

# Testing :

## 1 Unit testing :

### 1.1 Game Logic testing :

We have done 19 tests on the game logic to make sure it works properly. These are some of the tests and their explanations :

#### 1.1.1 Game initializes correctly

```
// Test Game Initialization
TEST_F(GameEngineTest, GameInitializesCorrectly) {
    Board board = game.get_board();

    // Check board is 3x3 and empty
    ASSERT_EQ(board.size(), 3);
    for (int i = 0; i < 3; ++i) {
        ASSERT_EQ(board[i].size(), 3);
        for (int j = 0; j < 3; ++j) {
            EXPECT_EQ(board[i][j], ' ');
        }
    }

    // Check initial player is X
    EXPECT_EQ(game.getCurrentPlayer(), 'X');
    EXPECT_EQ(game.getOpponentPlayer(), 'O');

    // Check move history is empty
    EXPECT_TRUE(game.get_moveHistory().empty());

    // Check game is not over initially
    EXPECT_FALSE(game.isGameOver());
}
```

The test verifies that the game initializes correctly with:

- A 3x3 empty board (all cells contain spaces)
- Player 'X' as the starting player
- Player 'O' as the opponent
- Empty move history
- Game not in a finished state initially

### 1.1.2 Invalid move on occupied cell

```
TEST_F(GameEngineTest, InvalidMoveOnOccupiedCell) {
    // Make a move at position 1
    ASSERT_TRUE(game.makeMove(1, "First move"));

    // Position 1 should now be invalid
    EXPECT_FALSE(game.isValidMove(1));

    // Other positions should still be valid
    EXPECT_TRUE(game.isValidMove(2));
    EXPECT_TRUE(game.isValidMove(9));
}
```

This test verifies that once a cell is occupied, it becomes invalid for future moves :

What it does :

- Makes a move at position 1 (places 'X' there)
- Checks that position 1 is now invalid (return false)
- Confirms other empty positions (2 and 9) are still valid

### 1.1.3 Undo move

```
// Test Undo Functionality
TEST_F(GameEngineTest, UndoMove) {
    // Make some moves
    game.makeMove(1, "First move");
    game.makeMove(2, "Second move");

    EXPECT_EQ(game.getCurrentPlayer(), 'X');
    EXPECT_EQ(game.get_moveHistory().size(), 2);

    // Undo last move
    EXPECT_TRUE(game.undoMove());
    EXPECT_EQ(game.getCurrentPlayer(), 'O');
    EXPECT_EQ(game.get_moveHistory().size(), 1);

    Board board = game.get_board();
    EXPECT_EQ(board[0][0], 'X'); // First move still there
    EXPECT_EQ(board[0][1], ' '); // Second move undone
}
```

The test verifies that the game supports undoing moves by:

- Removing the last move from history
- Clearing the corresponding board position
- Reverting to the previous player's turn

#### 1.1.4 Check win

##### 1.1.4.1 Check win by rows

```
// Test Win Conditions - Rows
TEST_F(GameEngineTest, CheckWinHorizontalRows) {
    // Test all three rows for X wins
    game.Game_init();
    game.makeMove(1, ""); game.makeMove(4, ""); // X: (0,0), O: (1,0)
    game.makeMove(2, ""); game.makeMove(5, ""); // X: (0,1), O: (1,1)
    game.makeMove(3, ""); // X: (0,2) - X wins row 0
    EXPECT_TRUE(game.checkWin("X"));
    EXPECT_FALSE(game.checkWin("O"));

    game.Game_init();
    game.makeMove(4, ""); game.makeMove(1, ""); // X: (1,0), O: (0,0)
    game.makeMove(5, ""); game.makeMove(2, ""); // X: (1,1), O: (0,1)
    game.makeMove(6, ""); // X: (1,2) - X wins row 1
    EXPECT_TRUE(game.checkWin("X"));

    game.Game_init();
    game.makeMove(7, ""); game.makeMove(1, ""); // X: (2,0), O: (0,0)
    game.makeMove(8, ""); game.makeMove(2, ""); // X: (2,1), O: (0,1)
    game.makeMove(9, ""); // X: (2,2) - X wins row 2
    EXPECT_TRUE(game.checkWin("X"));
}
```

The test verifies that the game checks all three rows :

- Row 0 : positions 1-2-3
- Row 1 : positions 4-5-6
- Row 2 : positions 7-8-9

##### 1.1.4.2 Check win by diagonals

```
// Test Win Conditions - Diagonals
TEST_F(GameEngineTest, CheckWinDiagonals) {
    // Test main diagonal (top-left to bottom-right)
    game.Game_init();
    game.makeMove(1, ""); game.makeMove(2, ""); // X: (0,0), O: (0,1)
    game.makeMove(5, ""); game.makeMove(3, ""); // X: (1,1), O: (0,2)
    game.makeMove(9, ""); // X: (2,2) - X wins main diagonal
    EXPECT_TRUE(game.checkWin("X"));

    // Test anti-diagonal (top-right to bottom-left)
    game.Game_init();
    game.makeMove(3, ""); game.makeMove(1, ""); // X: (0,2), O: (0,0)
    game.makeMove(5, ""); game.makeMove(2, ""); // X: (1,1), O: (0,1)
    game.makeMove(7, ""); // X: (2,0) - X wins anti-diagonal
    EXPECT_TRUE(game.checkWin("X")); }
```

The test verifies that the game checks two diagonals :

- Main diagonal : positions 1-5-9
- Anti-diagonal : positions 3-5-7

#### 1.1.5 Check draw

```
// Test Draw Conditions
TEST_F(GameEngineTest, CheckDraw) {
    // Fill board without winner: X O O / O X X / X X O
    game.makeMove(1, ""); // X
    game.makeMove(2, ""); // O
    game.makeMove(5, ""); // X
    game.makeMove(3, ""); // O
    game.makeMove(7, ""); // X
    game.makeMove(4, ""); // O
    game.makeMove(8, ""); // X
    game.makeMove(9, ""); // O
    game.makeMove(6, ""); // X

    EXPECT_FALSE(game.checkWin("X"));
    EXPECT_FALSE(game.checkWin("O"));
    EXPECT_TRUE(game.checkDraw());
    EXPECT_TRUE(game.isGameOver());
}
```

The test verifies that the game supports checking draw occurs when :

- All 9 board positions are filled
- No player has achieved a winning combination
- The game automatically ends in this scenario

#### 1.1.6 Gameover

##### 1.1.6.1 Gameover on win

```
TEST_F(GameEngineTest, GameOverOnWin) {
    // Create X win
    game.makeMove(1, ""); game.makeMove(4, "");
    game.makeMove(2, ""); game.makeMove(5, "");
    game.makeMove(3, ""); // X wins

    EXPECT_TRUE(game.isGameOver());
}
```

The test verifies that the game correctly detects when it's over due to a win condition :

What it does :

- X plays positions 1,2,3 (top row)
- O plays positions 4,5 (middle row, left and center)
- After X completes the top row, X wins
- Checks that isGameOver() returns true

#### 1.1.6.2 *Gameover on draw*

```
TEST_F(GameEngineTest, GameOverOnDraw) {  
    // Create draw scenario  
    game.makeMove(1, ""); game.makeMove(2, "");  
    game.makeMove(3, ""); game.makeMove(4, "");  
    game.makeMove(5, ""); game.makeMove(6, "");  
    game.makeMove(7, ""); game.makeMove(8, "");  
    game.makeMove(9, "");  
  
    EXPECT_TRUE(game.isGameOver());  
}
```

The test verifies that the game correctly detects when it's over due to a draw condition :

What it does :

- Fills all 9 positions on the board alternately (X and O)
- X plays positions 1,3,5,7,9 (odd positions)
- O plays positions 2,4,6,8 (even positions)
- After all cells are filled, checks that isGameOver() returns true

## 1.2 AI testing :

We have done 22 tests on the AI to make sure it works properly. These are some of the tests and their explanations :

### 1.2.1 Evaluate board functionality

#### 1.2.1.1 *Evaluate board detects X win in rows*

```
TEST_F(AITest, EvaluateBoardDetectsXWinInRows) {  
    std::vector<Board> winning_boards = {  
        CreateBoard({"XXX", "O O", " "}), // Row 0  
        CreateBoard({"O O", "XXX", " "}), // Row 1  
        CreateBoard({"O O", " ", "XXX"}) // Row 2  
    };  
};
```

```

for (const auto& board : winning_boards) {
    EXPECT_EQ(ai.EvaluateBoard(board, 'X', 'O'), 10);
}
}

```

This test verifies that the AI correctly identifies when player X has won by completing any horizontal row. It tests all three possible row victories (top, middle, bottom) and expects the evaluation function to return a score of 10, indicating X's victory. This ensures the AI can properly recognize winning game states.

#### 1.2.1.2 *Evaluate board detects incomplete game*

```

TEST_F(AITest, EvaluateBoardDetectsIncompleteGame) {
    Board incomplete = CreateBoard({"X O", "X ", "O "});
    EXPECT_EQ(ai.EvaluateBoard(incomplete, 'X', 'O'), 0);
}

```

This test confirms that the AI correctly evaluates ongoing games where neither player has won yet. With a mixed board state containing both X's and O's but no winning lines, the function should return 0 (neutral score). This is crucial for the AI to distinguish between terminal and non-terminal game states.

#### 1.2.2 *Get legal moves on partial board*

```

TEST_F(AITest, GetLegalMovesOnPartialBoard) {
    Board partial = CreateBoard({"X O", "X ", "O "});
    auto moves = ai.GetLegalMoves(partial);

    EXPECT_EQ(moves.size(), 5);

    std::set<std::pair<int, int>> expected = {{0,1}, {1,0}, {1,2}, {2,1}, {2,2}};
    std::set<std::pair<int, int>> actual(moves.begin(), moves.end());

    EXPECT_EQ(expected, actual);
}

```

This test validates that the AI accurately identifies all available moves on a partially filled board. Given a specific board configuration with 4 occupied squares, it expects exactly 5 legal moves and verifies the precise coordinates of each empty position. This ensures the AI only considers valid moves during decision-making.

#### 1.2.3 *AI decision making*

##### 1.2.3.1 *AI takes immediate win over block*

```

TEST_F(AITest, AITakesImmediateWinOverBlock) {
    Board board = CreateBoard({"XX ", "OO ", "  "});
    auto move = ai.GetBestMove(board, 'X');

    // AI should complete its own winning line
}

```

```
EXPECT_EQ(move.first, 0);
EXPECT_EQ(move.second, 2);
}
```

This test ensures the AI prioritizes winning over defensive play when both options are available. With X having two in a row and O also having two in a row, the AI should choose to complete its own winning line rather than block the opponent. This tests the AI's move prioritization logic.

#### 1.2.3.2 AI blocks opponent win when no win available

```
TEST_F(AITest, AIBlocksOpponentWinWhenNoWinAvailable) {
    Board board = CreateBoard({"OO ", "X ", " X "});
    auto move = ai.GetBestMove(board, 'X');

    // AI should block O's win
    EXPECT_EQ(move.first, 0);
    EXPECT_EQ(move.second, 2);
}
```

This test verifies that the AI plays defensively when it cannot win immediately. With O threatening to win on the next move and no winning opportunity for X, the AI should block O's winning move. This ensures the AI can recognize and respond to immediate threats.

#### 1.2.4 Generate explanation functionality

##### 1.2.4.1 Generate explanation detects winning move

```
TEST_F(AITest, GenerateExplanationDetectsWinningMove) {
    Board before = CreateBoard({"XX ", "OO ", "  "});
    Board after = before;
    after[0][2] = 'X';

    std::string explanation = ai.GenerateExplanation(before, after, 'X', 'O');
    EXPECT_NE(explanation.find("Winning move"), std::string::npos);
}
```

This test checks that the AI can properly explain its winning moves to users. After simulating a game-winning move, the explanation should contain text indicating it was a "Winning move". This ensures the AI provides meaningful feedback about its strategic decisions.

##### 1.2.4.2 Generate explanation detects blocking move

```
TEST_F(AITest, GenerateExplanationDetectsBlockingMove) {
    Board before = CreateBoard({"OO ", "X ", " X "});
    Board after = before;
    after[0][2] = 'X'; // Blocking move

    std::string explanation = ai.GenerateExplanation(before, after, 'X', 'O');
    EXPECT_NE(explanation.find("Blocked opponent"), std::string::npos);
}
```

```
}
```

This test validates that the AI can explain defensive moves appropriately. When the AI makes a move that blocks an opponent's winning threat, the explanation should mention "Blocked opponent". This helps users understand the AI's defensive reasoning and strategic thinking.

