QUORIDOR GAME IMPLEMENTATION

CSE472S: ARTIFICIAL INTELLIGENCE

TEAM MEMBERS

NAME	CODE
Mahmoud Mostafa Mahmoud	**2100403**
Khaled Mohamed Mohamed Ali	**2100689**
Mohamed Abdelbaky Tony Abdelbaky	**2101710**
Rana Ayman Hamdy	**2100830**
Farah Amr Mohamed	**2101034**

PROJECT DESCRIPTION

Quoridor is a strategic two-player board game implemented in Python with a modern graphical interface. The game features an AI opponent with multiple difficulty levels, allowing players to compete against the computer or another human player.

GAME RULES

- Each player starts on opposite ends of a 9×9 board
- Players take turns either moving their pawn or placing a wall
- Pawns can move one cell horizontally or vertically (or jump over opponent)
- Each player has 10 walls to block the opponent's path
- Walls cannot completely block a player's path to their goal
- The first player to reach the opposite side wins

PROJECT STRUCTURE

```
quoridor_game/
│
├── main.py                    # Application entry point
├── requirements.txt           # Python dependencies
├── README.md                  # Project documentation
└── src/                       # Source code modules
    ├── __init__.py            # Package initializer
    ├── constants.py           # Game constants and
configuration
    ├── game_logic.py          # Core game rules and
mechanics
    ├── ai.py                  # AI opponent implementation
    ├── board_renderer.py      # Board visualization
    ├── ui_components.py       # Reusable UI widgets
    ├── screens.py             # Menu and overlay screens
    └── gui.py                 # Main GUI controller
```

ALGORITHMS USED

1. MINIMAX ALGORITHM WITH ALPHA-BETA PRUNING

Location: src/ai.py

Purpose: Used for AI decision-making to determine the best move for the computer opponent.

Why Used:

- **Optimal for two-player games**: Minimax is the standard algorithm for adversarial games where two players take turns
- **Guaranteed optimal play**: Finds the best possible move assuming the opponent also plays optimally
- **Alpha-Beta Pruning optimization**: Reduces the number of nodes evaluated in the game tree by eliminating branches that cannot influence the final decision
- **Configurable difficulty**: By adjusting the search depth, we can create Easy (depth 1), Medium (depth 2), and Hard (depth 3) AI opponents

How It Works:

```
function minimax(state, depth, alpha, beta, maximizingPlayer):
    if depth = 0 or game over:
        return evaluate(state)

    if maximizingPlayer:
        maxEval = -infinity
        for each move in valid_moves:
            eval = minimax(apply(move), depth-1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta cutoff
        return maxEval
    else:
        minEval = +infinity
        for each move in valid_moves:
            eval = minimax(apply(move), depth-1, alpha, beta, true)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Alpha cutoff
        return minEval
```

Complexity:

- **Without pruning**: O(b^d) where b = branching factor, d = depth
- **With Alpha-Beta**: O(b^(d/2)) in best case

II. BREADTH-FIRST SEARCH (BFS)

Location: src/game_logic.py

Purpose: Used for pathfinding and path validation.

Why Used:

- **Shortest path guarantee**: BFS always finds the shortest path in an unweighted graph
- **Complete algorithm**: Will always find a path if one exists
- **Path validation**: Essential for checking if a wall placement is legal (must not block all paths to goal)
- **Move calculation**: Used to determine valid moves including special jump moves

How It Works:

```
function bfs_has_path(start, goal_row):
    queue = [start]
    visited = {start}

    while queue not empty:
        current = queue.pop_front()
        if current.row == goal_row:
            return True

        for neighbor in get_valid_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return False
```

Use Cases in Project:

1. **Wall validation**: Before placing a wall, check that both players still have a path to their goal
2. **AI evaluation**: Calculate distance to goal for position evaluation
3. **Valid moves**: Determine which cells a player can move to

Complexity

O(V + E) where V = vertices (81 cells), E = edges

III. HEURISTIC EVALUATION FUNCTION

Location: src/ai.py

Purpose: Evaluates board positions for the Minimax algorithm.

