אולג אולוקוב 324320712

רוסלן וסילייב 319333001

פיליפ שלייפר 322357781

**מסמך מסכם**

לינק לסרטון על המצגת-

https://youtu.be/qP-GKfrg2LU

לינק לסרטון על המשחק-

https://youtu.be/gElVKFb2WnI

לינק לגיטהאב-

https://github.com/RoslanVasilew/AdvancedAlgorithmGame

בפרויקט זה, יצרנו משחק בשם "תופסת שחמט" באמצעות הספרייה Pygame.

המשחק משלב אלמנטים של לוח שחמט עם אלמנטים של התחמקות ומניפולציה על מכשולים בשילוב עם כוחות מיוחדים.

### **הבעיה:**

הבעיה שהמשחק מציב בפני השחקן היא להצליח להוביל את הדמות שלו מפסגת לוח השחמט אל התחתית מבלי להיתפס על ידי חלקי השחמט המייצגים אויבים. השחקן צריך להתחמק מהאויבים, לעקוף מכשולים, ולנצל את היכולות המיוחדות שמוקצות לו.

### **פתרון:**

* השתמשנו במספר אלגוריתמים ותהליכים כדי לפתח את המשחק:
  1. **אלגוריתם חיפוש מינימקס עם גיזום אלפא-ביתא (Minimax with Alpha-Beta Pruning)**: כדי לחשב את מהלכי חלקי השחמט היריבים ולמקסם את הנזק לשחקן. האלגוריתם מעריך מהלכים אפשריים לפי מרחק ממיקום השחקן ומכשולים בשטח.
  2. **Heuristic Function**: פונקציה המשמשת להערכת מצבים לפי מיקום השחקן, מרחק ממכשולים, וקירבה לרכבת שיכולה לדחוף אותם מהמפה.
  3. *A Element*\*: שילוב של חישוב (Manhattan Distance) כדי לחשב מרחקי מינימום בין אויבים לשחקן ולהעדיף מהלכים שמקרבים את האויבים לשחקן.

### **אלגוריתמים ותהליכים נוספים:**

* **מניפולציה של מכשולים**: והשחקן יכול להניע מכשולים או להקפיא אוייבים, בהתאם ליכולות שקיבל.
* **חישוב מיקום מחדש של מכשולים**: יצרנו פונקציה שמוודאת שהמכשולים לא ייווצרו במיקום שייצור מצב בלתי אפשרי למשחק או חפיפות עם רכבת הנעה על המפה.

