 == 0 else WHITE
            pygame.draw.rect(win, color, rect)

            if (row, col) in highlighted_positions:
                pygame.draw.rect(win, BLUE, rect)

            val = board.matrix[row][col]
            if val != 0:
                # Determine base image
                if val == -1:
                    img = GREEN_IMG
                elif val == -2:
                    img = GREEN_KING_IMG
                elif val == 1:
                    img = RED_IMG
                elif val == 2:
                    img = RED_KING_IMG
                else:
                    if val == -3:
                        img = GREEN_KING_IMG
                    elif val == 3:
                        img = RED_KING_IMG
                    else:
                        img = None
                if img:
                    img_rect = img.get_rect(center=rect.center)
                    win.blit(img, img_rect)
                # If super king, overlay star
                if abs(val) == 3 and STAR_IMG:
                    star_rect = STAR_IMG.get_rect(center=(rect.centerx, rect.centery - SQUARE_SIZE//4))
                    win.blit(STAR_IMG, star_rect)

    # Grid lines
    for i in range(board.size + 1):
        pygame.draw.line(win, BLACK, (0, i * SQUARE_SIZE), (BOARD_PIXELS, i * SQUARE_SIZE))
        pygame.draw.line(win, BLACK, (i * SQUARE_SIZE, 0), (i * SQUARE_SIZE, BOARD_PIXELS))

    # Immunity glow: outline any piece whose owner has immunity_turns > 0
    for row in range(board.size):
        for col in range(board.size):
            val = board.matrix[row][col]
            if val != 0:
```

```
            player = 'R' if val > 0 else 'G'
            if board.immunity_turns[player] > 0:
                rect = pygame.Rect(col * SQUARE_SIZE, row * SQUARE_SIZE, SQUARE_SIZE,
SQUARE_SIZE)
                pygame.draw.rect(win, IMMUNITY_GLOW, rect, 4)

def draw_info_panel(win, board: Board, mode, algo_name, depth, curr_player, red_time_left,
green_time_left, score_green, score_red):
    panel_rect = pygame.Rect(0, BOARD_PIXELS, BOARD_PIXELS, PANEL_HEIGHT)
    pygame.draw.rect(win, PANEL_BG, panel_rect)
    pygame.draw.line(win, WHITE, (0, BOARD_PIXELS), (BOARD_PIXELS, BOARD_PIXELS), 2)

    x_left = 10
    y_top = BOARD_PIXELS + 10

    # -------- Mode / AI Info --------
    surf = FONT.render(f"Mode: {'1v1' if mode == '1v1' else 'Vs AI'}", True, WHITE)
    win.blit(surf, (x_left, y_top))
    if mode == 'AI':
        surf2 = FONT.render(f"AI: {algo_name}, Depth={depth}", True, WHITE)
        win.blit(surf2, (x_left, y_top + 26))

    # -------- Turn & Timers --------
    center_x = BOARD_PIXELS // 3 - 90
    timer_top_offset = 40 if mode == 'AI' else 0
    turn_text = FONT.render(f"Turn: {'Green' if curr_player == 'G' else 'Red'}", True, WHITE)
    win.blit(turn_text, (center_x, y_top))

    def fmt(t):
        if t is None:
            return "--:--"
        if t < 0:
            t = 0
        return f"{int(t)//60:02d}:{int(t)%60:02d}"

    green_t = fmt(green_time_left)
    red_t = fmt(red_time_left) if mode == '1v1' else "--:--"

    red_surf = FONT.render(f"Red Time: {red_t}", True, RED_COLOR if mode == '1v1' else
(120,120,120))
    green_surf = FONT.render(f"Green Time: {green_t}", True, GREEN_COLOR)
    win.blit(red_surf, (center_x, y_top + 28 + timer_top_offset))
    win.blit(green_surf, (center_x, y_top + 56 + timer_top_offset))