### 1.3 User System testing :

We have done 19 tests on the user system to make sure it works properly. These are some of the tests and their explanations :

#### 1.3.1 Core functionality tests

##### 1.3.1.1 *Successful user registration*

```
TEST_F(UserSystemTest, RegisterUserSuccess) {  
    bool result = userSystem->registerUser("testuser", "password123");  
    EXPECT_TRUE(result);  
}
```

This test verifies the basic user registration functionality works correctly. It attempts to register a new user with valid credentials and expects the operation to succeed (return true). This ensures the core user creation process functions properly and users can be added to the system.

##### 1.3.1.2 *Login with wrong password*

```
TEST_F(UserSystemTest, LoginUserWrongPassword) {  
    userSystem->registerUser("testuser", "password123");  
    bool result = userSystem->loginUser("testuser", "wrongpassword");  
    EXPECT_FALSE(result);  
}
```

This test validates that the authentication system properly rejects invalid login attempts. After registering a user with a specific password, it tries to log in with an incorrect password and expects failure (return false). This ensures the system maintains security by rejecting unauthorized access attempts.

##### 1.3.1.3 *Load game moves with comments*

```
TEST_F(UserSystemTest, LoadGameMovesWithComments) {  
    std::vector<std::pair<int, std::string>> moves = {  
        {0, "First move"},  
        {4, "Center control"},  
        {8, ""} // Empty comment  
    };  
  
    userSystem->saveGameWithMoves("player1", "player2", "X", moves);  
}
```



```

auto history = userSystem->getGameHistory("player1");
ASSERT_FALSE(history.empty());

int gameId = std::get<0>(history[0]);
auto movesLoaded = userSystem->loadGameMovesWithComments(gameId);

ASSERT_EQ(movesLoaded.size(), moves.size());
for (size_t i = 0; i < moves.size(); ++i) {
    EXPECT_EQ(movesLoaded[i].first, moves[i].first);
    EXPECT_EQ(movesLoaded[i].second, moves[i].second);
}
}

```

This test verifies that the system can save and retrieve complete game data including move positions and associated comments. It saves a game with three moves (some with comments, one empty), then loads the data back and confirms all move positions and comments are preserved exactly. This ensures game replay functionality works correctly.

#### 1.3.1.4 Head-to-head statistics

```

TEST_F(UserSystemTest, GetHeadToHeadStats) {
    std::vector<std::pair<int, std::string>> moves = {{0, ""}};

    // userA wins as X, userA wins as O, Tie game
    userSystem->saveGameWithMoves("userA", "userB", "X", moves);
    userSystem->saveGameWithMoves("userB", "userA", "O", moves);
    userSystem->saveGameWithMoves("userA", "userB", "Tie", moves);

    int userAWins, userBWins, ties;
    std::tie(userAWins, userBWins, ties) = userSystem->getHeadToHeadStats("userA", "userB");

    EXPECT_EQ(userAWins, 2);
    EXPECT_EQ(userBWins, 0);
    EXPECT_EQ(ties, 1);
}

```

This test validates the head-to-head statistics calculation between two specific players. It creates three games between userA and userB with different outcomes (userA wins twice, one tie), then verifies the statistics correctly show userA with 2 wins, userB with 0 wins, and 1 tie. This ensures accurate player-vs-player performance tracking.

#### 1.3.1.5 Human vs AI statistics

```

TEST_F(UserSystemTest, GetHumanVsAIStats) {
    std::vector<std::pair<int, std::string>> moves = {{0, ""}};

    // Human wins, AI wins, Tie game
    userSystem->saveGameWithMoves("human", "AI", "X", moves);

```

```

userSystem->saveGameWithMoves("AI", "human", "X", moves);
userSystem->saveGameWithMoves("human", "AI", "Tie", moves);

int humanWins, aiWins, ties;
std::tie(humanWins, aiWins, ties) = userSystem->getHumanVsAIStats("human");

EXPECT_EQ(humanWins, 1);
EXPECT_EQ(aiWins, 1);
EXPECT_EQ(ties, 1);
}

```

This test checks that human vs AI statistics are calculated correctly. It creates three games between a human player and AI with different outcomes (1 human win, 1 AI win, 1 tie), then verifies the stats accurately reflect each outcome type. This ensures proper tracking of human performance against AI opponents.

### 1.3.2 Head-to-head stats with no games

```

TEST_F(UserSystemTest, HeadToHeadStatsNoGames) {
    int user1Wins, user2Wins, ties;
    std::tie(user1Wins, user2Wins, ties) = userSystem->getHeadToHeadStats("user1", "user2");

    EXPECT_EQ(user1Wins, 0);
    EXPECT_EQ(user2Wins, 0);
    EXPECT_EQ(ties, 0);
}

// Test 14: Save game with tie result
TEST_F(UserSystemTest, SaveGameWithTieResult) {
    std::vector<std::pair<int, std::string>> moves = {{0, ""}, {1, ""}, {2, ""}};

    bool saveResult = userSystem->saveGameWithMoves("player1", "player2", "Tie", moves);
    EXPECT_TRUE(saveResult);

    auto history = userSystem->getGameHistory("player1");
    ASSERT_FALSE(history.empty());
    EXPECT_EQ(std::get<3>(history[0]), "Tie");
}

```

This test confirms that statistics functions handle empty datasets gracefully. When requesting head-to-head stats for players who haven't played any games together, it expects all counters (wins, losses, ties) to be zero. This prevents errors when querying statistics for non-existent game history.

### 1.3.3 Performance and concurrency tests

#### 1.3.3.1 Concurrent user registration

```

TEST_F(UserSystemTest, ConcurrentUserRegistration) {
    std::vector<std::thread> threads;

```

```

std::vector<bool> results(5);

// Simulate multiple threads trying to register the same username
for (int i = 0; i < 5; ++i) {
    threads.emplace_back([this, &results, i]() {
        results[i] = userSystem->registerUser("concurrent_user", "password");
    });
}

for (auto& thread : threads) {
    thread.join();
}

// Only one registration should succeed due to UNIQUE constraint
int successCount = std::count(results.begin(), results.end(), true);
EXPECT_EQ(successCount, 1);
}

```

This test validates thread safety during simultaneous user registration attempts. It spawns 5 concurrent threads all trying to register the same username and expects only one to succeed due to database uniqueness constraints. This ensures the system handles race conditions properly and maintains data integrity under concurrent access.

#### 1.3.3.2 *Large dataset handling*

```

TEST_F(UserSystemTest, LargeDatasetHandling) {
    const int NUM_USERS = 50;
    const int GAMES_PER_USER = 3;

    // Register many users
    for (int i = 0; i < NUM_USERS; ++i) {
        std::string username = "user" + std::to_string(i);
        EXPECT_TRUE(userSystem->registerUser(username, "password"));
    }

    // Create many games
    std::vector<std::pair<int, std::string>> moves = {{0, "move1"}, {1, "move2"}};
    for (int i = 0; i < NUM_USERS; i += 2) {
        for (int j = 0; j < GAMES_PER_USER; ++j) {
            std::string player1 = "user" + std::to_string(i);
            std::string player2 = "user" + std::to_string(i + 1);
            EXPECT_TRUE(userSystem->saveGameWithMoves(player1, player2, "X", moves));
        }
    }

    // Verify data retrieval
    auto history = userSystem->getGameHistory("user0");
    EXPECT_EQ(history.size(), GAMES_PER_USER);
}

```

```
}
```

This test verifies the system's performance and reliability with substantial amounts of data. It registers 50 users and creates multiple games for each, then confirms all data can be stored and retrieved correctly. This ensures the system scales appropriately and maintains functionality under realistic usage loads.

#### 1.3.3.3 *Complex statistics scenario*

```
TEST_F(UserSystemTest, ComplexStatisticsScenario) {
    std::vector<std::pair<int, std::string>> moves = {{0, ""}};

    // Create complex game history for both head-to-head and AI stats
    userSystem->saveGameWithMoves("playerA", "playerB", "X", moves); // A wins
    userSystem->saveGameWithMoves("playerB", "playerA", "X", moves); // B wins
    userSystem->saveGameWithMoves("playerA", "playerB", "Tie", moves); // Tie
    userSystem->saveGameWithMoves("playerA", "AI", "X", moves); // A beats AI
    userSystem->saveGameWithMoves("AI", "playerA", "X", moves); // AI beats A

    // Test head-to-head stats
    int aWins, bWins, ties;
    std::tie(aWins, bWins, ties) = userSystem->getHeadToHeadStats("playerA", "playerB");
    EXPECT_EQ(aWins, 1);
    EXPECT_EQ(bWins, 1);
    EXPECT_EQ(ties, 1);

    // Test AI stats
    int humanWins, aiWins, aiTies;
    std::tie(humanWins, aiWins, aiTies) = userSystem->getHumanVsAIStats("playerA");
    EXPECT_EQ(humanWins, 1);
    EXPECT_EQ(aiWins, 1);
    EXPECT_EQ(aiTies, 0);
}
```

This test validates that statistics calculations work correctly with mixed game types and overlapping player relationships. It creates games involving both human-vs-human and human-vs-AI scenarios, then verifies that each statistics function returns accurate results for its specific scope. This ensures different statistical views don't interfere with each other and remain accurate in complex scenarios.

## 1.4 GUI testing :

## 2 Integration testing :

We have done 23 tests on the game wrapper to make sure it works properly. These are some of the tests and their explanations :

### 2.1 Authentication Tests

#### 2.1.1 Register new user

```
TEST_F(GameWrapperTest, RegisterNewUser) {  
    EXPECT_TRUE(wrapper->Register_Wrapper(CreateUniqueUser(), "pass123"));  
}
```

Tests successful user registration by creating a new unique username and verifying that the registration process completes successfully with valid credentials.

#### 2.1.2 Login valid user

```
TEST_F(GameWrapperTest, LoginValidUser) {  
    string user = CreateUniqueUser();  
    wrapper->Register_Wrapper(user, "pass");  
    EXPECT_TRUE(wrapper->Login_Wrapper(user, "pass"));  
}
```

Verifies successful login functionality by registering a user and then confirming they can log in with correct credentials.

#### 2.1.3 Login invalid user

```
TEST_F(GameWrapperTest, LoginInvalidUser) {  
    EXPECT_FALSE(wrapper->Login_Wrapper("nonexistent", "wrong"));  
}
```

Ensures the system properly rejects login attempts with non-existent usernames or incorrect passwords.

### 2.2 Occupied position move

```
TEST_F(GameWrapperTest, OccupiedPositionMove) {  
    wrapper->StartNewGame("P1", "P2", false, "", "X");  
    wrapper->MakeHumanMove(1);  
    EXPECT_FALSE(wrapper->MakeHumanMove(1));  
}
```

Tests that players cannot make moves to positions already occupied on the game board, ensuring move validation works correctly.

## 2.3 Win Detection Tests

### 2.3.1 horizontal win

```
TEST_F(GameWrapperTest, HorizontalWin) {
    wrapper->StartNewGame("P1", "P2", false, "", "X");
    wrapper->MakeHumanMove(1); // X
    wrapper->MakeHumanMove(4); // O
    wrapper->MakeHumanMove(2); // X
    wrapper->MakeHumanMove(5); // O
    wrapper->MakeHumanMove(3); // X wins

    auto winner = wrapper->CheckWinner();
    EXPECT_TRUE(get<0>(winner));
    EXPECT_EQ(get<1>(winner), "X");
}
```

Tests the game's ability to detect a horizontal three-in-a-row win condition by simulating moves that create a winning pattern across the top row.

### 2.3.2 diagonal win

```
TEST_F(GameWrapperTest, DiagonalWin) {
    wrapper->StartNewGame("P1", "P2", false, "", "X");
    wrapper->MakeHumanMove(1); // X
    wrapper->MakeHumanMove(2); // O
    wrapper->MakeHumanMove(5); // X
    wrapper->MakeHumanMove(3); // O
    wrapper->MakeHumanMove(9); // X wins

    auto winner = wrapper->CheckWinner();
    EXPECT_TRUE(get<0>(winner));
    EXPECT_EQ(get<1>(winner), "X");
}
```

Validates diagonal win detection by creating a winning pattern from top-left to bottom-right (positions 1, 5, 9).

## 2.4 Tie game

```
TEST_F(GameWrapperTest, TieGame) {
    wrapper->StartNewGame("P1", "P2", false, "", "X");
    // Create tie: X O X / O X X / O X O
    wrapper->MakeHumanMove(1); wrapper->MakeHumanMove(2); wrapper->MakeHumanMove(3);
    wrapper->MakeHumanMove(4); wrapper->MakeHumanMove(5); wrapper->MakeHumanMove(7);
    wrapper->MakeHumanMove(6); wrapper->MakeHumanMove(9); wrapper->MakeHumanMove(8);

    EXPECT_TRUE(wrapper->CheckTie());
}
```

Tests tie game detection by filling the board in a specific pattern that results in no winner.