**הקוד:**

**import pygame**

**import sys**

**import random**

**import math**

**from collections import namedtuple**

**# Initialize Pygame**

**pygame.init()**

**size = width, height = 800, 800 # Size of the window**

**screen = pygame.display.set\_mode(size)**

**pygame.display.set\_caption('Chess Evasion Game')**

**# Define constants**

**BOARD\_SIZE = 600 # Size of the chess board**

**BLOCK\_SIZE = BOARD\_SIZE // 8**

**BUTTON\_WIDTH = 200**

**BUTTON\_HEIGHT = 50**

**BUTTON\_MARGIN = 20**

**ABILITY\_TEXT\_SIZE = 24**

**NUM\_OBSTACLES = 5 # Number of obstacles**

**TRAIN\_WIDTH = 2 # Train is 2 blocks wide**

**# Load ability icons**

**PUSH\_ICON = pygame.image.load('Assets/mighty-force.png')**

**FREEZE\_ICON = pygame.image.load('Assets/ice-cube.png')**

**OBSTACLE\_SHIFT\_ICON = pygame.image.load('Assets/earth-spit.png')**

**DOUBLE\_MOVE\_ICON = pygame.image.load('Assets/kangaroo.png')**

**# Resize icons to fit the buttons**

**ICON\_SIZE = (BUTTON\_HEIGHT - 10, BUTTON\_HEIGHT - 10) # Slightly smaller than the button**

**PUSH\_ICON = pygame.transform.scale(PUSH\_ICON, ICON\_SIZE)**

**FREEZE\_ICON = pygame.transform.scale(FREEZE\_ICON, ICON\_SIZE)**

**OBSTACLE\_SHIFT\_ICON = pygame.transform.scale(OBSTACLE\_SHIFT\_ICON, ICON\_SIZE)**

**DOUBLE\_MOVE\_ICON = pygame.transform.scale(DOUBLE\_MOVE\_ICON, ICON\_SIZE)**

**# Load icons**

**RAM\_ICON = pygame.image.load('Assets/ram.svg')**

**WARLORD\_HELMET\_ICON = pygame.image.load('Assets/warlord-helmet.svg')**

**PLAYER\_ICON = pygame.image.load('Assets/crown.png')**

**TRAIN\_ICON = pygame.image.load('Assets/steam-locomotive.png')**

**# Resize the icons to fit your pieces**

**RAM\_ICON = pygame.transform.scale(RAM\_ICON, (BLOCK\_SIZE, BLOCK\_SIZE))**

**WARLORD\_HELMET\_ICON = pygame.transform.scale(WARLORD\_HELMET\_ICON, (BLOCK\_SIZE, BLOCK\_SIZE))**

**PLAYER\_ICON = pygame.transform.scale(PLAYER\_ICON, (BLOCK\_SIZE, BLOCK\_SIZE))**

**TRAIN\_ICON = pygame.transform.scale(TRAIN\_ICON, (BLOCK\_SIZE, BLOCK\_SIZE)) # Resize to fit one block**

**# Colors**

**WHITE = pygame.Color("white")**

**GRAY = pygame.Color("gray")**

**BLUE = pygame.Color("blue")**

**RED = pygame.Color("red")**

**GOLD = pygame.Color("gold")**

**BLACK = pygame.Color("black")**

**GREEN = pygame.Color("green")**

**DARK\_GRAY = pygame.Color("darkgray")**

**# Define an Ability namedtuple**

**Ability = namedtuple('Ability', ['name', 'icon', 'key'])**

**# Define all available abilities**

**ALL\_ABILITIES = [**

**Ability('Freeze', FREEZE\_ICON, pygame.K\_r),**

**Ability('Push', PUSH\_ICON, pygame.K\_t),**

**Ability('Double Move', DOUBLE\_MOVE\_ICON, pygame.K\_y),**

**Ability('Obstacle Shift', OBSTACLE\_SHIFT\_ICON, pygame.K\_u)**

**]**

**# Define the board drawing function**

**def draw\_board(screen):**

***"""***

***Draws the chess board on the screen with alternating colors.***

***"""***

**colors = [WHITE, GRAY]**

**for row in range(8):**

**for col in range(8):**

**color = colors[(row + col) % 2]**

**pygame.draw.rect(screen, color, (col \* BLOCK\_SIZE, row \* BLOCK\_SIZE, BLOCK\_SIZE, BLOCK\_SIZE))**

**def spawn\_or\_respawn\_obstacles(player, pieces, train):**

***"""***

***Spawns or respawns obstacles on the board avoiding player, pieces, and train positions.***

***"""***

**global obstacles**

**new\_obstacles = []**

**available\_positions = [(x, y) for x in range(8) for y in range(8)]**

**# Remove occupied positions**

**occupied\_positions = [player.position] + [piece.position for piece in pieces]**

**if train:**

**occupied\_positions.extend([(train.position[0], train.position[1]), (train.position[0] + 1, train.position[1])])**

**available\_positions = [pos for pos in available\_positions if pos not in occupied\_positions]**

**# Spawn NUM\_OBSTACLES obstacles**

**for \_ in range(NUM\_OBSTACLES):**

**if available\_positions:**

**pos = random.choice(available\_positions)**

**new\_obstacles.append(Obstacle(pos))**

**available\_positions.remove(pos)**

**obstacles = new\_obstacles**

**def draw\_abilities\_window(screen):**

***"""***

***Draws the abilities window with buttons and icons.***

***"""***

**ability\_window\_rect = pygame.Rect(0, height - BUTTON\_HEIGHT - BUTTON\_MARGIN, width, BUTTON\_HEIGHT + BUTTON\_MARGIN)**