    # -------- Score & Buttons --------
    right_x = BOARD_PIXELS * 2 // 3 - 110

    font_btn = pygame.font.SysFont('Arial', 18)
    btn_w, btn_h = 160, 30
    gap_y = 4
    button_rects = {}
```

```
    # -------- Green Button --------
    btn_x = right_x
    btn_y = y_top + 10
    green_btn_rect = pygame.Rect(btn_x, btn_y, btn_w, btn_h)
    mouse_pos = pygame.mouse.get_pos()
    green_used = board.power_used['G']
    green_active = board.active_powers['G'].value if board.active_powers['G'] else None

    if green_used:
        text = green_active or "Used"
        color = POWER_BUTTON_USED
    else:
        color = POWER_BUTTON_HOVER if green_btn_rect.collidepoint(mouse_pos) else
POWER_BUTTON_COLOR
        text = "Generate"
    pygame.draw.rect(win, color, green_btn_rect)
    pygame.draw.rect(win, WHITE, green_btn_rect, 2)
    txt_surf = font_btn.render(text, True, BUTTON_TEXT_COLOR)
    win.blit(txt_surf, txt_surf.get_rect(center=green_btn_rect.center))

    # Label: if used, show power name instead of "Green Generate Power"
    green_label = green_active if green_used else "Green Generate Power"
    label_color = GREEN_COLOR
    label_surf = font_btn.render(green_label, True, label_color)
    win.blit(label_surf, (btn_x + btn_w + 8, btn_y + (btn_h - label_surf.get_height())//2))
    button_rects['G'] = green_btn_rect if not green_used else None

    # -------- Red/AI Button --------
    btn_y2 = btn_y + btn_h + 10
    red_btn_rect = pygame.Rect(btn_x, btn_y2, btn_w, btn_h)
    red_used = board.power_used['R']
    red_active = board.active_powers['R'].value if board.active_powers['R'] else None

    if mode == 'AI':
        # AI Mode: button is always disabled
        if red_used:
            text = red_active or "Used"
        else:
            text = "Generate"
        color = POWER_BUTTON_USED
    else:
        if red_used:
            text = red_active or "Used"
            color = POWER_BUTTON_USED
        else:
            color = POWER_BUTTON_HOVER if red_btn_rect.collidepoint(mouse_pos) else
POWER_BUTTON_COLOR
            text = "Generate"

    pygame.draw.rect(win, color, red_btn_rect)
    pygame.draw.rect(win, WHITE, red_btn_rect, 2)
    txt2 = font_btn.render(text, True, BUTTON_TEXT_COLOR)
```

```
    win.blit(txt2, txt2.get_rect(center=red_btn_rect.center))

    red_label = red_active if red_used else ("Red Generate Power" if mode == '1v1' else "AI(Red) Generate Power")
    label_color = RED_COLOR if mode == '1v1' else (120,120,120)
    label_surf2 = font_btn.render(red_label, True, label_color)
    win.blit(label_surf2, (btn_x + btn_w + 8, btn_y2 + (btn_h - label_surf2.get_height())//2))

    if mode == '1v1' and not red_used:
        button_rects['R'] = red_btn_rect
    else:
        button_rects['R'] = None

    return button_rects

def draw_end_message(win, message):
    # Dim background with overlay
    overlay = pygame.Surface((BOARD_PIXELS, BOARD_PIXELS), pygame.SRCALPHA)
    overlay.fill(OVERLAY_COLOR)
    win.blit(overlay, (0, 0))