## 2.5 Undo move

```
TEST_F(GameWrapperTest, UndoMove) {
    wrapper->StartNewGame("P1", "P2", false, "", "X");
    wrapper->MakeHumanMove(5);

    wrapper->Undo();
    auto board = wrapper->GetBoard();
    EXPECT_EQ(board[1][1], ' ');
    EXPECT_EQ(wrapper->getCurrentUserSymbol(), 'X');
}
```

Verifies the undo functionality works correctly by making a move, undoing it, and checking that the board position is cleared and the current player reverts.

## 2.6 Database Tests

### 2.6.1 Get game history

```
TEST_F(GameWrapperTest, GetGameHistory) {
    string user = CreateUniqueUser();
    wrapper->Register_Wrapper(user, "pass");

    vector<pair<int, string>> moves = {{1, "move"}};
    wrapper->SaveGameWithMoves(user, "opponent", "X", moves);

    auto history = wrapper->GetGameHistory(user);
    EXPECT_GE(history.size(), 1);
}
```

Tests the system's ability to save and retrieve a player's game history after completing games.

### 2.6.2 Head to head stats

```
TEST_F(GameWrapperTest, HeadToHeadStats) {
    string user1 = CreateUniqueUser();
    string user2 = CreateUniqueUser();
    wrapper->Register_Wrapper(user1, "pass");
    wrapper->Register_Wrapper(user2, "pass");

    vector<pair<int, string>> moves = {{1, "move"}};
    wrapper->SaveGameWithMoves(user1, user2, "X", moves);
    wrapper->SaveGameWithMoves(user1, user2, "O", moves);
    wrapper->SaveGameWithMoves(user1, user2, "Tie", moves);

    auto stats = wrapper->GetHeadToHeadStats(user1, user2);
    EXPECT_EQ(get<0>(stats), 1); // user1 wins
    EXPECT_EQ(get<1>(stats), 1); // user2 wins
    EXPECT_EQ(get<2>(stats), 1); // ties
}
```

Validates head-to-head statistics tracking between two players by simulating multiple games with different outcomes and verifying the win/loss/tie counts.

### 2.6.3 Human vs AI stats

```
TEST_F(GameWrapperTest, HumanVsAIStats) {  
    string user = CreateUniqueUser();  
    wrapper->Register_Wrapper(user, "pass");  
  
    vector<pair<int, string>> moves = {{1, "move"}};  
    wrapper->SaveGameWithMoves(user, "AI", "X", moves);  
    wrapper->SaveGameWithMoves(user, "AI", "O", moves);  
    wrapper->SaveGameWithMoves(user, "AI", "Tie", moves);  
  
    auto stats = wrapper->GetHumanVsAIStats(user);  
    EXPECT_EQ(get<0>(stats), 1); // human wins  
    EXPECT_EQ(get<1>(stats), 1); // AI wins  
    EXPECT_EQ(get<2>(stats), 1); // ties  
}
```

Tests statistics tracking for human vs AI games, ensuring proper categorization and counting of game outcomes against computer opponents.

### 2.7 Player symbol alternation

```
TEST_F(GameWrapperTest, PlayerSymbolAlternation) {  
    wrapper->StartNewGame("P1", "P2", false, "", "X");  
    EXPECT_EQ(wrapper->getCurrentUserSymbol(), 'X');  
  
    wrapper->MakeHumanMove(1);  
    EXPECT_EQ(wrapper->getCurrentUserSymbol(), 'O');  
  
    wrapper->MakeHumanMove(2);  
    EXPECT_EQ(wrapper->getCurrentUserSymbol(), 'X');  
}
```

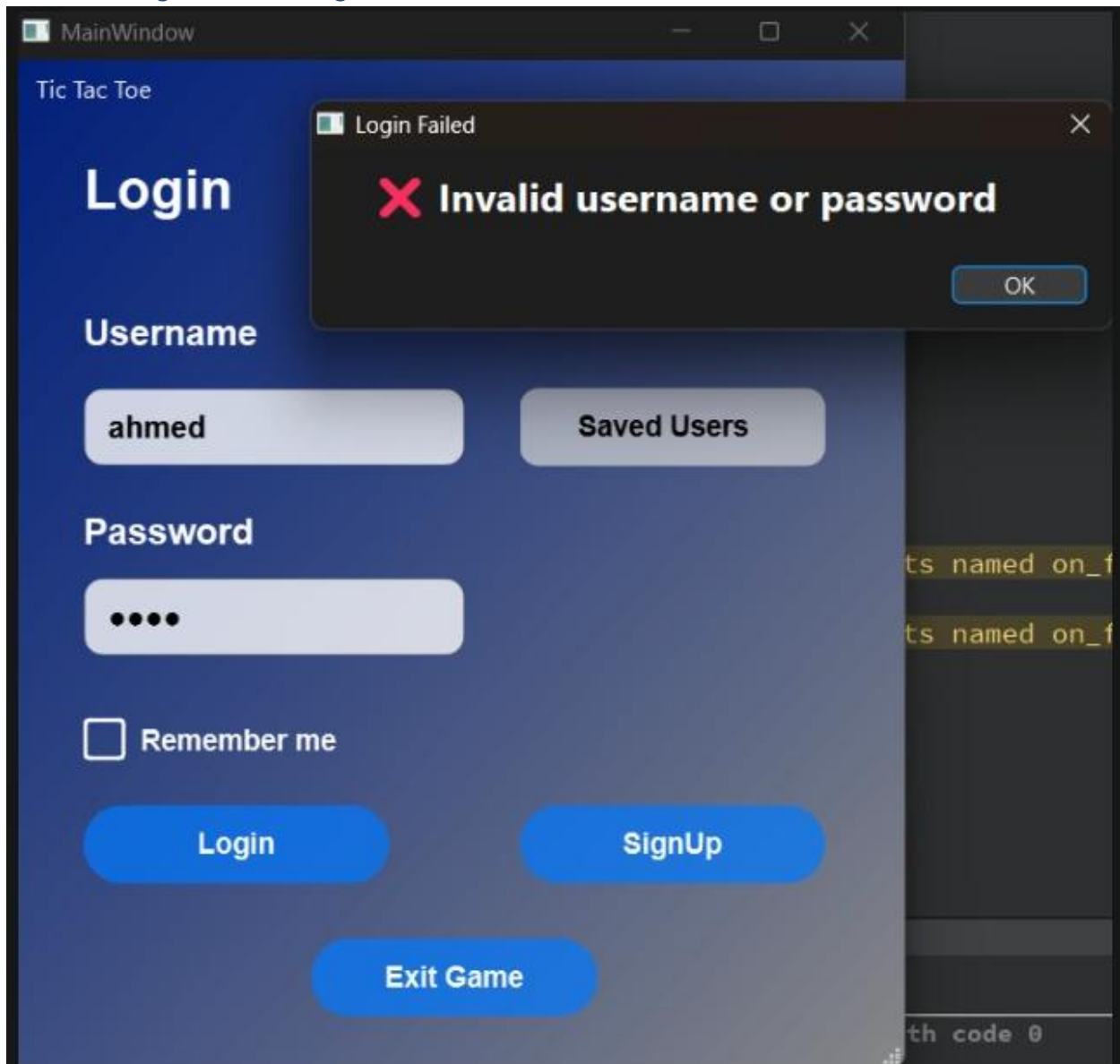
Verifies that player symbols (X and O) alternate properly between turns during gameplay.