**pygame.draw.rect(screen, BLACK, ability\_window\_rect)**

**font = pygame.font.Font(None, ABILITY\_TEXT\_SIZE)**

**# Helper function to draw a button with an icon**

**def draw\_button\_with\_icon(rect, ability, used):**

**pygame.draw.rect(screen, DARK\_GRAY if used else WHITE, rect)**

**screen.blit(ability.icon, (rect.left + 5, rect.top + 5))**

**text\_surface = font.render(ability.name, True, BLACK)**

**text\_rect = text\_surface.get\_rect(midleft=(rect.left + ICON\_SIZE[0] + 10, rect.centery))**

**screen.blit(text\_surface, text\_rect)**

**button\_rects = []**

**for i, ability in enumerate(available\_abilities):**

**button\_rect = pygame.Rect(BUTTON\_MARGIN + i \* (BUTTON\_WIDTH + BUTTON\_MARGIN),**

**height - BUTTON\_HEIGHT - BUTTON\_MARGIN + BUTTON\_MARGIN,**

**BUTTON\_WIDTH, BUTTON\_HEIGHT)**

**used = ability\_used[ability.name]**

**draw\_button\_with\_icon(button\_rect, ability, used)**

**button\_rects.append(button\_rect)**

**return button\_rects**

**def display\_message(screen, message, color, position):**

***"""***

***Displays a message on the screen.***

***"""***

**screen.fill(BLACK) # Fill screen with black before displaying message**

**font = pygame.font.Font(None, 74)**

**text = font.render(message, True, color)**

**rect = text.get\_rect(center=position)**

**screen.blit(text, rect)**

**pygame.display.update()**

**def reset\_game(difficulty):**

***"""***

***Resets the game with the given difficulty level.***

***"""***

**global player, pieces, player\_turn, game\_over, available\_abilities, train, obstacles, ability\_used**

**player = Player((random.randint(0, 7), 0)) # Randomly spawn player on top edge**

**if difficulty == "easy":**

**pieces = [ChessPiece((random.randint(0, 7), 7), 'directional')]**

**train = None**

**obstacles = []**

**elif difficulty == "medium":**

**pieces = [**

**ChessPiece((random.randint(0, 7), 7), 'directional'),**

**ChessPiece((random.randint(0, 7), 7), 'diagonal')**

**]**

**train = Train()**

**obstacles = []**

**else: # hard**

**pieces = [**

**ChessPiece((random.randint(0, 7), 7), 'directional'),**

**ChessPiece((random.randint(0, 7), 7), 'diagonal'),**

**ChessPiece((random.randint(0, 7), 7), 'directional')**

**]**

**train = Train()**

**# Make sure obstacles do not spawn on the train's path**

**spawn\_or\_respawn\_obstacles(player, pieces, train)**

**player\_turn = True**

**game\_over = False**

**# Randomly select 2 abilities**

**available\_abilities = random.sample(ALL\_ABILITIES, 2)**

**# Initialize ability usage tracking**

**ability\_used = {ability.name: False for ability in available\_abilities}**

**# Player class**

**class Player:**

**def \_\_init\_\_(self, position):**

**self.position = position**

**self.color = BLUE**

**self.double\_move\_active = False**

**self.icon = PLAYER\_ICON**

**def draw(self, screen):**

***"""***

***Draws the player on the screen.***

***"""***

**pygame.draw.rect(screen, self.color,**

**(self.position[0] \* BLOCK\_SIZE, self.position[1] \* BLOCK\_SIZE, BLOCK\_SIZE, BLOCK\_SIZE))**

**screen.blit(self.icon, (self.position[0] \* BLOCK\_SIZE, self.position[1] \* BLOCK\_SIZE))**

**def move(self, direction, pieces, obstacles, train, board\_size=8):**

***"""***

***Moves the player in the given direction.***

***"""***

**if self.double\_move\_active:**

**moved = self.\_move\_once(direction, pieces, obstacles, train, board\_size)**

**if moved:**

**self.\_move\_once(direction, pieces, obstacles, train, board\_size)**

**self.double\_move\_active = False**

**else:**

**moved = self.\_move\_once(direction, pieces, obstacles, train, board\_size)**

**return moved**

**def \_move\_once(self, direction, pieces, obstacles, train, board\_size):**

***"""***

***Moves the player one step in the given direction.