    # Main message (e.g. "Green Wins!" or "Red Wins!" or "Draw")
    text_surf = BIG_FONT.render(message, True, WHITE)
    rect = text_surf.get_rect(center=(BOARD_PIXELS // 2, BOARD_PIXELS // 2 - 20))
    win.blit(text_surf, rect)
    # Only show "Press Q to quit"
    sub_surf = FONT.render("Press Q to quit", True, WHITE)
    sub_rect = sub_surf.get_rect(center=(BOARD_PIXELS // 2, BOARD_PIXELS // 2 + 20))
    win.blit(sub_surf, sub_rect)

def get_clicked_pos(pos):
    x, y = pos
    if y >= BOARD_PIXELS:
        return None
    col = x // SQUARE_SIZE
    row = y // SQUARE_SIZE
    return (row, col)

def draw_alert(win, alert_message, start_time, total_duration=5000, display_time=4000):
    """
    Draw a pop-out alert: fully visible for `display_time` ms, then fades out over (total_duration - display_time) ms.
    alert_message: string (may be multi-line separated by '\n').
    start_time: pygame.time.get_ticks() when alert started.
    total_duration: total ms before expiration (default 5000).
    display_time: ms to keep fully opaque (default 4000).
    """
    now = pygame.time.get_ticks()
    elapsed = now - start_time
    if elapsed >= total_duration:
        return False  # alert expired
```

```
    # Determine alpha: fully opaque until display_time, then fade out over the remaining time
    if elapsed < display_time:
        alpha = 255
    else:
        fade_elapsed = elapsed - display_time
        fade_duration = total_duration - display_time
        alpha = int(255 * (1 - fade_elapsed / fade_duration))
        if alpha < 0:
            alpha = 0

    # Render a semi-transparent rectangle background
    lines = alert_message.split('\n')
    font = pygame.font.SysFont('Arial', 24)
    text_surfs = [font.render(line, True, WHITE) for line in lines]
    width = max(surf.get_width() for surf in text_surfs) + 20
    height = sum(surf.get_height() for surf in text_surfs) + 20

    # Position: centered horizontally, y at e.g. 20 px below top of board
    x = (BOARD_PIXELS - width) // 2
    y = 20  # 20 px below top of board

    # Create surface with per-pixel alpha
    s = pygame.Surface((width, height), pygame.SRCALPHA)
    # Background semi-transparent black at half of text alpha
    bg_color = (0, 0, 0, alpha // 2)
    s.fill(bg_color)

    # Blit text with full alpha onto this surface, but set text alpha
    y_offset = 10
    for surf in text_surfs:
        text_surf = surf.copy()
        text_surf.set_alpha(alpha)
        s.blit(text_surf, (10, y_offset))
        y_offset += surf.get_height()

    # Blit s onto main window
    win.blit(s, (x, y))
    return True  # alert still active

# ------------------------------
# Animation for Moves
# ------------------------------
def animate_move(board, move_seq, piece_val, mode, algo_name, ai_depth, curr_player,
                 red_time_left, green_time_left, score_green, score_red, alert_message=None,
alert_start_time=None):
    """
    Animate a move. After each capture segment, pause for 0.5 seconds at the landing square.
    alert_message & alert_start_time: if an alert is active, continue drawing/fading it during animation &
pauses.
    """
    def piece_image(val):
        if val == -1:
```

```
            return GREEN_IMG
        elif val == -2:
            return GREEN_KING_IMG
        elif val == 1:
            return RED_IMG
        elif val == 2:
            return RED_KING_IMG
        elif val == -3:
            return GREEN_KING_IMG
        elif val == 3:
            return RED_KING_IMG
        else:
            return None

    initial_piece = piece_val
    # Pre-calc total capture count if needed for promotion later
    capture_count = 0
    for i in range(len(move_seq) - 1):
        r0, c0 = move_seq[i]
        r1, c1 = move_seq[i+1]
        dr = r1 - r0
        dc = c1 - c0
        if abs(initial_piece) == 3:
            # super king: any jump distance >1 is a capture
            if abs(dr) > 1 or abs(dc) > 1:
                capture_count += 1
        else:
            if abs(dr) == 2 and abs(dc) == 2:
                capture_count += 1