Why Used:

- **Position scoring**: Converts game state into a numerical value
- **Strategic factors**: Considers multiple aspects of the game
- **Performance**: Fast computation allows deeper search

Evaluation Factors:

Factor	Weight	Description
Distance to goal	High	Fewer steps = better position
Walls remaining	Medium	More walls = more options
Opponent's distance	Medium	Farther opponent = advantage
Path options	Low	Multiple paths = flexibility

CODE OVERVIEW BY MODULE

- **main.py** - Entry point that initializes and runs the game.
- **src/constants.py** - Contains all game configuration: Screen dimensions and colors, Board sizes and cell dimensions, Animation speeds and UI settings.
- **src/game_logic.py** - Core game mechanics: QuoridorGame class - Main game state management, GameState dataclass - Stores current game state, Move validation and execution, Wall placement logic with BFS validation, Win condition checking.
- **src/ai.py** - AI opponent implementation: QuoridorAI class - AI player, Minimax with Alpha-Beta pruning, Three difficulty levels, Strategic move generation.
- **src/board_renderer.py** - Visual board representation: Cell and wall rendering, Pawn animations, Valid move indicators, Hover effects.
- **src/ui_components.py** - Reusable UI elements: ModernButton - Animated buttons, GlassPanel - Glassmorphism panels, ProgressBar - Wall count display, Tooltip - Help text.
- **src/screens.py** - Screen management: MenuScreen - Main menu with animated title, GameOverOverlay - Victory screen, ScreenManager - Screen transitions.
- **src/gui.py** - Main GUI controller: Event handling, Game loop, Screen coordination, Fullscreen support.

LINKS

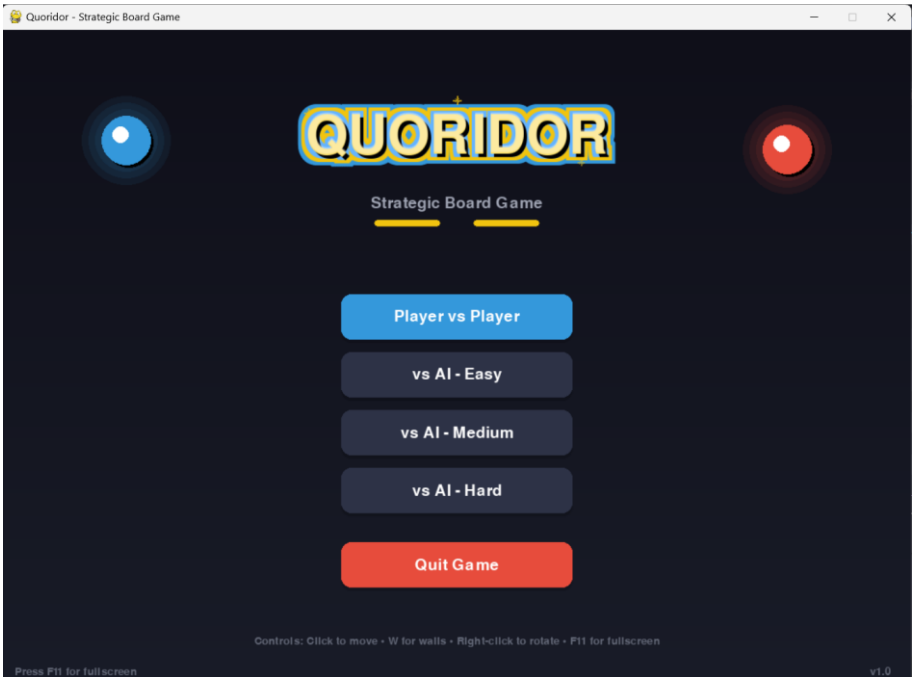
Resource	Link
GitHub Repository	https://github.com/MahMoudMostaAfa/AI-Quoridor-Game
Demo Video	https://drive.google.com/drive/folders/1sNrzOPxb1PpBLQ9oGdKHJZDytU5FwSwx?usp=sharing