***

***"""***

**x, y = self.position**

**move\_offsets = {**

**'UP': (0, -1), 'DOWN': (0, 1), 'LEFT': (-1, 0), 'RIGHT': (1, 0),**

**'UP\_LEFT': (-1, -1), 'UP\_RIGHT': (1, -1), 'DOWN\_LEFT': (-1, 1), 'DOWN\_RIGHT': (1, 1)**

**}**

**dx, dy = move\_offsets.get(direction, (0, 0))**

**new\_x, new\_y = x + dx, y + dy**

**if 0 <= new\_x < board\_size and 0 <= new\_y < board\_size:**

**if not any(piece.position == (new\_x, new\_y) for piece in pieces) and \**

**not any(obstacle.position == (new\_x, new\_y) for obstacle in obstacles) and \**

**(train is None or not train.is\_occupied((new\_x, new\_y))) and \**

**(train is None or not self.is\_blocked\_by\_train((x, y), (new\_x, new\_y), train)):**

**self.position = (new\_x, new\_y)**

**return True**

**return False**

**def is\_blocked\_by\_train(self, start, end, train):**

***"""***

***Checks if the player is blocked by the train.***

***"""***

**x1, y1 = start**

**x2, y2 = end**

**tx, ty = train.position**

**if y1 == y2 == ty: # Moving horizontally on the same row as the train**

**min\_x = min(x1, x2)**

**max\_x = max(x1, x2)**

**return min\_x <= tx <= max\_x or min\_x <= tx + 1 <= max\_x**

**return False # Not blocked**

**# ChessPiece class**

**class ChessPiece:**

**def \_\_init\_\_(self, position, type):**

**self.position = position**

**self.type = type**

**self.color = RED if type != 'flag' else GOLD**

**self.frozen = False**

**self.frozen\_turns = 0**

**self.icon = RAM\_ICON if type == 'diagonal' else WARLORD\_HELMET\_ICON**

**def draw(self, screen):**

***"""***

***Draws the chess piece on the screen.***

***"""***

**pygame.draw.rect(screen, self.color,**

**(self.position[0] \* BLOCK\_SIZE, self.position[1] \* BLOCK\_SIZE, BLOCK\_SIZE, BLOCK\_SIZE))**

**screen.blit(self.icon, (self.position[0] \* BLOCK\_SIZE, self.position[1] \* BLOCK\_SIZE))**

**def possible\_moves(self, board\_size=8):**

***"""***

***Returns a list of possible moves for the chess piece.***

***"""***

**x, y = self.position**

**moves = []**

**if self.type == 'directional':**

**move\_offsets = [(0, 1), (0, -1), (1, 0), (-1, 0)]**

**elif self.type == 'diagonal':**

**move\_offsets = [(1, 1), (1, -1), (-1, 1), (-1, -1)]**

**else:**

**move\_offsets = []**

**for dx, dy in move\_offsets:**

**nx, ny = x + dx, y + dy**

**if 0 <= nx < board\_size and 0 <= ny < board\_size:**

**moves.append((nx, ny))**

**return moves**

**def calculate\_move(self, player\_position, pieces, obstacles, train, depth=5):**

***"""***

***Calculates the best move for the chess piece using minimax algorithm with alpha-beta pruning.***

***"""***

**if self.frozen:**

**return self.position**

**best\_move = self.minimax\_alpha\_beta(depth, float('-inf'), float('inf'), True, player\_position, pieces,**

**obstacles, train)**

**# Ensure best\_move is a valid position tuple**

**if isinstance(best\_move, tuple) and len(best\_move) == 2:**

**return best\_move**

**else:**

**# If best\_move is not a valid position, return the current position**

**return self.position**

**def minimax\_alpha\_beta(self, depth, alpha, beta, maximizing\_player, player\_position, pieces, obstacles, train):**

***"""***

***Implements the minimax algorithm with alpha-beta pruning.***

***"""***

**if depth == 0 or self.is\_terminal\_state(player\_position):**

**return self.evaluate\_position(player\_position, obstacles, train)**

**moves = self.possible\_moves()**

**moves.sort(key=lambda move: self.heuristic(move, player\_position, obstacles, train), reverse=maximizing\_player)**

**if maximizing\_player:**

**max\_eval = float('-inf')**

**best\_move = self.position # Default to current position**

**for move in moves:**

**if self.is\_valid\_move(move, pieces, obstacles, train):**

**eval = self.minimax\_alpha\_beta(depth - 1, alpha, beta, False, player\_position, pieces, obstacles,**

**train)**

**if isinstance(eval, tuple):**

**eval = self.evaluate\_position(eval, obstacles, train)**

**if eval > max\_eval:**

**max\_eval = eval**

**best\_move = move**