    # Loop through each segment
    for idx in range(len(move_seq) - 1):
        sr, sc = move_seq[idx]
        er, ec = move_seq[idx+1]
        img = piece_image(piece_val)
        start_px = (sc * SQUARE_SIZE + SQUARE_SIZE//2, sr * SQUARE_SIZE + SQUARE_SIZE//2)
        end_px   = (ec * SQUARE_SIZE + SQUARE_SIZE//2, er * SQUARE_SIZE + SQUARE_SIZE//2)
        # Remove piece from source
        board.matrix[sr][sc] = 0

        # Animate the move from start_px to end_px over 'steps' frames
        steps = 10
        for step in range(1, steps + 1):
            t = step / steps
            x = start_px[0] + (end_px[0] - start_px[0]) * t
            y = start_px[1] + (end_px[1] - start_px[1]) * t
            WIN.fill((0, 0, 0))
            draw_board(WIN, board, set())
            if img:
                WIN.blit(img, img.get_rect(center=(x, y)))
            if abs(piece_val) == 3 and STAR_IMG:
                star_pos = (x, y - SQUARE_SIZE//4)
```

```
            WIN.blit(STAR_IMG, STAR_IMG.get_rect(center=star_pos))
        # Draw info panel
        btns = draw_info_panel(WIN, board, mode, algo_name, ai_depth, curr_player,
                               red_time_left, green_time_left, score_green, score_red)
        # Draw alert if active
        if alert_message is not None and alert_start_time is not None:
            still = draw_alert(WIN, alert_message, alert_start_time)
            if not still:
                alert_message = None
                alert_start_time = None
        pygame.display.update()
        CLOCK.tick(FPS)

    # After animation: handle capture removal at landing square
    captured = False
    if abs(initial_piece) == 3:
        # super king: if distance > 1 in either direction, remove the captured piece on path
        dr = er - sr
        dc = ec - sc
        if abs(dr) > 1 or abs(dc) > 1:
            captured = True
            step_r = 1 if dr > 0 else -1
            step_c = 1 if dc > 0 else -1
            r_iter, c_iter = sr + step_r, sc + step_c
            while (r_iter, c_iter) != (er, ec):
                val = board.matrix[r_iter][c_iter]
                if val != 0:
                    if val * initial_piece < 0:
                        board.matrix[r_iter][c_iter] = 0
                    break
                r_iter += step_r
                c_iter += step_c
    else:
        # normal or king: capture if abs(dr)==2 and abs(dc)==2
        dr = er - sr
        dc = ec - sc
        if abs(dr) == 2 and abs(dc) == 2:
            captured = True
            mid_r = (sr + er) // 2
            mid_c = (sc + ec) // 2
            board.matrix[mid_r][mid_c] = 0

    # Place the piece at landing square
    board.matrix[er][ec] = piece_val

    # If this segment was a capture, pause for 0.5 seconds showing the current board
    if captured:
        pause_start = pygame.time.get_ticks()
        while pygame.time.get_ticks() - pause_start < 500:
            # Draw the board state with piece at (er,ec)
            WIN.fill((0, 0, 0))
            draw_board(WIN, board, set())
```

```
            # Optionally, highlight the capturing piece? Here we just draw normally.
            # Draw info panel
            btns = draw_info_panel(WIN, board, mode, algo_name, ai_depth, curr_player,
                                   red_time_left, green_time_left, score_green, score_red)
            # Draw alert if active
            if alert_message is not None and alert_start_time is not None:
                still = draw_alert(WIN, alert_message, alert_start_time)
                if not still:
                    alert_message = None
                    alert_start_time = None
            pygame.display.update()
            CLOCK.tick(FPS)
            # Process events lightly to keep window responsive
            for ev in pygame.event.get():
                if ev.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()
        # End of pause for this capture segment