### 3 GUI testing

#### 3.1 Login Page

##### 3.1.1 Unregistered user login test

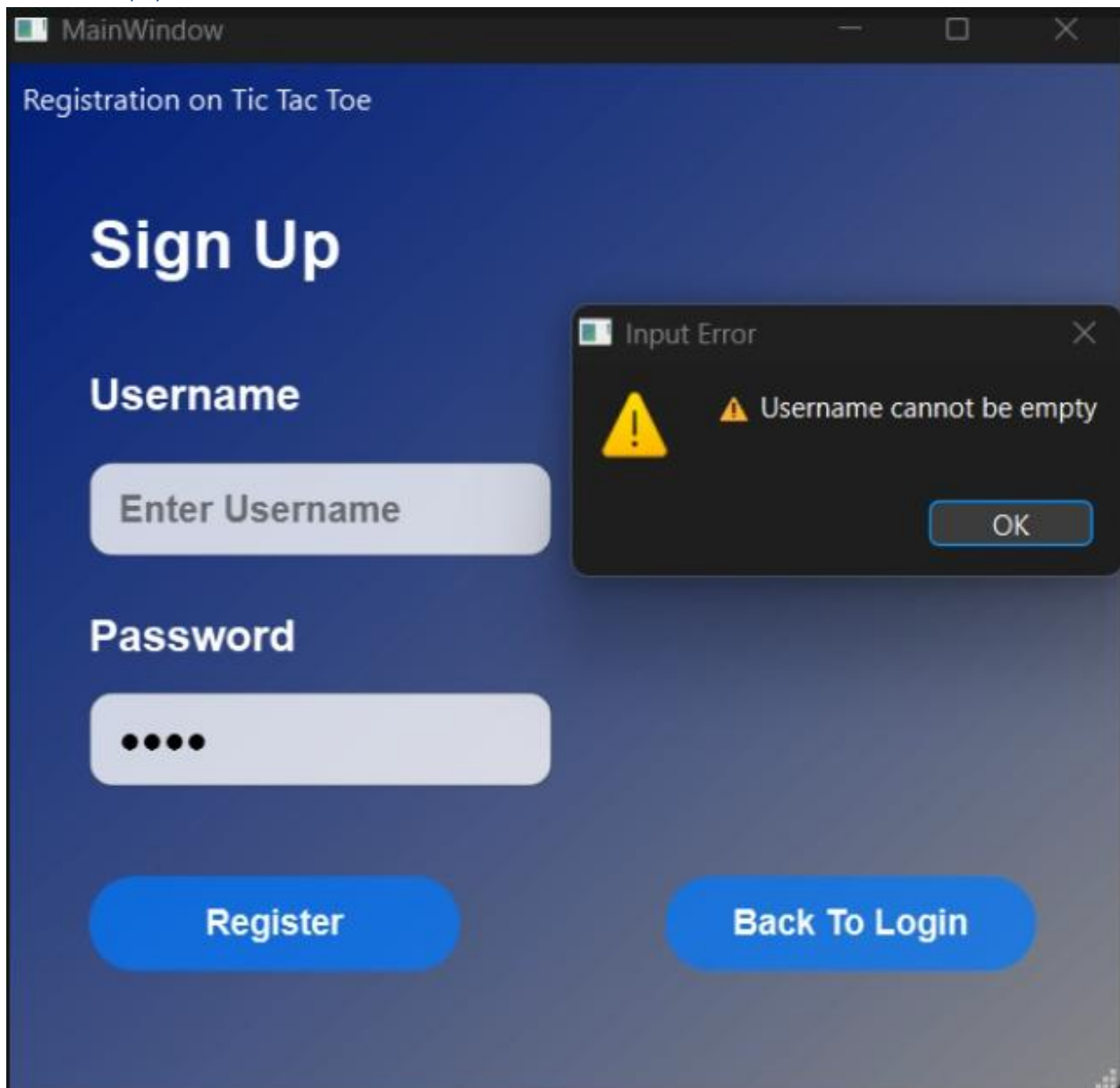


### 3.1.2 correct username and wrong password test

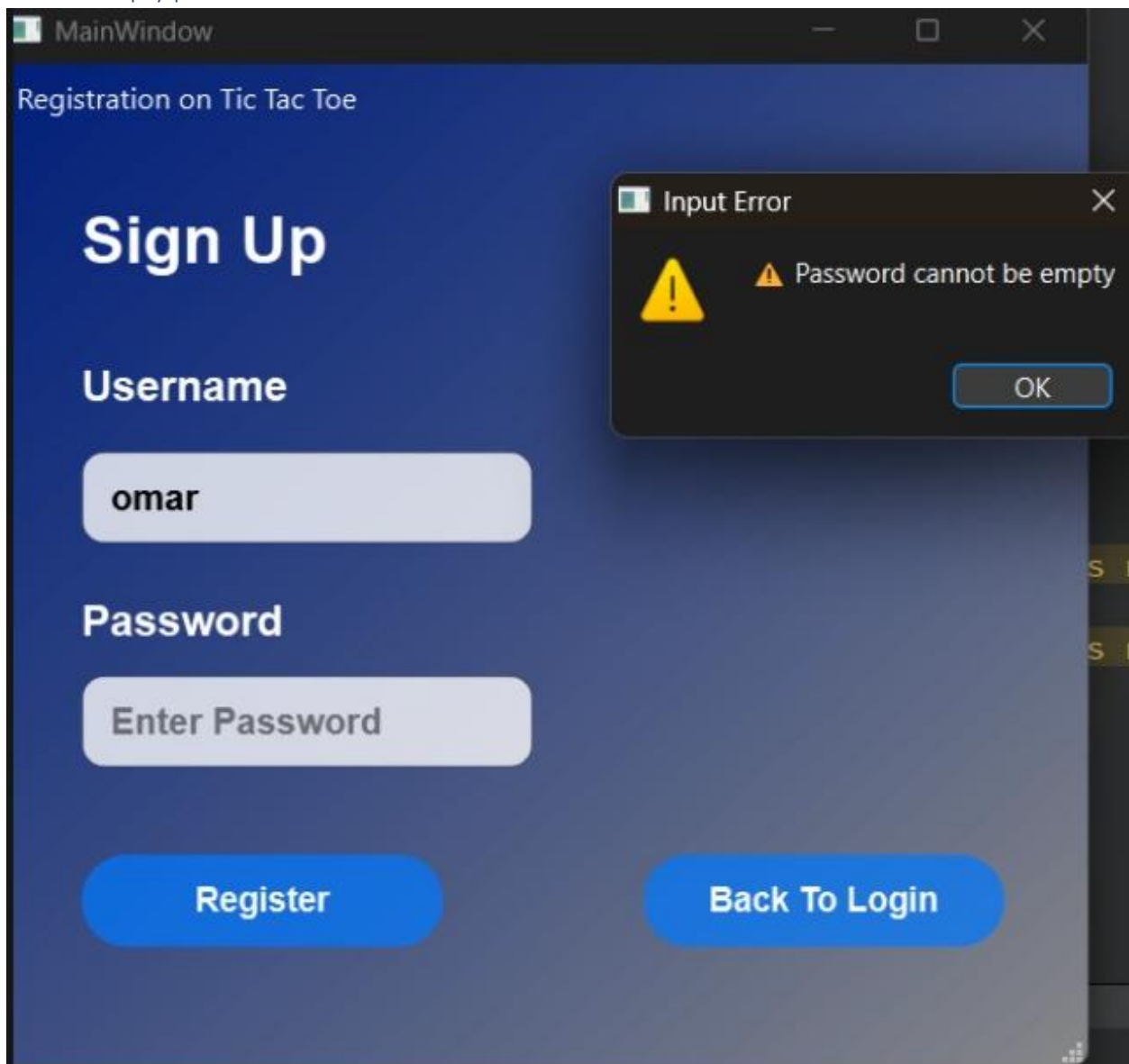


### 3.2 Sign up page

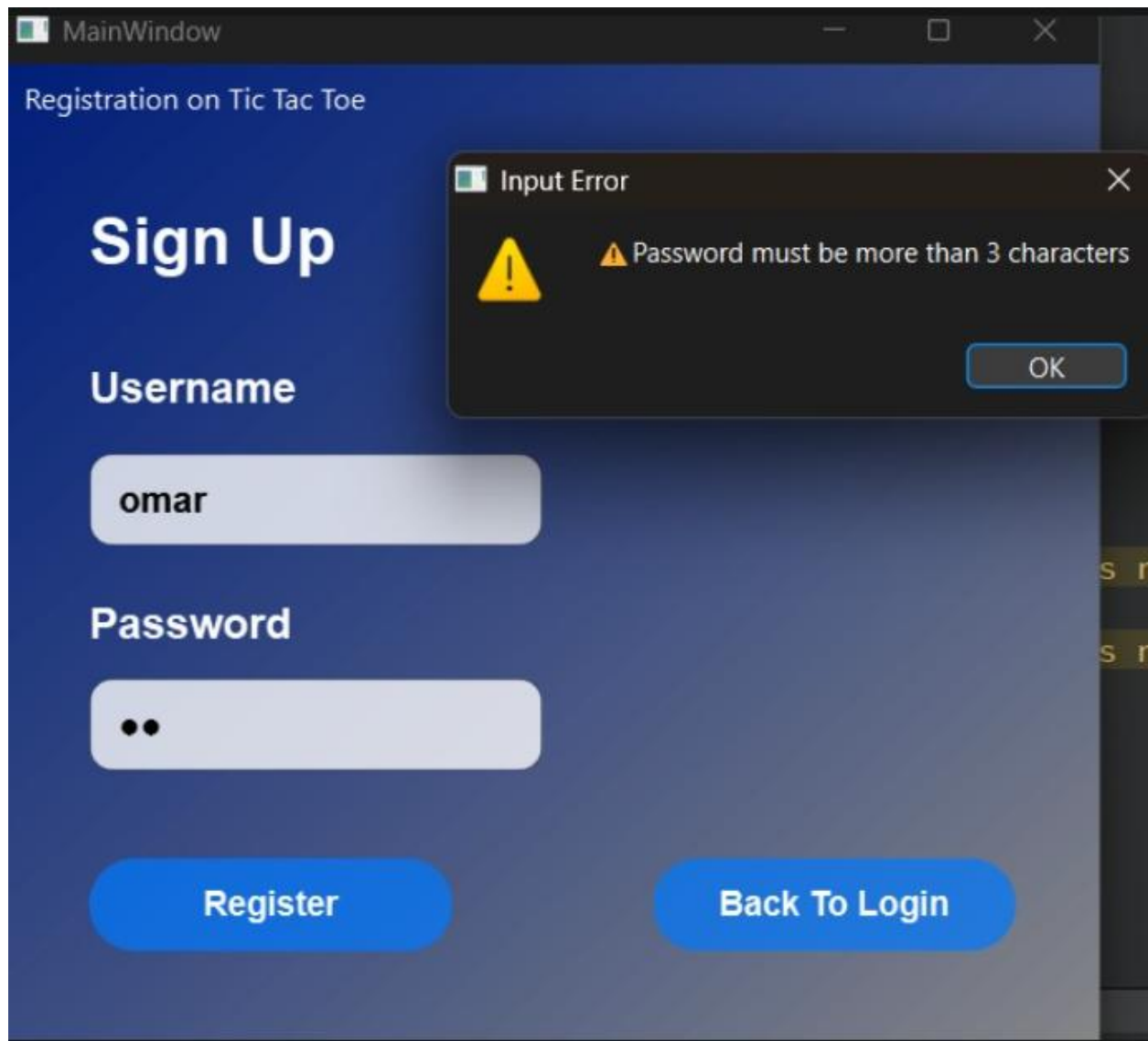
#### 3.2.1 empty username test



### 3.2.2 empty password test

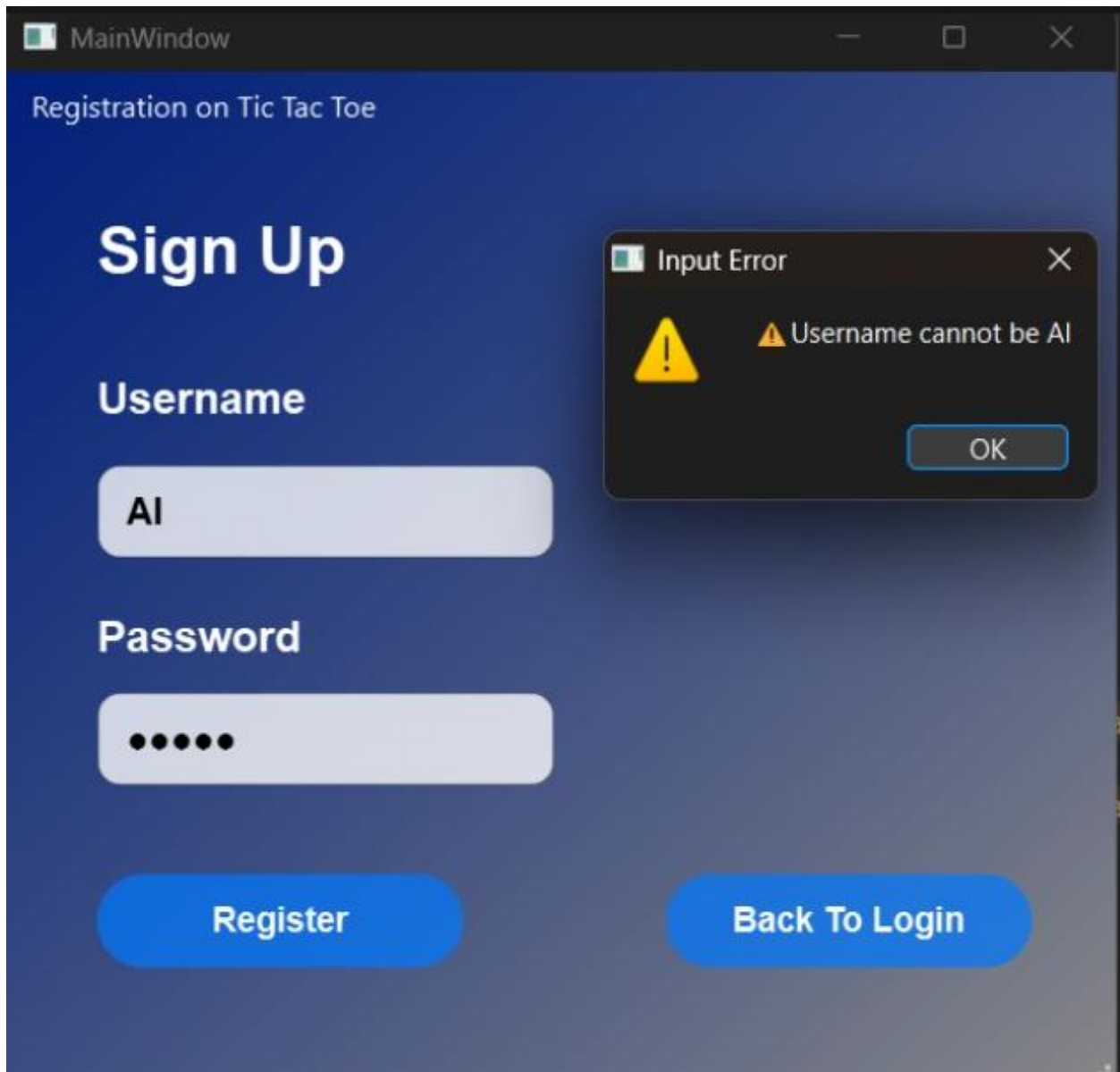


### 3.2.3 Password constraints test



Comment: Password must be more than 3 characters

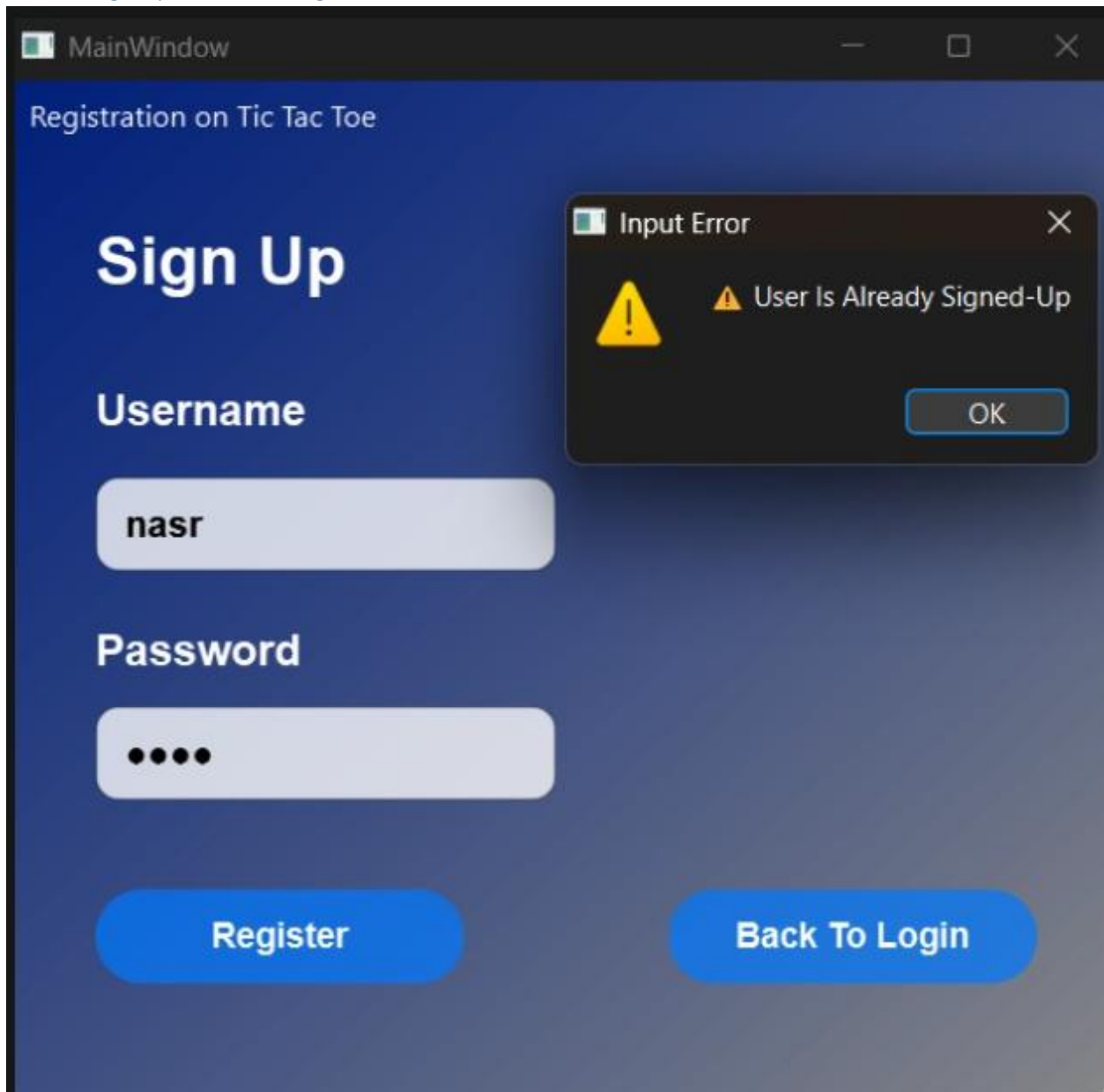
### 3.2.4 Username constraints test



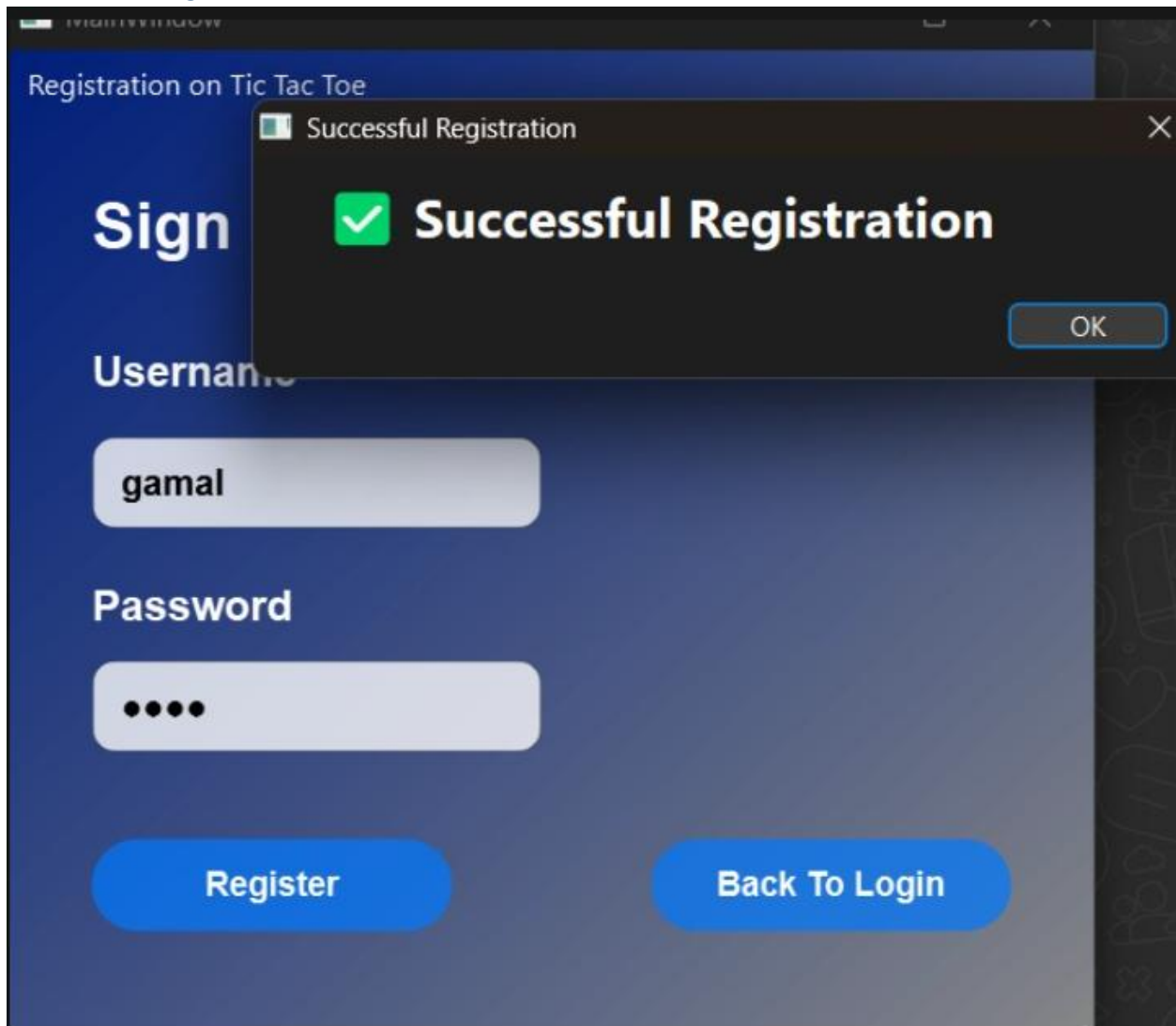
Comment:

- Username must be more than 3 characters
- Username can't be AI (or its combinations)

### 3.2.5 sign up same user again test



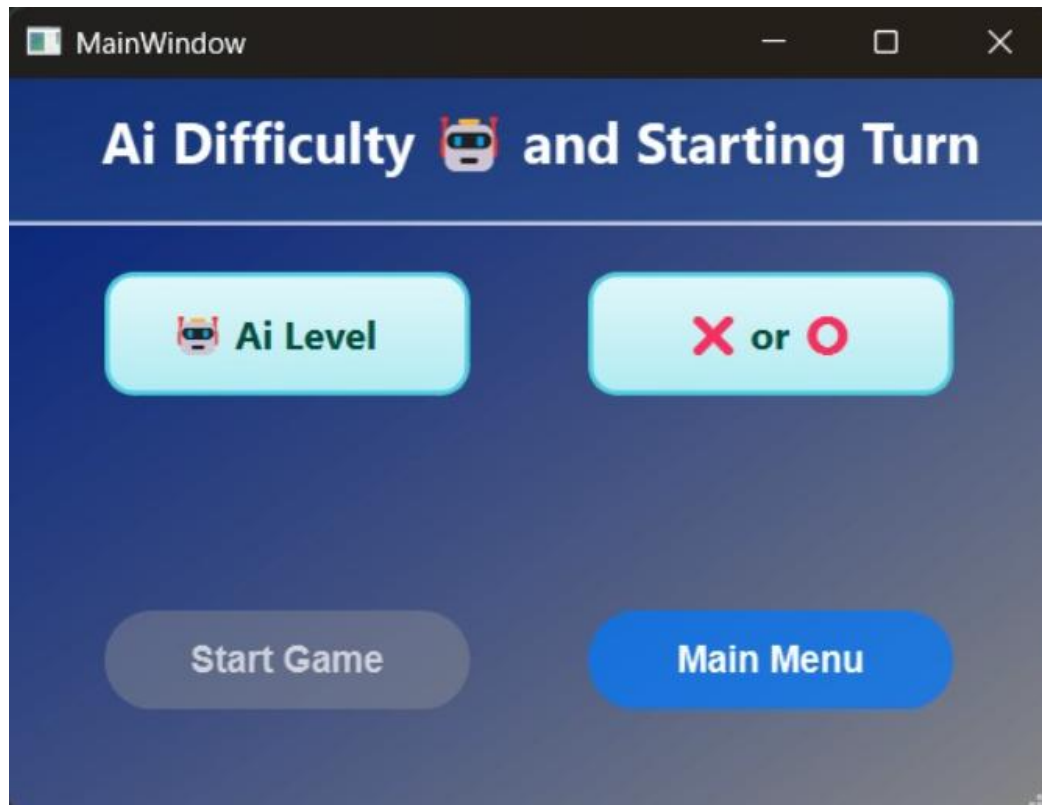
### 3.2.6 True registration test





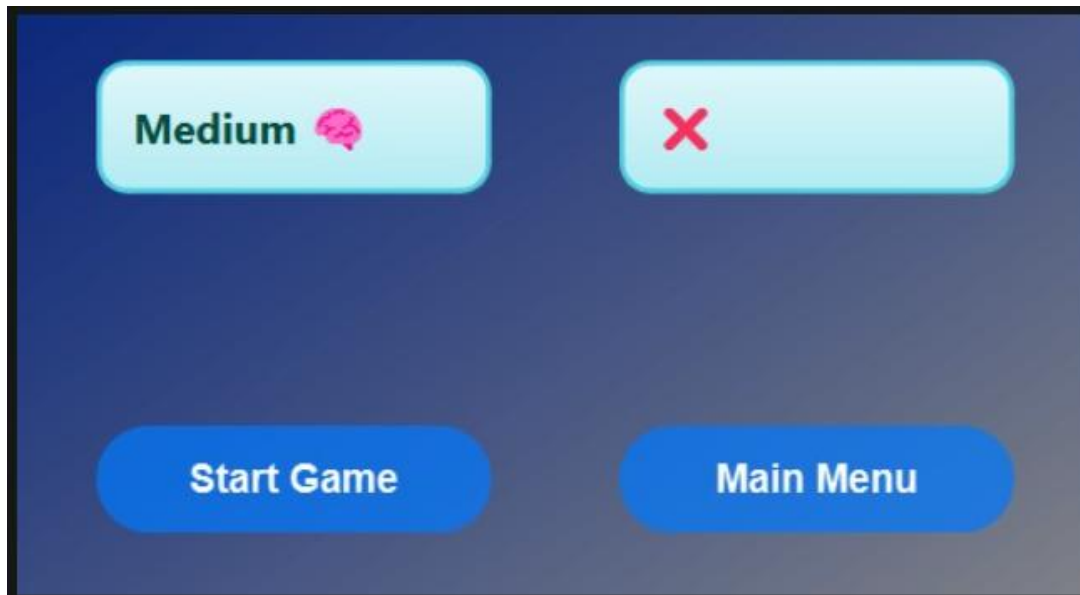