**alpha = max(alpha, eval)**

**if beta <= alpha:**

**break**

**return best\_move if depth == 5 else max\_eval**

**else:**

**min\_eval = float('inf')**

**for move in moves:**

**if self.is\_valid\_move(move, pieces, obstacles, train):**

**eval = self.minimax\_alpha\_beta(depth - 1, alpha, beta, True, player\_position, pieces, obstacles,**

**train)**

**if isinstance(eval, tuple):**

**eval = self.evaluate\_position(eval, obstacles, train)**

**min\_eval = min(min\_eval, eval)**

**beta = min(beta, eval)**

**if beta <= alpha:**

**break**

**return min\_eval**

**def is\_valid\_move(self, move, pieces, obstacles, train):**

***"""***

***Checks if the move is valid.***

***"""***

**return not any(piece.position == move for piece in pieces if piece != self) and \**

**not any(obstacle.position == move for obstacle in obstacles) and \**

**(not train or not train.is\_occupied(move))**

**def is\_terminal\_state(self, player\_position):**

***"""***

***Checks if the current state is a terminal state.***

***"""***

**return self.position == player\_position**

**def evaluate\_position(self, player\_position, obstacles, train):**

***"""***

***Evaluates the position based on distance to player, obstacles, and train.***

***"""***

**distance = self.manhattan\_distance(self.position, player\_position)**

**obstacle\_penalty = sum(**

**5 for obstacle in obstacles if self.manhattan\_distance(self.position, obstacle.position) < 2)**

**train\_penalty = 10 if train and self.manhattan\_distance(self.position, train.position) < 2 else 0**

**# A\* element: Prefer moves that are closer to the player**

**a\_star\_score = -distance \* 2**

**return a\_star\_score - obstacle\_penalty - train\_penalty**

**def heuristic(self, move, player\_position, obstacles, train):**

***"""***

***Calculates the heuristic value of a move.***

***"""***

**distance = self.manhattan\_distance(move, player\_position)**

**obstacle\_penalty = sum(5 for obstacle in obstacles if self.manhattan\_distance(move, obstacle.position) < 2)**

**train\_penalty = 10 if train and self.manhattan\_distance(move, train.position) < 2 else 0**

**# A\* element: Prefer moves that are closer to the player**

**a\_star\_score = -distance \* 2**

**# Prefer moves that are in the direction of the player**

**direction\_score = self.direction\_score(move, player\_position)**

**return a\_star\_score + direction\_score - obstacle\_penalty - train\_penalty**

**@staticmethod**

**def manhattan\_distance(pos1, pos2):**

***"""***

***Calculates the Manhattan distance between two positions.***

***"""***

**return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])**

**def direction\_score(self, move, player\_position):**

***"""***

***Calculates the direction score for a move.***

***"""***

**current\_distance = self.manhattan\_distance(self.position, player\_position)**

**new\_distance = self.manhattan\_distance(move, player\_position)**

**return 10 if new\_distance < current\_distance else -5**

**def update\_frozen\_status(self):**

***"""***

***Updates the frozen status of the chess piece.***

***"""***

**if self.frozen:**

**self.frozen\_turns += 1**

**if self.frozen\_turns >= 2:**

**self.frozen = False**

**self.frozen\_turns = 0**

**def push\_away(self, player\_position, pieces, obstacles, board\_size=8):**

***"""***

***Pushes the chess piece away from the player.***

***"""***

**direction = (**

**self.position[0] - player\_position[0],**

**self.position[1] - player\_position[1]**

**)**

**magnitude = max(abs(direction[0]), abs(direction[1]))**

**if magnitude != 0:**

**direction = (direction[0] / magnitude, direction[1] / magnitude)**

**new\_x = self.position[0] + int(direction[0] \* 2)**

**new\_y = self.position[1] + int(direction[1] \* 2)**

**if 0 <= new\_x < board\_size and 0 <= new\_y < board\_size:**

**if not any(piece.position == (new\_x, new\_y) for piece in pieces if piece != self) and \**

**not any(obstacle.position == (new\_x, new\_y) for obstacle in obstacles):**

**self.position = (new\_x, new\_y)**

**class Obstacle:**

**def \_\_init\_\_(self, position):**

**self.position = position**

**self.color = BLACK**

**def draw(self, screen):**

***"""***

***Draws the obstacle on the screen.