    # After all segments: handle final promotion
    final_r, final_c = move_seq[-1]
    board.matrix[final_r][final_c] = 0
    if abs(initial_piece) == 3:
        new_piece = initial_piece
    else:
        if capture_count >= 3:
            new_piece = 3 if initial_piece > 0 else -3
        else:
            if initial_piece == 1 and final_r == board.size - 1:
                new_piece = 2
            elif initial_piece == -1 and final_r == 0:
                new_piece = -2
            else:
                new_piece = initial_piece
    board.matrix[final_r][final_c] = new_piece

# ------------------------------
# Main Game Flow
# ------------------------------
def main():
    global ROWS, SQUARE_SIZE, BOARD_PIXELS, WIN
    global RED_IMG, GREEN_IMG, RED_KING_IMG, GREEN_KING_IMG, STAR_IMG

    # 1. Board size selection
    ROWS = None
    temp_win = pygame.display.set_mode((600, 400))
    pygame.display.set_caption('Checkers - Select Board Size')
    while ROWS is None:
        temp_win.fill((30, 30, 30))
        title = BIG_FONT.render("Select Board Size", True, WHITE)
        temp_win.blit(title, title.get_rect(center=(300, 100)))
        o1 = FONT.render("Press 1: 8 x 8", True, WHITE)
```

```
        o2 = FONT.render("Press 2: 10 x 10", True, WHITE)
        o3 = FONT.render("Press 3: 12 x 12", True, WHITE)
        temp_win.blit(o1, (300 - o1.get_width()//2, 160))
        temp_win.blit(o2, (300 - o2.get_width()//2, 190))
        temp_win.blit(o3, (300 - o3.get_width()//2, 220))
        pygame.display.update()
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_1:
                    ROWS = 8
                elif event.key == pygame.K_2:
                    ROWS = 10
                elif event.key == pygame.K_3:
                    ROWS = 12
        CLOCK.tick(FPS)
    # Determine BOARD_PIXELS based on screen size
    max_board_h = SCREEN_H - PANEL_HEIGHT - 50
    max_board_w = SCREEN_W - 50
    BOARD_PIXELS = min(max_board_h, max_board_w)
    BOARD_PIXELS = (BOARD_PIXELS // ROWS) * ROWS
    SQUARE_SIZE = BOARD_PIXELS // ROWS
    WIN = pygame.display.set_mode((BOARD_PIXELS, BOARD_PIXELS + PANEL_HEIGHT))
    pygame.display.set_caption('Enhanced Checkers')

    # Load images
    try:
        RED_IMG = pygame.transform.scale(pygame.image.load(r'images/redcircle.png'),
(SQUARE_SIZE, SQUARE_SIZE))
        GREEN_IMG = pygame.transform.scale(pygame.image.load(r'images/greencircle.png'),
(SQUARE_SIZE, SQUARE_SIZE))
        RED_KING_IMG = pygame.transform.scale(pygame.image.load(r'images/redking.png'),
(SQUARE_SIZE, SQUARE_SIZE))
        GREEN_KING_IMG = pygame.transform.scale(pygame.image.load(r'images/greenking.png'),
(SQUARE_SIZE, SQUARE_SIZE))
        STAR_IMG = pygame.transform.scale(pygame.image.load(r'images/star.png'),
(SQUARE_SIZE//2, SQUARE_SIZE//2))
    except Exception as e:
        print("Error loading images. Ensure 'images' folder has correct files.")
        raise

    # 2. Mode selection
    mode = None         # '1v1' or 'AI'
    ai_algorithm = None # "Minimax" or "AlphaBeta"
    ai_depth = None     # integer

    while mode is None:
        WIN.fill((30, 30, 30))
        title = BIG_FONT.render("Select Mode", True, WHITE)
        WIN.blit(title, title.get_rect(center=(BOARD_PIXELS//2, BOARD_PIXELS//2 - 60)))
```