GAME CONTROLS

Control	Action
Left Click	Move pawn to highlighted cell
W Key	Toggle wall placement mode
Right Click	Rotate wall orientation
ESC	Cancel wall placement / Exit fullscreen
F11	Toggle fullscreen mode

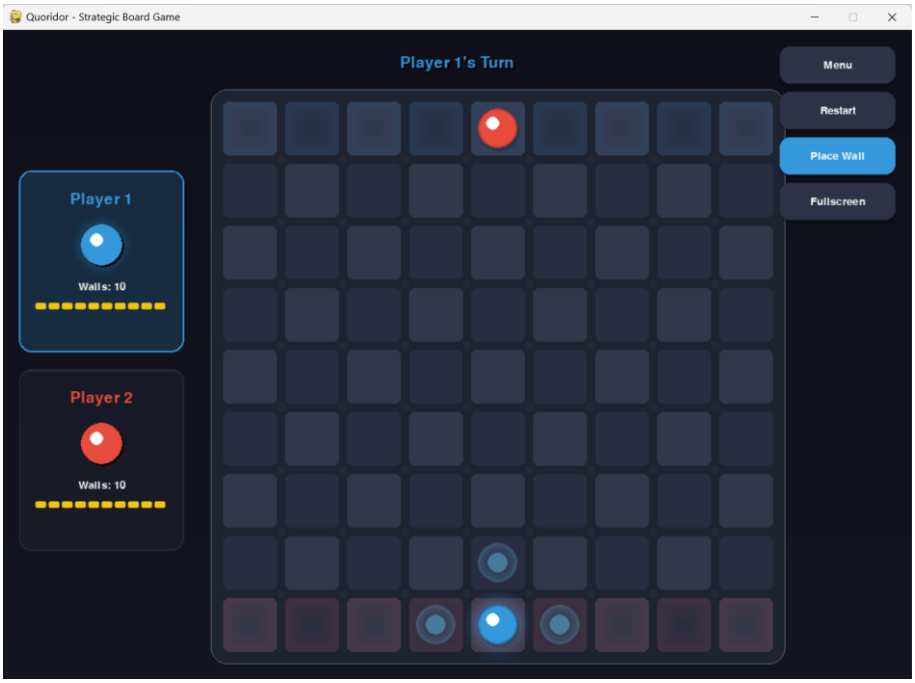
SCREENSHOTS

1. MAIN MENU



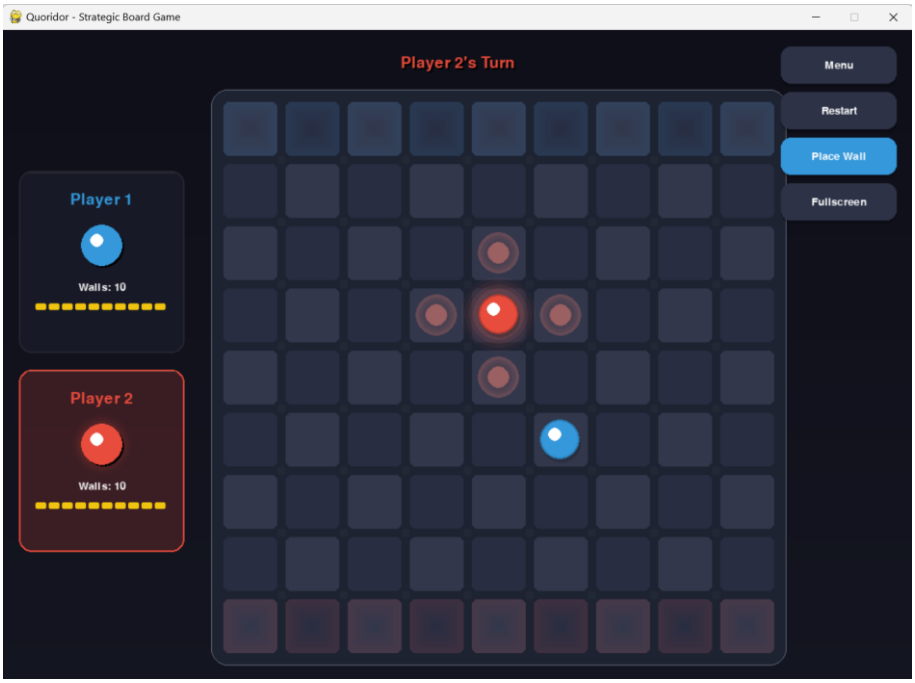
Description: The main menu featuring the animated "QUORIDOR" title with sparkle effects, game mode selection buttons (PvP and AI difficulties), and control instructions.