### 3.3 AI difficulty and starting turn page

#### 3.3.1



Comment: we can't start until we select the difficulty and starting turn

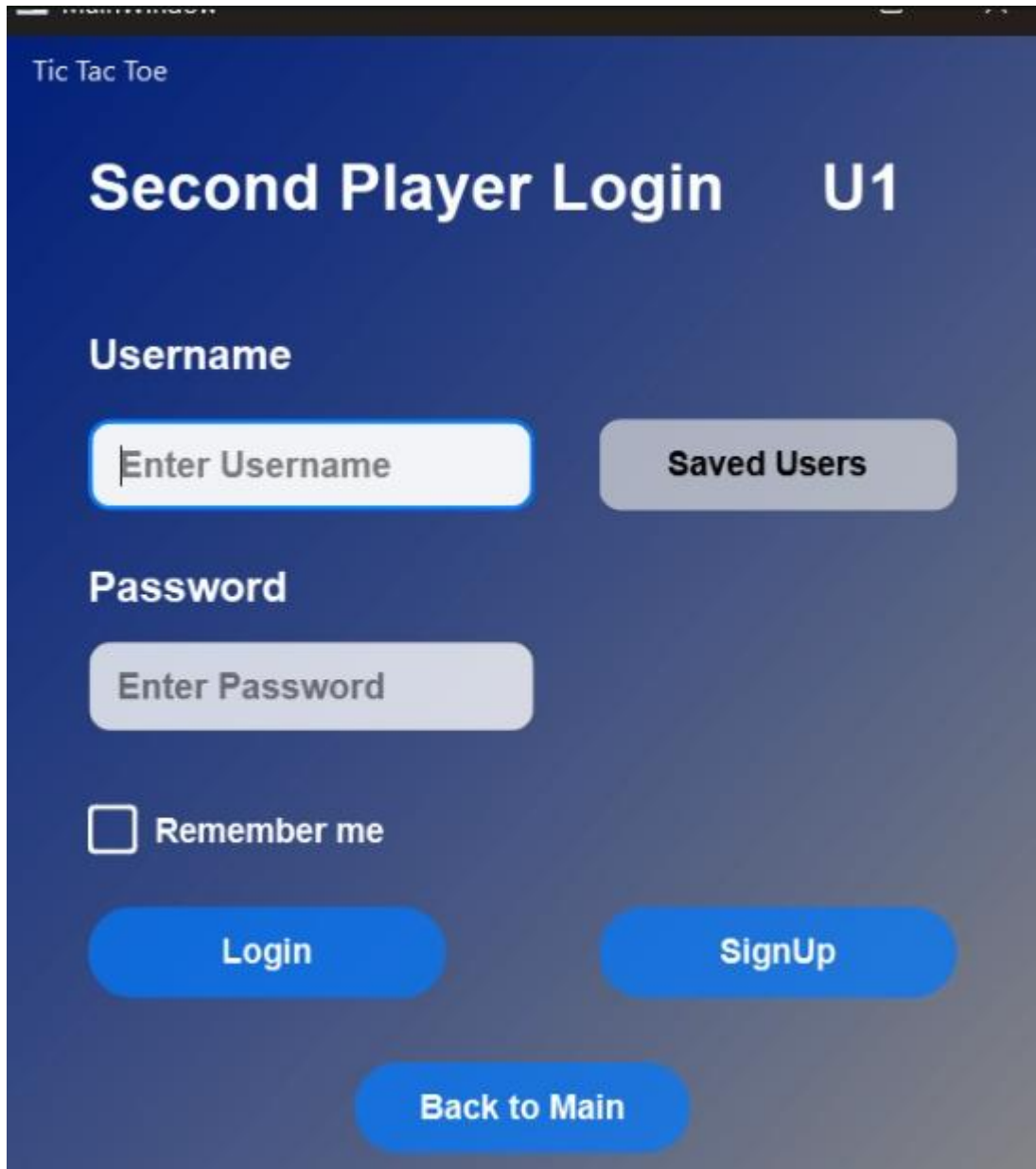
3.3.2



Comment: Start Game button is enabled when we selected the difficulty and starting turn

### 3.4 Second Player Login page

#### 3.4.1



The screenshot shows a mobile application interface for 'Tic Tac Toe'. The title 'Tic Tac Toe' is in the top left corner. The main heading is 'Second Player Login' followed by 'U1'. Below this, there are three input sections: 'Username' with a text field containing 'Enter Username' and a 'Saved Users' button; 'Password' with a text field containing 'Enter Password'; and a 'Remember me' checkbox. At the bottom, there are three blue buttons: 'Login', 'SignUp', and 'Back to Main'.