***

***"""***

**pygame.draw.rect(screen, self.color,**

**(self.position[0] \* BLOCK\_SIZE, self.position[1] \* BLOCK\_SIZE, BLOCK\_SIZE, BLOCK\_SIZE))**

**class Train:**

**def \_\_init\_\_(self):**

**self.position = (7, random.randint(1, 6))**

**self.direction = -1**

**self.color = pygame.Color("brown")**

**self.moving = False**

**self.front\_icon = TRAIN\_ICON**

**self.back\_icon = pygame.transform.rotate(TRAIN\_ICON, 180) # Rotate the icon for the back of the train**

**def draw(self, screen):**

***"""***

***Draws the train on the screen.***

***"""***

**x, y = self.position**

**pygame.draw.rect(screen, self.color, (x \* BLOCK\_SIZE, y \* BLOCK\_SIZE, BLOCK\_SIZE \* 2, BLOCK\_SIZE))**

**if self.direction == -1: # Moving left**

**screen.blit(self.front\_icon, (x \* BLOCK\_SIZE, y \* BLOCK\_SIZE))**

**screen.blit(self.back\_icon, ((x + 1) \* BLOCK\_SIZE, y \* BLOCK\_SIZE))**

**else: # Moving right**

**screen.blit(self.back\_icon, (x \* BLOCK\_SIZE, y \* BLOCK\_SIZE))**

**screen.blit(self.front\_icon, ((x + 1) \* BLOCK\_SIZE, y \* BLOCK\_SIZE))**

**def move(self, pieces, board\_size=8):**

***"""***

***Moves the train and pushes any pieces in its way.***

***"""***

**if self.moving:**

**x, y = self.position**

**new\_x = x + self.direction**

**if new\_x < 0 or new\_x >= board\_size - 1: # -1 because train is 2 blocks wide**

**self.direction \*= -1 # Reverse direction**

**new\_x = x + self.direction # Recalculate new\_x with updated direction**

**# Check for pieces to push**

**for piece in pieces:**

**if piece.position[1] == y and (new\_x <= piece.position[0] < new\_x + 2):**

**# Push the piece**

**push\_x = new\_x - 1 if self.direction == -1 else new\_x + 2**

**if 0 <= push\_x < board\_size:**

**piece.position = (push\_x, y)**

**else:**

**# If can't push, don't move the train**

**return**

**self.position = (new\_x, y)**

**self.moving = False # Train moves once per turn**

**def is\_occupied(self, position):**

***"""***

***Checks if the given position is occupied by the train.***

***"""***

**x, y = position**

**tx, ty = self.position**

**return tx <= x < tx + 2 and ty == y**

**def start\_turn(self):**

***"""***

***Allows the train to move on the next turn.***

***"""***

**self.moving = True # Allow the train to move next turn**

**def display\_difficulty\_menu(screen):**

***"""***

***Displays the difficulty selection menu and returns the selected difficulty.***

***"""***

**screen.fill(BLACK)**

**font = pygame.font.Font(None, 74)**

**easy\_text = font.render("Easy", True, GREEN)**

**medium\_text = font.render("Medium", True, pygame.Color("yellow"))**

**hard\_text = font.render("Hard", True, RED)**

**easy\_rect = easy\_text.get\_rect(center=(width // 2, height // 2 - 100))**

**medium\_rect = medium\_text.get\_rect(center=(width // 2, height // 2))**

**hard\_rect = hard\_text.get\_rect(center=(width // 2, height // 2 + 100))**

**screen.blit(easy\_text, easy\_rect)**

**screen.blit(medium\_text, medium\_rect)**

**screen.blit(hard\_text, hard\_rect)**

**pygame.display.flip()**

**while True:**

**for event in pygame.event.get():**

**if event.type == pygame.QUIT:**

**pygame.quit()**

**sys.exit()**

**if event.type == pygame.MOUSEBUTTONDOWN:**

**mouse\_pos = pygame.mouse.get\_pos()**

**if easy\_rect.collidepoint(mouse\_pos):**

**return "easy"**

**elif medium\_rect.collidepoint(mouse\_pos):**

**return "medium"**

**elif hard\_rect.collidepoint(mouse\_pos):**

**return "hard"**

**# Initialize game entities**

**difficulty = display\_difficulty\_menu(screen)**