```
    opt1 = FONT.render("Press 1: Two Players (1v1)", True, WHITE)
    opt2 = FONT.render("Press 2: Play vs AI", True, WHITE)
    WIN.blit(opt1, (BOARD_PIXELS//2 - opt1.get_width()//2, BOARD_PIXELS//2))
    WIN.blit(opt2, (BOARD_PIXELS//2 - opt2.get_width()//2, BOARD_PIXELS//2 + 30))
    pygame.display.update()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_1:
                mode = '1v1'
            elif event.key == pygame.K_2:
                mode = 'AI'
    CLOCK.tick(FPS)

# 3. AI selection
if mode == 'AI':
    while ai_algorithm is None:
        WIN.fill((30, 30, 30))
        prompt = BIG_FONT.render("Select AI Algorithm:", True, WHITE)
        WIN.blit(prompt, prompt.get_rect(center=(BOARD_PIXELS//2, BOARD_PIXELS//2 - 60)))
        o1 = FONT.render("Press 1: Minimax (no pruning)", True, WHITE)
        o2 = FONT.render("Press 2: Minimax w/ Alpha-Beta", True, WHITE)
        WIN.blit(o1, (BOARD_PIXELS//2 - o1.get_width()//2, BOARD_PIXELS//2))
        WIN.blit(o2, (BOARD_PIXELS//2 - o2.get_width()//2, BOARD_PIXELS//2 + 30))
        pygame.display.update()
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_1:
                    ai_algorithm = "Minimax"
                elif event.key == pygame.K_2:
                    ai_algorithm = "AlphaBeta"
        CLOCK.tick(FPS)

    # 4. AI depth
    while ai_depth is None:
        WIN.fill((30, 30, 30))
        prompt = BIG_FONT.render("Select AI Depth (1-8): Press number key", True, WHITE)
        WIN.blit(prompt, prompt.get_rect(center=(BOARD_PIXELS//2, BOARD_PIXELS//2 - 30)))
        pygame.display.update()
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            if event.type == pygame.KEYDOWN:
                if pygame.K_1 <= event.key <= pygame.K_8:
                    depth = event.key - pygame.K_0
                    ai_depth = depth
```

```
        CLOCK.tick(FPS)

# Scores across games
score_green = 0
score_red = 0

# Outer loop
while True:
    board = Board(ROWS)
    highlighted_piece = None
    possible_moves = []
    highlighted_positions = set()

    force_swap_mode = False
    force_swap_player = None

    curr_player = 'G'  # Green starts
    green_time_left = TIME_PER_TURN
    red_time_left = TIME_PER_TURN
    turn_start_ticks = pygame.time.get_ticks()

    # Alert state
    alert_message = None
    alert_start_time = None

    game_over = False
    end_message = ""

    while True:
        now_ticks = pygame.time.get_ticks()
        elapsed = (now_ticks - turn_start_ticks) / 1000.0

        # Timers
        if mode == '1v1' or (mode == 'AI' and curr_player == 'G'):
            if curr_player == 'G':
                green_time_left = TIME_PER_TURN - elapsed
            else:
                red_time_left = TIME_PER_TURN - elapsed

        # Check timeout
        if ((mode == '1v1' and curr_player == 'G' and green_time_left <= 0) or
            (mode == '1v1' and curr_player == 'R' and red_time_left <= 0) or
            (mode == 'AI' and curr_player == 'G' and green_time_left <= 0)):
            game_over = True
            if mode == '1v1':
                if curr_player == 'G':
                    end_message = "Green timed out! Red wins!"
                    score_red += 1
                else:
                    end_message = "Red timed out! Green wins!"
                    score_green += 1
            else:
```

```
            end_message = "Green timed out! Red (AI) wins!"
            score_red += 1