Tic Tac Toe

## Second Player Login U1

**Username**

Enter Username

Saved Users

**Password**

Enter Password

☐ Remember me

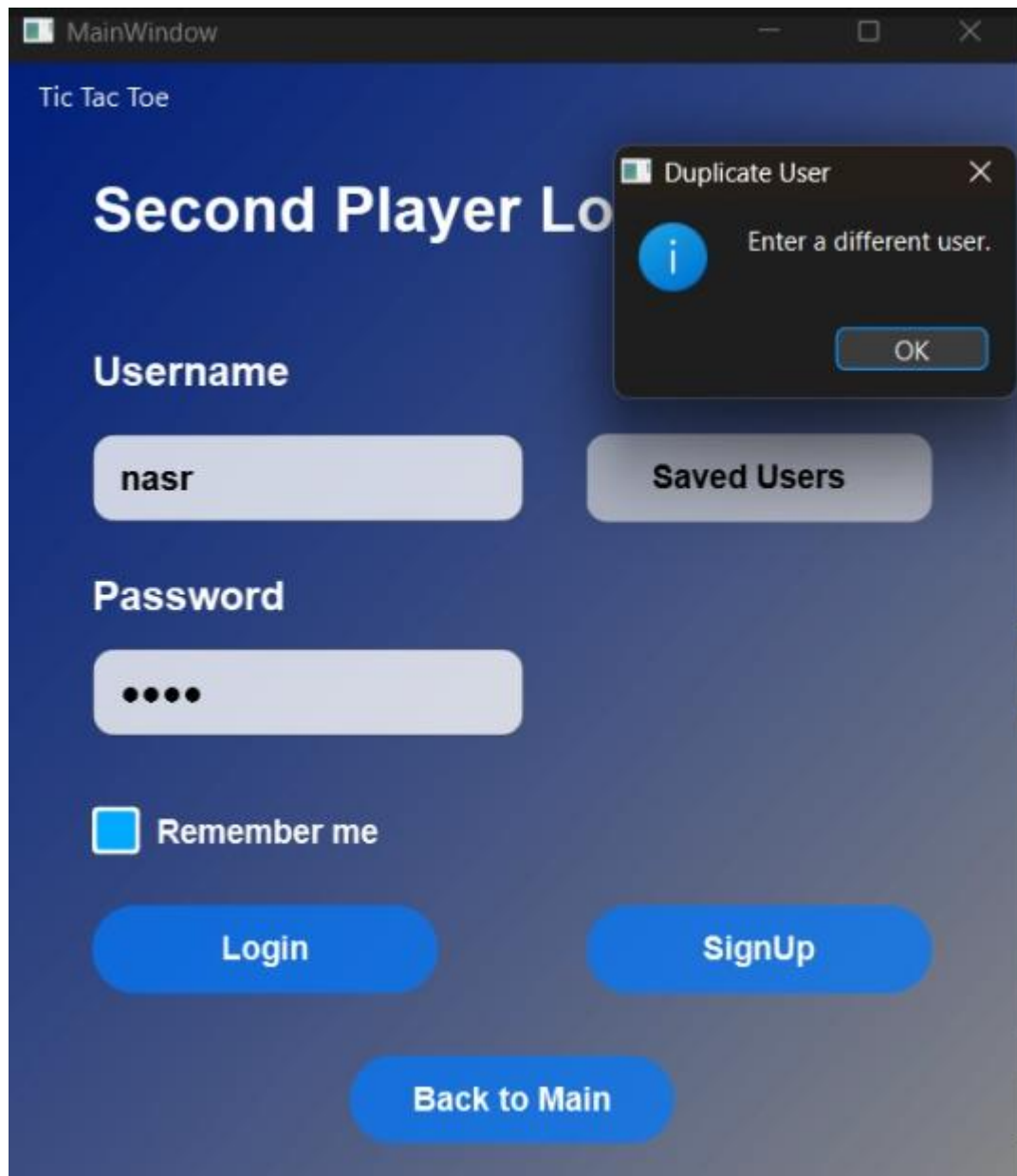
Login

SignUp

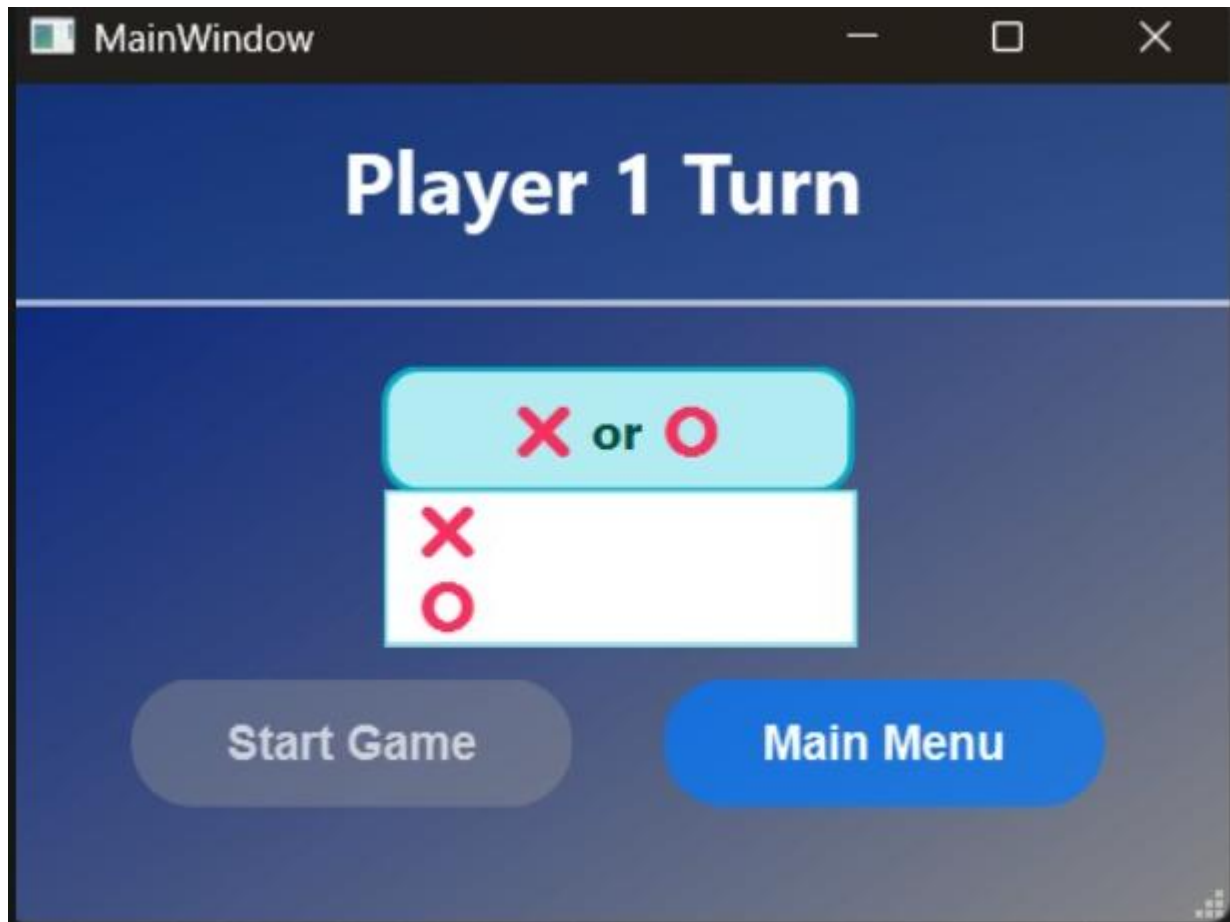
Back to Main

Comment: At choosing PvP, 2<sup>nd</sup> player must be logged in

### 3.4.2 Duplicate user issue test



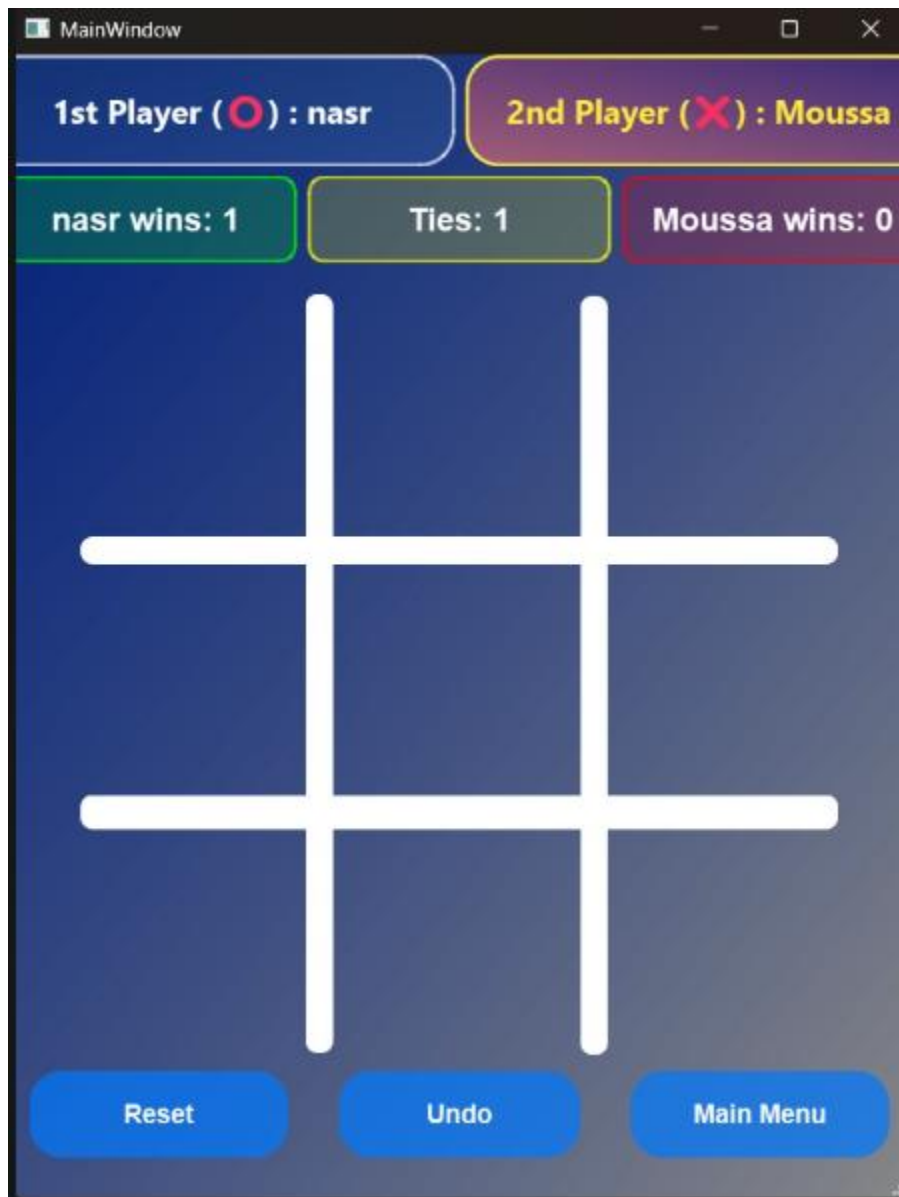
### 3.5 Player 1 Turn page



Comment: After signing in the 2<sup>nd</sup> player, we must choose a symbol to enable Start Game button

