Q-learning Maze Game Implementation

This document contains the full implementation of the Q-learning Maze Game using Python, Pygame, and NumPy.  
  
1. Maze Setup:  
 - The game consists of a 5x5 grid.  
 - There are walls, a goal, and an enemy in the maze.  
  
2. Q-learning Setup:  
 - The agent (blue ball) learns to navigate the maze.  
 - The Q-learning algorithm is used to train the agent.  
  
3. The Enemy:  
 - The enemy (red ball) moves in a square pattern.  
 - The agent is penalized if it collides with the enemy.  
  
4. The Goal:  
 - The agent earns a reward when it reaches the goal.  
  
5. Training Process:  
 - The agent explores the maze using the Q-learning algorithm with exploration (epsilon-greedy).  
 - The agent updates its Q-table based on rewards and penalties received.  
  
6. Pygame Visualization:  
 - The maze is visualized with the agent, walls, goal, and enemy.  
 - The agent explores the maze and reaches the goal after training.  
  
Full Implementation:  
  
import numpy as np  
import random  
import pygame  
  
# Maze setup  
MAZE\_SIZE = 5 # Grid size (5x5)  
REWARD\_GOAL = 10  
REWARD\_STEP = -1  
REWARD\_WALL = -5  
REWARD\_ENEMY = -10  
  
# Define maze (0 = empty, 1 = wall, 2 = goal)  
maze = np.zeros((MAZE\_SIZE, MAZE\_SIZE))  
maze[4, 4] = 2 # Goal position at (4, 4)  
  
# Add obstacles (walls)  
wall\_positions = [(1, 1), (1, 2), (2, 3), (3, 1)]  
for pos in wall\_positions:  
 maze[pos] = 1 # Mark as a wall  
  
# Q-learning setup  
alpha = 0.1 # Learning rate  
gamma = 0.9 # Discount factor  
epsilon = 1.0 # Exploration rate  
epsilon\_decay = 0.995 # Epsilon decay rate  
min\_epsilon = 0.1 # Minimum epsilon value  
ACTIONS = [(0, -1), (0, 1), (-1, 0), (1, 0)] # Up, Down, Left, Right  
  
# Initialize Q-table  
Q\_table = np.zeros((MAZE\_SIZE, MAZE\_SIZE, 4))  
  
# Enemy setup (starts at (2, 2))  
enemy\_pos = (2, 2)  
enemy\_moves = [(0, 1), (1, 0), (0, -1), (-1, 0)] # Moves in a square  
enemy\_step = 0 # Track current movement step  
  
# Function to take a step in the environment  
def take\_step(state, action):  
 global enemy\_pos, enemy\_step  
  
 # Move enemy in a fixed pattern  
 enemy\_step = (enemy\_step + 1) % len(enemy\_moves)  
 enemy\_x = max(0, min(enemy\_pos[0] + enemy\_moves[enemy\_step][0], MAZE\_SIZE - 1))  
 enemy\_y = max(0, min(enemy\_pos[1] + enemy\_moves[enemy\_step][1], MAZE\_SIZE - 1))  
 enemy\_pos = (enemy\_x, enemy\_y)  
  
 # Move agent  
 new\_x = max(0, min(state[0] + ACTIONS[action][0], MAZE\_SIZE - 1))  
 new\_y = max(0, min(state[1] + ACTIONS[action][1], MAZE\_SIZE - 1))  
  
 # If new position is a wall, stay in the same place  
 if maze[new\_x, new\_y] == 1:  
 return state, REWARD\_WALL  
  
 new\_state = (new\_x, new\_y)  
  
 # If the agent collides with the enemy, give a penalty  
 if new\_state == enemy\_pos:  
 return new\_state, REWARD\_ENEMY  
  
 reward = REWARD\_GOAL if maze[new\_x, new\_y] == 2 else REWARD\_STEP  
 return new\_state, reward  
  
# Train the agent using Q-learning  
num\_episodes = 2000   
for episode in range(num\_episodes):  
 state = (0, 0) # Start position  
 done = False  
  
 while not done:  
 action = np.random.randint(4) if np.random.rand() < epsilon else np.argmax(Q\_table[state[0], state[1]])  
 new\_state, reward = take\_step(state, action)  
  
 # Q-learning update rule  
 best\_future\_q = np.max(Q\_table[new\_state[0], new\_state[1]])  
 Q\_table[state[0], state[1], action] += alpha \* (reward + gamma \* best\_future\_q - Q\_table[state[0], state[1], action])  
  
 state = new\_state  
 if maze[state[0], state[1]] == 2: # Goal reached  
 done = True  
  
 # Decay epsilon  
 epsilon = max(min\_epsilon, epsilon \* epsilon\_decay)  
  
# Pygame setup  
pygame.init()  
WIDTH, HEIGHT = 500, 500  
CELL\_SIZE = WIDTH // MAZE\_SIZE  
screen = pygame.display.set\_mode((WIDTH, HEIGHT))  
pygame.display.set\_caption("Q-learning Maze Game")  
  
# Colors  
WHITE = (255, 255, 255)  
BLACK = (0, 0, 0)  
GREEN = (0, 255, 0) # Goal  
BLUE = (0, 0, 255) # Agent  
RED = (255, 0, 0) # Enemy  
GRAY = (100, 100, 100) # Obstacles  
  
# Draw the maze  
def draw\_maze():  
 screen.fill(WHITE)  
   
 for x in range(MAZE\_SIZE):  
 for y in range(MAZE\_SIZE):  
 rect = pygame.Rect(x \* CELL\_SIZE, y \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)  
 if maze[x, y] == 1:  
 pygame.draw.rect(screen, GRAY, rect) # Walls  
 elif maze[x, y] == 2:  
 pygame.draw.rect(screen, GREEN, rect) # Goal  
 pygame.draw.rect(screen, BLACK, rect, 2) # Grid outline  
  
# Draw agent  
def draw\_agent(agent\_pos):  
 pygame.draw.circle(screen, BLUE, (agent\_pos[0] \* CELL\_SIZE + CELL\_SIZE // 2, agent\_pos[1] \* CELL\_SIZE + CELL\_SIZE // 2), CELL\_SIZE // 3)  
  
# Draw enemy  
def draw\_enemy():  
 pygame.draw.circle(screen, RED, (enemy\_pos[0] \* CELL\_SIZE + CELL\_SIZE // 2, enemy\_pos[1] \* CELL\_SIZE + CELL\_SIZE // 2), CELL\_SIZE // 3)  
  
# Run the game with AI  
def run\_game():  
 agent\_pos = (0, 0)  
 running = True  
  
 while running:  
 screen.fill(WHITE)  
 draw\_maze()  
 draw\_enemy()  
 draw\_agent(agent\_pos)  
 pygame.display.flip()  
  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 running = False  
  
 # Use Q-table to decide movement  
 action = np.argmax(Q\_table[agent\_pos[0], agent\_pos[1]])  
 agent\_pos, \_ = take\_step(agent\_pos, action)  
  
 pygame.time.delay(300) # Slow down movement  
  
 pygame.quit()  
  
run\_game()