        # Draw background & board
        WIN.fill((0, 0, 0))
        # Draw board
        draw_board(WIN, board, highlighted_positions)
        # Draw info panel with integrated Generate Power buttons
        btn_rects = draw_info_panel(WIN, board, mode, ai_algorithm, ai_depth, curr_player,
                                    red_time_left if mode=='1v1' else None,
                                    green_time_left, score_green, score_red)
        # Draw alert if active
        if alert_message is not None and alert_start_time is not None:
            still = draw_alert(WIN, alert_message, alert_start_time)
            if not still:
                alert_message = None
                alert_start_time = None

        pygame.display.update()
        CLOCK.tick(FPS)

        # Event handling
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

            # If game over: only allow R/Q
            if game_over:
                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_r:
                        break  # restart outer
                    elif event.key == pygame.K_q:
                        pygame.quit()
                        sys.exit()
                continue

            # Detect clicks on Generate Power buttons
            if event.type == pygame.MOUSEBUTTONDOWN:
                mpos = pygame.mouse.get_pos()
                # Green button?
                if btn_rects.get('G') and btn_rects['G'].collidepoint(mpos):
                    if not board.power_used['G']:
                        power = board.generate_random_power('G')
                        if power:
                            # Immediately activate or set force-swap mode
                            if power == PowerType.FORCE_SWAP:
                                force_swap_mode = True
                                force_swap_player = 'G'
                            else:
                                board.activate_power('G', power)
                            # Set alert
```

```
                name = power.value
                desc = POWER_DESCRIPTIONS.get(power, "")
                alert_message = f"{name}\n{desc}"
                alert_start_time = pygame.time.get_ticks()
            continue
        # Red/AI button?
        if btn_rects.get('R') and btn_rects['R'].collidepoint(mpos):
            if not board.power_used['R']:
                power = board.generate_random_power('R')
                if power:
                    if power == PowerType.FORCE_SWAP:
                        force_swap_mode = True
                        force_swap_player = 'R'
                    else:
                        board.activate_power('R', power)
                    # Set alert
                    name = power.value
                    desc = POWER_DESCRIPTIONS.get(power, "")
                    alert_message = f"{name}\n{desc}"
                    alert_start_time = pygame.time.get_ticks()
            continue

    # Human turn handling (select/move pieces)
    human_turn = (mode == '1v1') or (mode == 'AI' and curr_player == 'G')
    if human_turn and event.type == pygame.MOUSEBUTTONDOWN:
        mouse_pos = pygame.mouse.get_pos()
        # Handle force-swap
        if force_swap_mode and force_swap_player == curr_player:
            click = get_clicked_pos(mouse_pos)
            if click:
                r, c = click
                if board.activate_power(force_swap_player, PowerType.FORCE_SWAP, (r, c)):
                    force_swap_mode = False
                    force_swap_player = None
                    board.active_powers[curr_player] = None
                    # Show alert for FORCE_SWAP
                    power = PowerType.FORCE_SWAP
                    name = power.value
                    desc = POWER_DESCRIPTIONS.get(power, "")
                    alert_message = f"{name}\n{desc}"
                    alert_start_time = pygame.time.get_ticks()
            continue

        click = get_clicked_pos(mouse_pos)
        if click:
            r, c = click
            if highlighted_piece and (r, c) in highlighted_positions:
                # Apply move
                for mv in possible_moves:
                    if mv[-1] == (r, c):
                        pause_start = pygame.time.get_ticks()
                        piece_val = board.matrix[highlighted_piece[0]][highlighted_piece[1]]
```