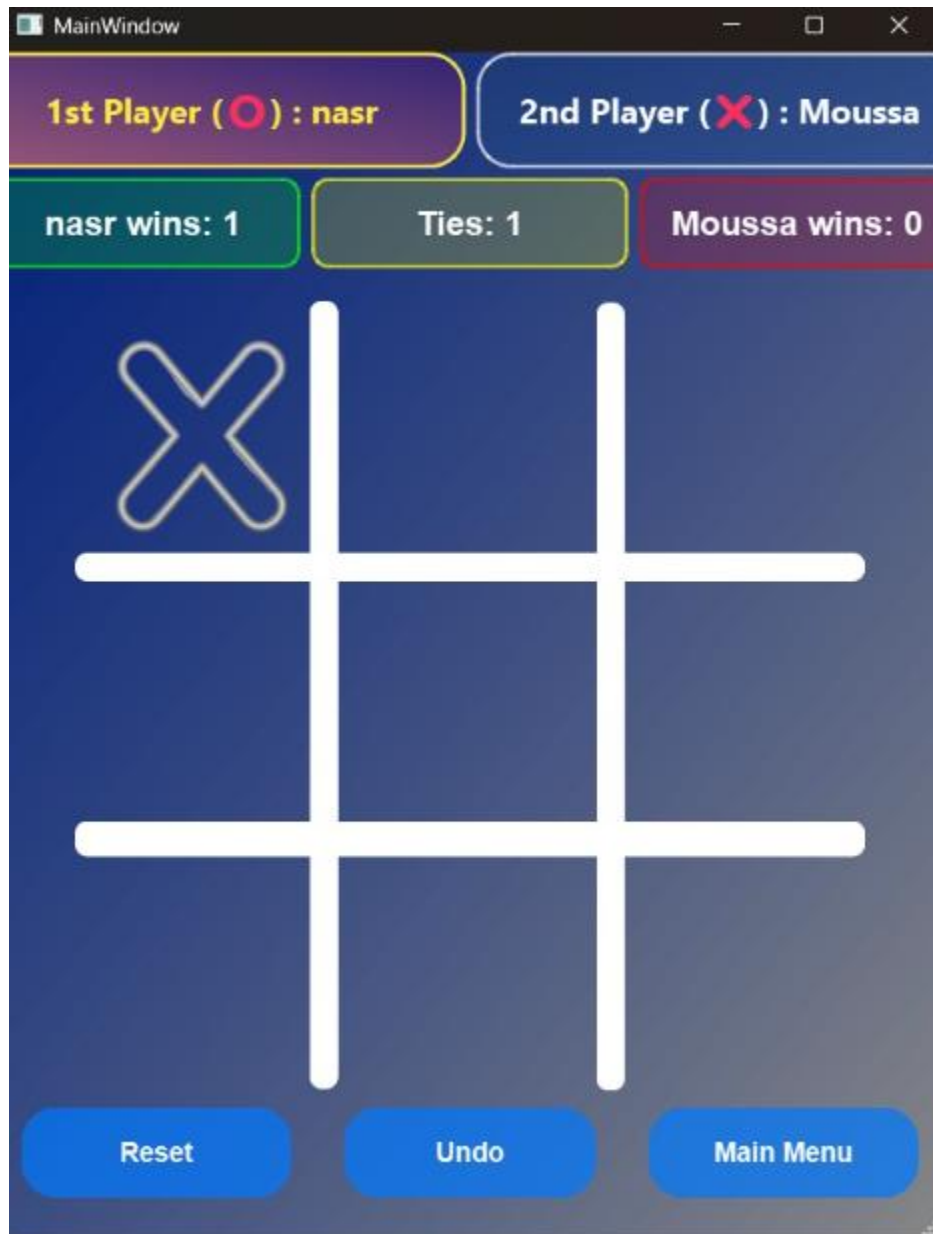
### 3.6 Game Board page

#### 3.6.1



Comment: In game board, The current player has pink neon color and the next has normal color

### 3.6.2 1<sup>st</sup> play verification test



### 3.6.3 Labels

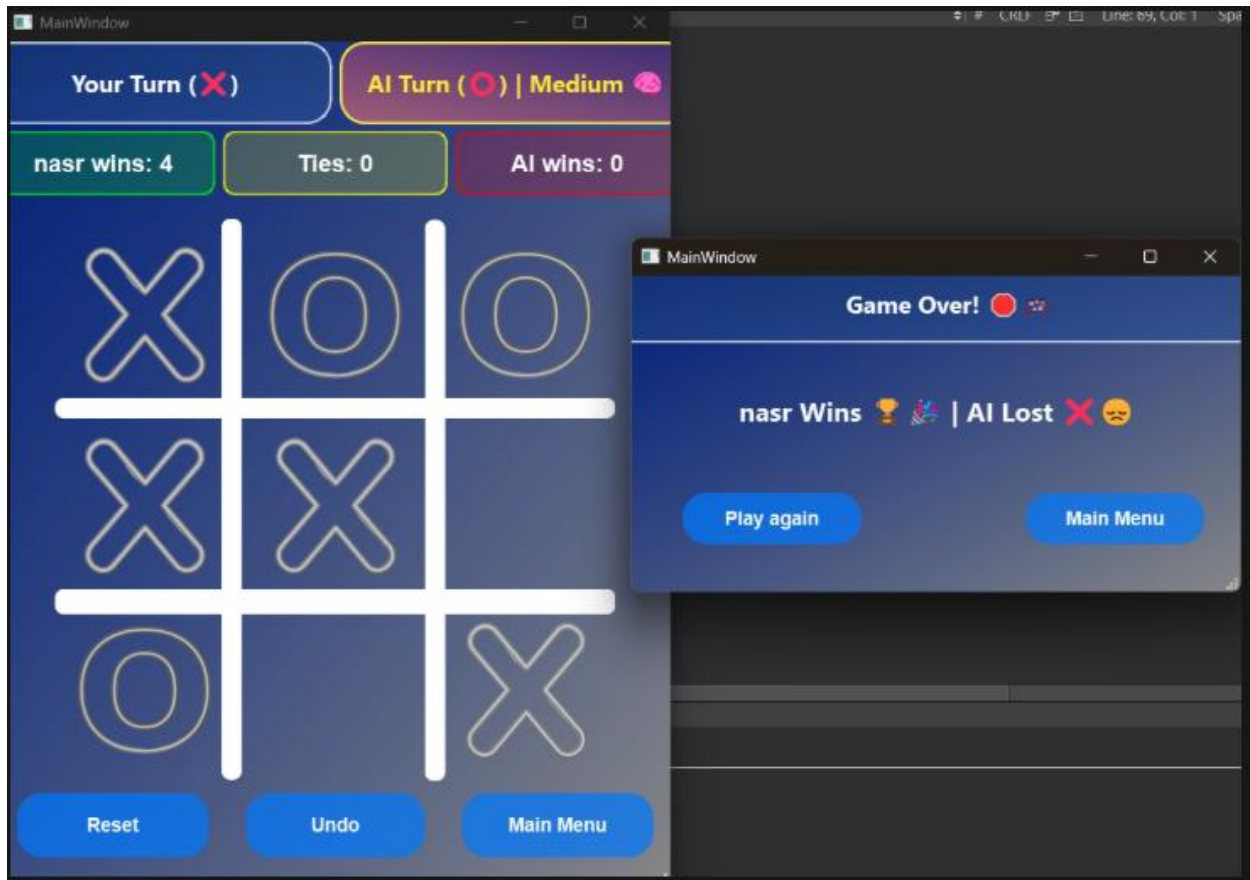


Comment:

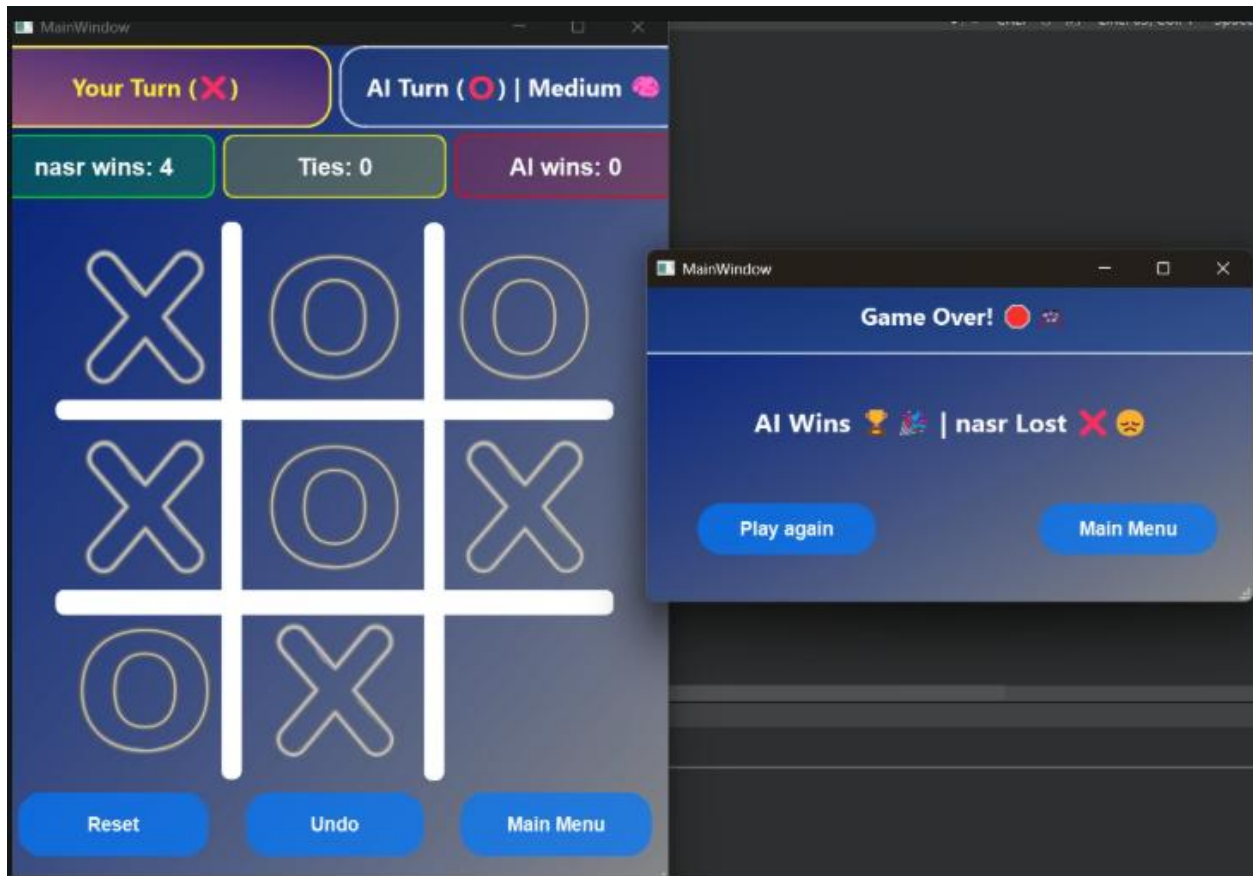
- These labels are to show accumulative game states between certain 2 players
- Green is for wins, Red is for Losses and ties is for yellow



