Project Name: Board Game Collection (Assignment 3)

Course Name: Programming II (CS213)

Professor Name: Dr. Mohammad El-Ramly

Section No: 16

Seif El-Din Khaled Ragab **20221077**

Eslam Ahmed Salah Abbas **20230056**

Ahmed Amr Ibrahim **20240036**

Eslam Ezzat Rasmy **20242042**

Technical Description & OOP Concepts

1. Class Structure Description : The project is built upon a robust, object-oriented architecture defined in `BoardGame_Classes.h`. This file contains the generic classes that serve as the engine for all games.

- `Board<T>` (Base Class): A generic abstract template class that represents any game board. It manages the grid dimensions (`rows`, `columns`) and the board data structure (2D vector). It defines the contract that all games must follow through pure virtual functions like `update_board()`, `is_win()`, `is_draw()`, and `game_is_over()`.
- `Player<T>`: Represents a player in the game. It stores the player's name, symbol (e.g., 'X', 'O'), and type (Human or Computer). It holds a pointer to the board to access the game state.
- `Move<T>`: Encapsulates the details of a single move, storing the coordinates (`x`, `y`) and the symbol to be placed.
- `GameManager<T>`: The controller class that manages the game flow. It initializes the board and players, and runs the main loop, switching turns and checking for win/draw conditions after each move.
- `UI<T>`: Handles user interaction, such as displaying the board and reading moves from players.
- Specific Game Classes: Classes like `X_O_Board`, `FourInARow_Board`, `SUS_Board`, and `Pyramid_Board` inherit from the base `Board` class. They implement the specific logic for their respective games, such as unique win conditions and move validation rules.

2. Application of OOP Principles (A PIE)

- Abstraction: We utilized abstraction by creating the `GameManager` class, which operates on the abstract `Board` pointer. The manager does not need to know the specific details of the game being played (e.g., whether it is Tic-Tac-Toe or Connect 4); it simply invokes abstract behaviors like `game_is_over()` or `update_board()`. This separates the high-level game flow from specific game rules.
- Polymorphism: Polymorphism is the core of our design, primarily achieved through virtual functions and pointers.
    - Runtime Polymorphism: The `GameManager` holds a pointer of type `Board<char>*`, but at runtime, this pointer can point to an object of `X_O_Board`, `SUS_Board`, or `FourInARow_Board`.
    - Virtual Functions: Functions like `is_win()` and `update_board()` are declared as `pure virtual` in the parent class. Each specific game provides its own implementation. For instance, `is_win()` in `Pyramid_Board` checks specific diagonal and horizontal lines tailored to its shape, while `FourInARow_Board` checks for 4 consecutive symbols.
- Inheritance: We applied inheritance to promote code reusability and hierarchical organization.
    - Code Reuse: All specific game boards (e.g., `Misere_XO_Board`, `WordTae_Board`) inherit from the generic `Board<T>` class. They inherit common attributes like `n_moves`, `rows`, `columns`, and the `board` matrix itself, eliminating the need to redefine them.
    - Extensibility: Adding a new game is straightforward; we simply derive a new class from `Board` and implement the required virtual functions.

- Encapsulation: We enforced encapsulation to protect the integrity of the game state.
    - Data Hiding: Critical data members in the `Board` class, such as the `board` 2D vector, are declared as `protected`. This ensures that they cannot be modified directly by external code (like `main.cpp`).
    - Controlled Access: External entities must interact with the board through public methods like `update_board()`. This method validates the move (checking boundaries and cell availability) before modifying the internal state, preventing invalid states.

Technical Descriptions for All Implemented Games

1. Games by Seif El-Din (ID: 20221077)

- Game 1: SUS Game
  - Class: SUS_Board (Inherits from Board<char>).
  - Key Logic: Implemented countSUSAtCell() to dynamically scan horizontal, vertical, and diagonal lines for the "S-U-S" pattern. The game tracks scores using scoreS and scoreU variables instead of immediate win conditions, declaring the winner only when the board is full.
- Game 2: Four-in-a-Row
  - Class: FourInARow_Board (Inherits from Board<char>).
  - Key Logic: Overrode update_board() to implement "gravity logic," ensuring moves always drop to the lowest available row in the selected column. Win checking (checkWinAt) scans for 4 consecutive symbols in all directions from the last placed piece.

2. Games by Eslam Ahmed (ID: 20230056)

- Game 3: 5x5 Tic Tac Toe
  - Class: FiveByFive_Board (Inherits from Board<char>).
  - Key Logic: Developed a count_triplets() function using a sliding window algorithm to count all occurrences of three-in-a-row patterns across the 5x5 grid. The winner is determined by comparing total triplet counts after 24 moves.
- Game 4: Word Tic-Tac-Toe
  - Class: WordTae_Board (Inherits from Board<char>).
  - Key Logic: Integrated std::set<string> for $O(1)$ dictionary lookups. The is_win() function constructs strings from board rows/cols/diagonals and validates them against the loaded dic.txt file.