```
                        animate_move(board, mv, piece_val,
                                mode, ai_algorithm, ai_depth, curr_player,
                                red_time_left, green_time_left,
                                score_green, score_red,
                                alert_message, alert_start_time)
                        pause_end = pygame.time.get_ticks()
                        turn_start_ticks += (pause_end - pause_start)
                        # Handle MOVE_TWICE
                        if board.move_twice_active[curr_player] and board.active_powers[curr_player]
== PowerType.MOVE_TWICE:
                            board.active_powers[curr_player] = None
                            board.move_twice_active[curr_player] = False
                            # same player's turn continues
                        else:
                            board.update_turn_effects(curr_player)
                            curr_player = 'R' if curr_player == 'G' else 'G'
                        green_time_left = TIME_PER_TURN
                        red_time_left = TIME_PER_TURN
                        turn_start_ticks = pygame.time.get_ticks()
                        highlighted_piece = None
                        possible_moves = []
                        highlighted_positions.clear()
                        break
                else:
                    val = board.matrix[r][c]
                    if val != 0 and ((val < 0 and curr_player == 'G') or (val > 0 and curr_player == 'R')):
                        highlighted_piece = (r, c)
                        possible_moves = board._get_piece_moves((r, c))
                        highlighted_positions = {mv[-1] for mv in possible_moves}
                    else:
                        highlighted_piece = None
                        possible_moves = []
                        highlighted_positions.clear()

        # If game over: show overlay
        if game_over:
            WIN.fill((0, 0, 0))
            draw_board(WIN, board, set())
            draw_info_panel(WIN, board, mode, ai_algorithm, ai_depth,
                        curr_player,
                        red_time_left if mode=='1v1' else None,
                        green_time_left,
                        score_green, score_red)
            draw_end_message(WIN, end_message)
            pygame.display.update()
            CLOCK.tick(FPS)
            continue

        # Check no moves => game over
        if not game_over:
            moves_avail = board.get_all_moves(curr_player)
            if not moves_avail:
```

```
                game_over = True
                winner = 'R' if curr_player == 'G' else 'G'
                if winner == 'G':
                    end_message = "No moves: Green wins!"
                    score_green += 1
                elif winner == 'R':
                    end_message = "No moves: Red wins!"
                    score_red += 1
                else:
                    end_message = "No moves: Draw!"
                WIN.fill((0, 0, 0))
                draw_board(WIN, board, set())
                draw_info_panel(WIN, board, mode, ai_algorithm, ai_depth,
                                curr_player,
                                red_time_left if mode=='1v1' else None,
                                green_time_left,
                                score_green, score_red)
                draw_end_message(WIN, end_message)
                pygame.display.update()
                CLOCK.tick(FPS)
                continue

            # AI turn
            if mode == 'AI' and curr_player == 'R':
                # AI may auto-generate power here if desired:
                if should_ai_use_power(board, 'R') and not board.power_used['R']:
                    power = board.generate_random_power('R')
                    if power:
                        if power == PowerType.FORCE_SWAP:
                            force_swap_mode = True
                            force_swap_player = 'R'
                        else:
                            board.activate_power('R', power)
                        # Alert
                        name = power.value
                        desc = POWER_DESCRIPTIONS.get(power, "")
                        alert_message = f"{name}\n{desc}"
                        alert_start_time = pygame.time.get_ticks()
                pause_start = pygame.time.get_ticks()
                ai_move = find_best_move(board, 'R', ai_depth, ai_algorithm)
                pause_end = pygame.time.get_ticks()
                turn_start_ticks += (pause_end - pause_start)
                if ai_move:
                    piece_val = board.matrix[ai_move[0][0]][ai_move[0][1]]
                    animate_move(board, ai_move, piece_val,
                                 mode, ai_algorithm, ai_depth, curr_player,
                                 None, green_time_left,
                                 score_green, score_red,
                                 alert_message, alert_start_time)
                if board.move_twice_active['R'] and board.active_powers['R'] ==
PowerType.MOVE_TWICE:
                    board.active_powers['R'] = None
```

```
                    board.move_twice_active['R'] = False
                else:
                    board.update_turn_effects('R')
                    curr_player = 'G'
                green_time_left = TIME_PER_TURN
                red_time_left = TIME_PER_TURN
                turn_start_ticks = pygame.time.get_ticks()
                highlighted_piece = None
                possible_moves = []
                highlighted_positions.clear()

        # End of single game: loops to restart automatically on R key

# Entry point
if __name__ == "__main__":
    main()
```