**reset\_game(difficulty)**

**# Main game loop**

**running = True**

**while running:**

**screen.fill(BLACK) # Clear screen**

**button\_rects = draw\_abilities\_window(screen)**

**for event in pygame.event.get():**

**if event.type == pygame.QUIT:**

**pygame.quit()**

**sys.exit()**

**elif event.type == pygame.KEYDOWN:**

**if player\_turn:**

**key\_mapping = {**

**pygame.K\_w: 'UP', pygame.K\_s: 'DOWN', pygame.K\_a: 'LEFT', pygame.K\_d: 'RIGHT',**

**pygame.K\_q: 'UP\_LEFT', pygame.K\_e: 'UP\_RIGHT', pygame.K\_z: 'DOWN\_LEFT', pygame.K\_c: 'DOWN\_RIGHT'**

**}**

**direction = key\_mapping.get(event.key)**

**if direction:**

**moved = player.move(direction, pieces, obstacles, train)**

**if moved:**

**player\_turn = False**

**if event.key == pygame.K\_r and not freeze\_used:**

**player\_x, player\_y = player.position**

**for piece in pieces:**

**if piece.type != 'flag':**

**piece\_x, piece\_y = piece.position**

**if abs(piece\_x - player\_x) <= 2 and abs(piece\_y - player\_y) <= 2:**

**piece.frozen = True**

**piece.frozen\_turns = 0**

**freeze\_used = True**

**if event.key == pygame.K\_t and not push\_used:**

**player\_x, player\_y = player.position**

**for piece in pieces:**

**if piece.type != 'flag':**

**piece.push\_away((player\_x, player\_y), pieces, obstacles)**

**push\_used = True**

**if event.key == pygame.K\_y and not double\_move\_used:**

**player.double\_move\_active = True**

**double\_move\_used = True**

**elif event.type == pygame.MOUSEBUTTONDOWN:**

**mouse\_x, mouse\_y = event.pos**

**for i, button\_rect in enumerate(button\_rects):**

**if button\_rect.collidepoint(mouse\_x, mouse\_y):**

**ability = available\_abilities[i]**

**if not ability\_used[ability.name]:**

**if ability.name == 'Freeze':**

**player\_x, player\_y = player.position**

**for piece in pieces:**

**if piece.type != 'flag':**

**piece\_x, piece\_y = piece.position**

**if abs(piece\_x - player\_x) <= 2 and abs(piece\_y - player\_y) <= 2:**

**piece.frozen = True**

**piece.frozen\_turns = 0**

**elif ability.name == 'Push':**

**player\_x, player\_y = player.position**

**for piece in pieces:**

**if piece.type != 'flag':**

**piece.push\_away((player\_x, player\_y), pieces, obstacles)**

**elif ability.name == 'Double Move':**

**player.double\_move\_active = True**

**elif ability.name == 'Obstacle Shift':**

**spawn\_or\_respawn\_obstacles(player, pieces, train)**

**ability\_used[ability.name] = True**

**if not player\_turn and not game\_over:**

**for piece in pieces:**

**if piece.type != 'flag':**

**if not piece.frozen:**

**new\_position = piece.calculate\_move(player.position, pieces, obstacles, train)**

**if new\_position:**

**piece.position = new\_position**

**piece.update\_frozen\_status()**

**player\_turn = True**

**if train:**

**train.start\_turn()**

**if not game\_over and train:**

**train.move(pieces)**

**# Check if player reached the bottom row**

**if player.position[1] == 7:**

**display\_message(screen, "U WIN", GREEN, (width // 2, height // 2))**

**pygame.time.delay(2000) # Pause for 2 seconds**

**difficulty = display\_difficulty\_menu(screen) # Show difficulty menu again**

**reset\_game(difficulty) # Reset the game with new difficulty**

**for piece in pieces:**

**if piece.type != 'flag' and piece.position == player.position:**

**display\_message(screen, "YOU DIED", RED, (width // 2, height // 2))**

**pygame.time.delay(2000) # Pause for 2 seconds**

**difficulty = display\_difficulty\_menu(screen) # Show difficulty menu again**

**reset\_game(difficulty) # Reset the game with new difficulty**

**break**

**# Draw the chess board**

**draw\_board(screen)**

**# Draw the obstacles**

**for obstacle in obstacles:**

**obstacle.draw(screen)**

**# Draw the player, pieces, and train**

**player.draw(screen)**

**for piece in pieces:**

**piece.draw(screen)**

**if train:**

**train.draw(screen)**

**pygame.display.flip()**