3. Games by Ahmed Amr (ID: 20240036)

- Game 5: Misère Tic Tac Toe
  - Class: Misere_XO_Board (Inherits from Board<char>).
  - Key Logic: Inverted the standard winning condition. The is_lose() function detects if a player has formed 3-in-a-row, which triggers a loss state instead of a win.
- Game 6: Diamond Tic-Tac-Toe
  - Class: DiamondBoard (Inherits from Board<char>).
  - Key Logic: Used a mask (2D boolean vector) to define the valid diamond-shaped cells within the 5x5 grid. The update_board() function validates moves against this mask before placement.

4. Games by Eslam Ezzat (ID: 20242042)

- Game 7: 4x4 Tic-Tac-Toe
  - Class: TicTacToe4x4_Board (Inherits from Board<char>).
  - Key Logic: Implemented token movement logic where update_board() validates source and destination indices to ensure tokens only move to adjacent empty cells. Win checking accounts for the larger grid dimensions.
- Game 8: Pyramid Tic-Tac-Toe
  - Class: Pyramid_Board (Inherits from Board<char>).
  - Key Logic: Mapped the pyramid structure to a 2D array using specific valid indices. The is_win() function performs hardcoded checks for the specific horizontal, vertical, and diagonal lines possible in the pyramid geometry.

5. Group Games (Team Work)

- Game 9: Numerical Tic-Tac-Toe
  - Class: NumericalBoard (Inherits from Board<int>).
  - Key Logic: Used a std::vector<int> used_numbers to track numbers already played. The is_win() function checks if the sum of any row, column, or diagonal equals 15.
- Game 10: Obstacles Tic-Tac-Toe
  - Class: ObstaclesTTT_Board (Inherits from Board<char>).
  - Key Logic: Implemented add_obstacles() which uses a random number generator to place blocker cells ('#') on the board dynamically after each round, preventing players from using those spots.
- Game 11: Infinity Tic-Tac-Toe
  - Class: Infinity_Board (Inherits from Board<char>).
  - Key Logic: Utilized a std::queue<pair<int, int>> to track the history of moves. When the number of moves exceeds the limit (6 moves), the oldest move is dequeued and removed from the board, creating the "infinity" loop effect.

Work Breakdown & Task Distribution

Seif El-Din Khaled Ragab Abdel-Mohsen	**Game 1: SUS Game** **Game 2: Four-in-a-row**
Eslam Ahmed Salah Abbas	**Game 3: 5x5 Tic Tac Toe** **Game 4: Word Tic-Tac-Toe**
Ahmed Amr Ibrahim	**Game 5: Misère Tic Tac Toe** **Game 6: Diamond Tic-Tac-Toe**
Eslam Ezzat Rasmy	**Game 7: 4x4 Tic-Tac-Toe** **Game 8: Pyramid Tic-Tac-Toe**

Sample Screenshots of Gameplay

Main Menu:

```
Welcome to FCAI Board Games!
1. X-O
2. Infinity TicTacToe
3. SUS Game
4. Four-In-A-Row
5. Pyramid Tic-Tac-Toe
6. TicTacToe4x4
7. Misere Tic Tac Toe
8. Diamond Tic Tac Toe
9. Numerical Tic-Tac-Toe
10. 5x5 Tic Tac Toe
11. Word Tic-Tac-Toe
12. Obstacles Tic-Tac-Toe
```

Some screenshots of some games :

SUS Game

```
Please enter your move x and y (0 to 2): 1
0

      0   1   2
   ---------------
 0 |  S |    |    |
   ---------------
 1 |  S |    |    |
   ---------------
 2 |    |    |  U |
   ---------------


      0   1   2
   ---------------
 0 |  S |    |    |
   ---------------
 1 |  S |  U |    |
   ---------------
 2 |    |    |  U |
   ---------------

Please enter your move x and y (0 to 2):
```

Four-in-a-row

```
      0   1   2   3   4   5   6
   -----------------------------------
 0 |  .  |  .  |  .  |  .  |  .  |  .  |  .  |
   -----------------------------------
 1 |  .  |  .  |  .  |  .  |  .  |  .  |  .  |
   -----------------------------------
 2 |  .  |  .  |  .  |  .  |  .  |  .  |  .  |
   -----------------------------------
 3 |  .  |  .  |  .  |  .  |  .  |  .  |  .  |
   -----------------------------------
 4 |  .  |  .  |  .  |  .  |  X  |  .  |  .  |
   -----------------------------------
 5 |  .  |  O  |  O  |  X  |  X  |  O  |  .  |
   -----------------------------------

Please enter your move x and y (0 to 2):
```

Infinity TicTacToe

```
Please enter your move x and y (0 to 2): 2
0

      0   1   2
   ---------------
 0 |  .  |  X  |  .  |
   ---------------
 1 |  .  |  .  |  .  |
   ---------------
 2 |  X  |  .  |  O  |
   ---------------


      0   1   2
   ---------------
 0 |  .  |  X  |  .  |
   ---------------
 1 |  .  |  O  |  .  |
   ---------------
 2 |  X  |  .  |  O  |
   ---------------


Please enter your move x and y (0 to 2):
```

