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](https://www.canva.com/design/DAGtNsouC-g/bHd4Gc4Vqc1OWLBm7drA6A/edit?utm_content=DAGtNsouC-g&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton)  
[סרט הדגמה של המצגת – Google Photos](https://photos.app.goo.gl/AHqXqP6L1u5Uj5TdA)

[סרט הדגמה של המצגת – Youtube](https://youtu.be/Z7dN5dqJ0K4)

[סרט הדגמה של המשחק – Google Photos](https://photos.app.goo.gl/tXhEqa7ec9CofhuS8)

[סרט הדגמה של המשחק – Youtube](https://youtu.be/ebZwFTUdHjI)

[GitHub](https://github.com/Sportify-Sport/Enhanced-Checkers)

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**: 8×8 / 10×10 / 12×12.
* **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(bᵈ).

### **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 →

= score (יריב) Σ – (שחקן) Σ

* 1. **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
  2. **אכילה לאחור:** יתרון לחייל רגיל → מגדילה את מגוון המהלכים החוקיים
  3. **התאמות לגודל לוח** (לוחות גדולים → משקל גבוה יותר על יתרון חומרי).
  4. **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.

**Minimax ו‑Alpha‑Beta**

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:

# הפקת אנימציה חלקה ב‑steps

if capture: remove piece immediately; pause 0.5s

# בסיום: קידום לפי capture\_count

## 

## 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()