2. GAME BOARD - PLAYER VS PLAYER



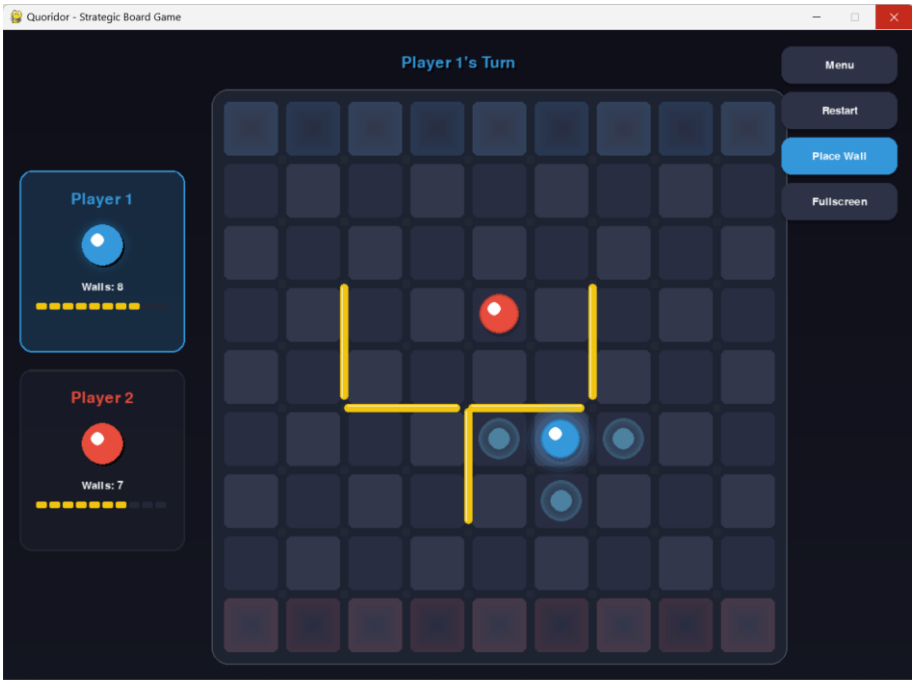
Description: Two-player gameplay showing the 9×9 board with both pawns, placed walls, and player information panels on the left side.