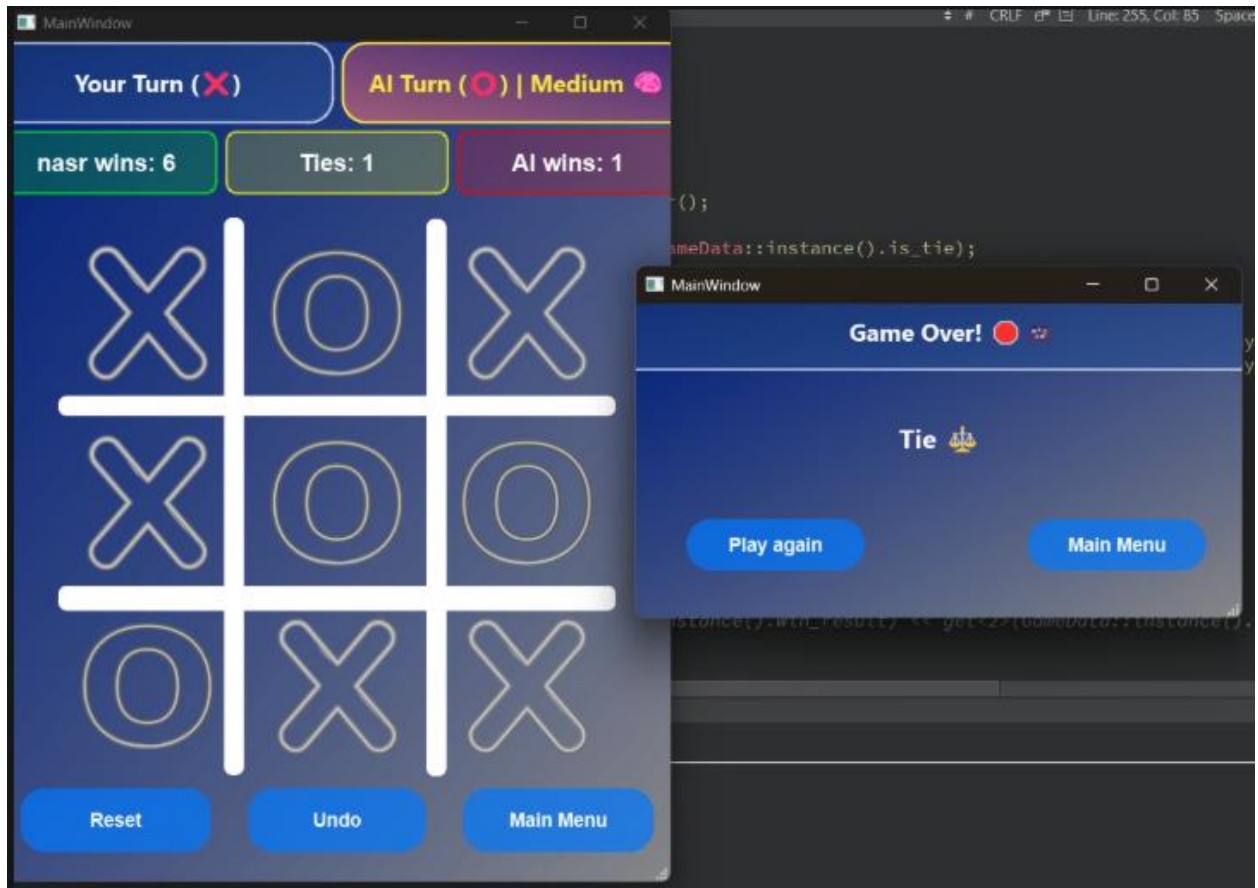
### 3.6.4 Check human win state test



### 3.6.5 Check human win state test

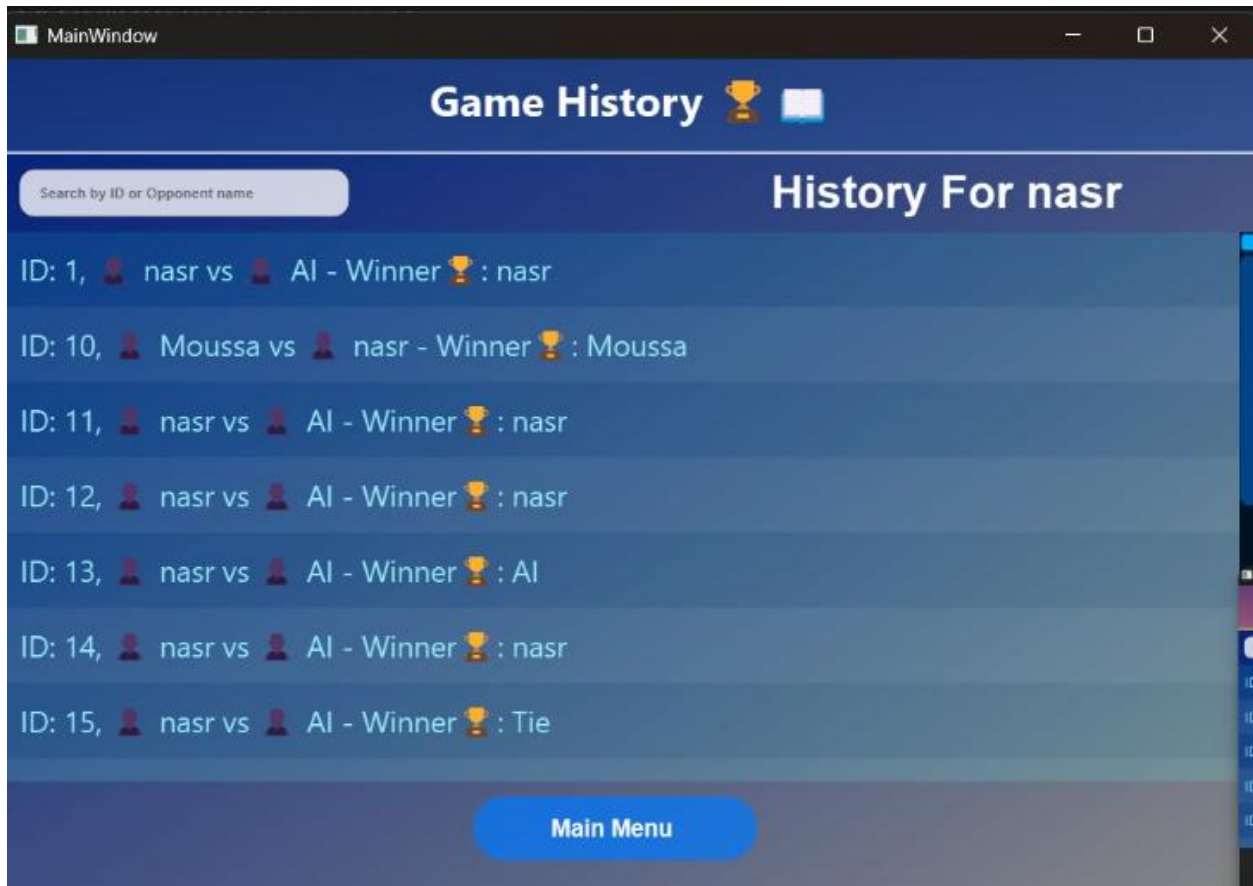


### 3.6.6 Check Tie state test

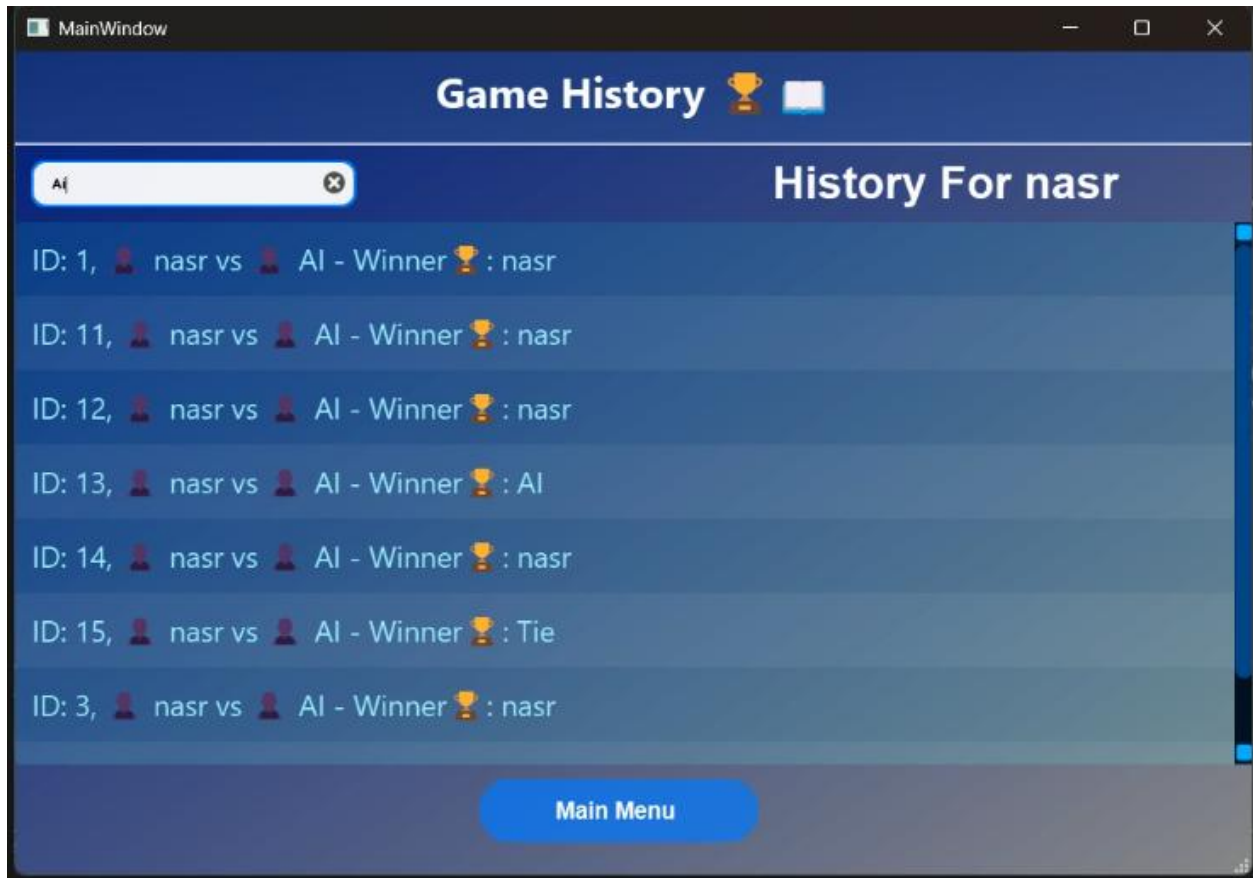


### 3.7 Game History page

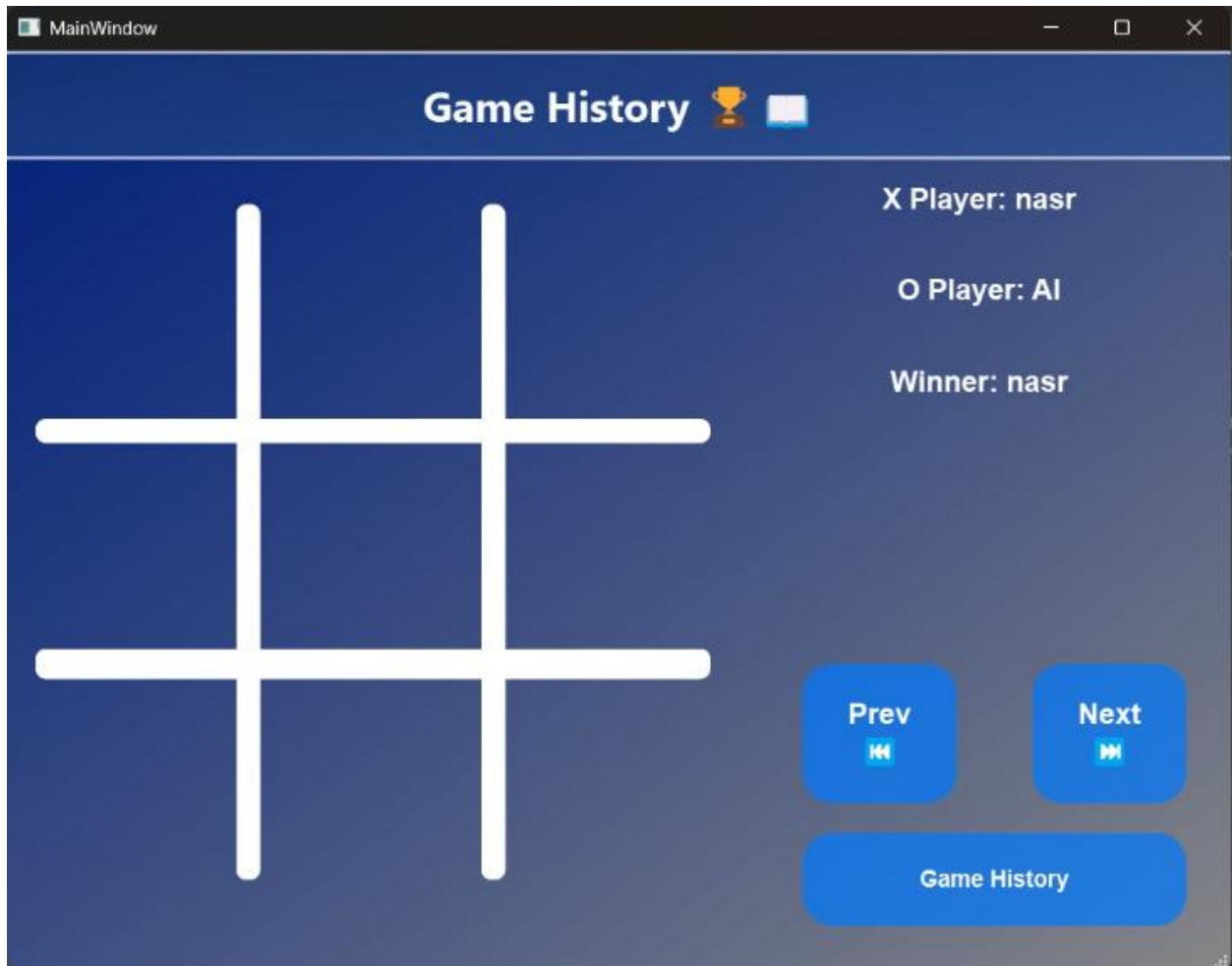
#### 3.7.1 Game history display test



### 3.7.2 Check search output test



### 3.7.3 Game reviewer display test



### 3.7.4 Next button test



Comment: 2 clicks were tested

### 3.7.5 Previous button test