3. VALID MOVES INDICATOR



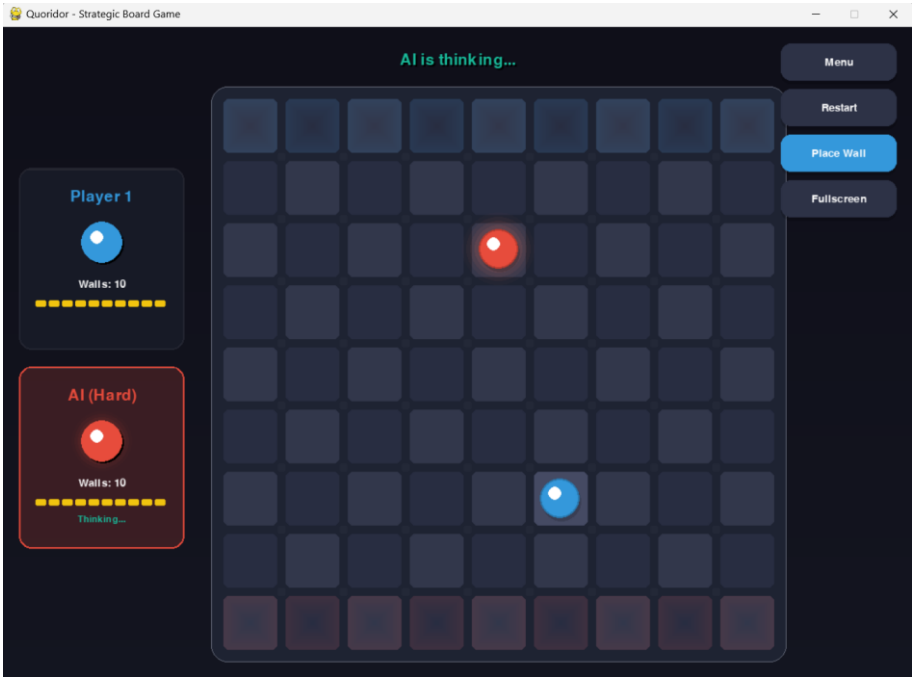
Description: Player's available moves highlighted with their color (cyan for Player 1, coral for Player 2) with pulsing animation effect.

4. WALL PLACEMENT MODE



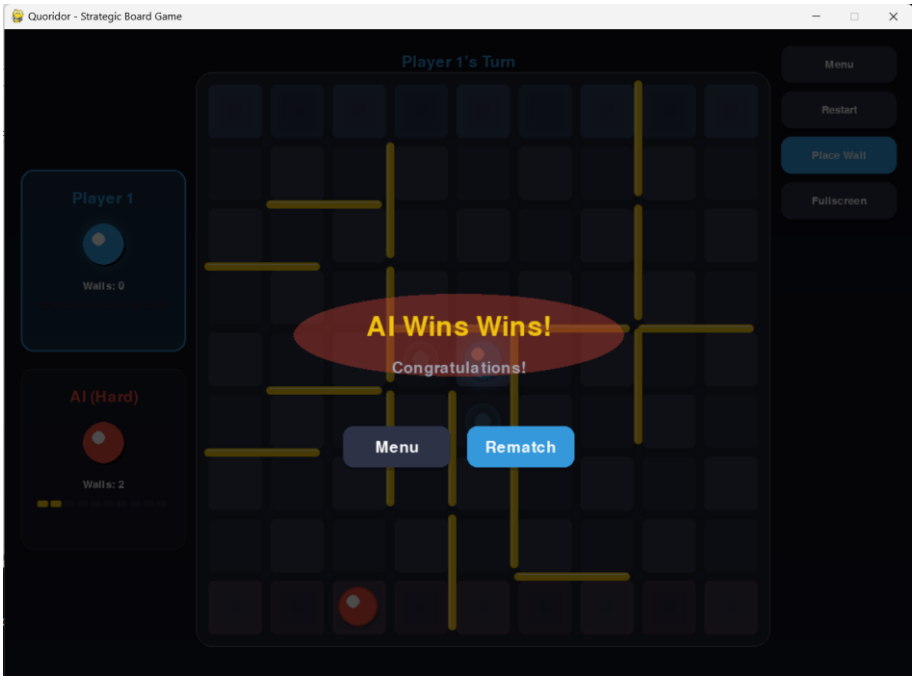
Description: Wall placement preview showing where the wall will be placed, with green indicator for valid placement and red for invalid.

5. AI OPPONENT GAMEPLAY



Description: Playing against the AI opponent, showing the "AI is thinking..." indicator and difficulty level display.

6. GAME OVER SCREEN



Description: The victory overlay showing the winner announcement with celebratory effects and options to rematch or return to menu

REFERENCES

1. Minimax Algorithm - Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach

2. Alpha-Beta Pruning - Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning

3. Pygame Documentation - https://www.pygame.org/docs/

4. Quoridor Official Rules - Gigamic Games