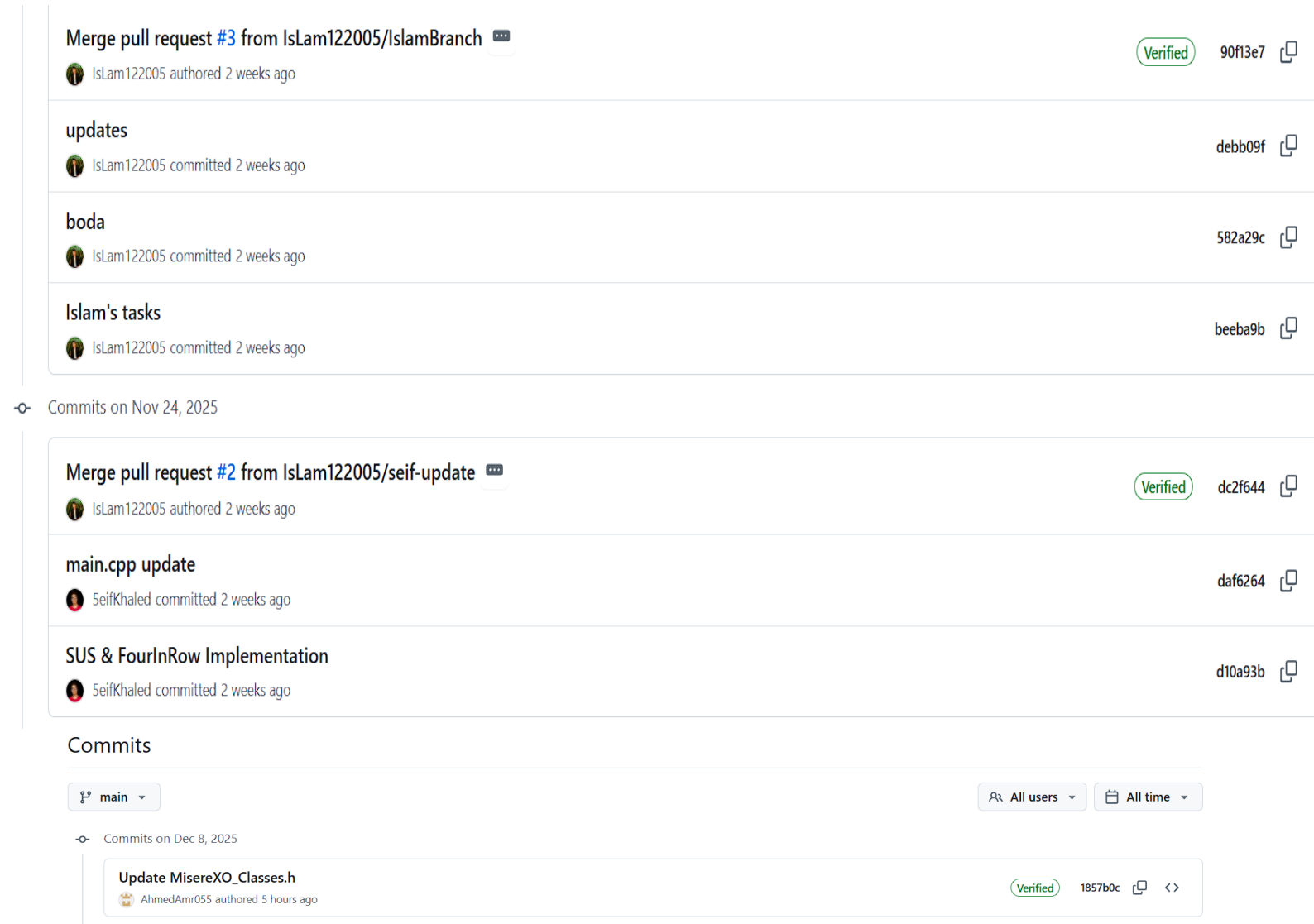
Pyramid Tic-Tac-Toe

```
Please enter your move x and y (0 to 2): 0
2

      0   1   2   3   4
   -------------------------
 0 |   |   |  X  |   |   |
   -------------------------
 1 |   |   |  X  |   |   |
   -------------------------
 2 |   |  X  |  O  |   |  O  |
   -------------------------


      0   1   2   3   4
   -------------------------
 0 |   |   |  X  |   |   |
   -------------------------
 1 |   |   |  X  |   |   |
   -------------------------
 2 |  O  |  X  |  O  |   |  O  |
   -------------------------


Please enter your move x and y (0 to 2):
```

GitHub Collaboration Evidence

To ensure effective collaboration and version control, our team utilized a private GitHub repository throughout the development process. This allowed us to work on different game modules simultaneously and merge our contributions seamlessly.

The screenshot below illustrates the project's commit history, verifying the active participation of all team members.



rebo link :https://github.com/IsLam122005/OOPGAMES

video link: https://www.youtube.com/watch?v=2vWKwNXVw9c

code review

Seif El-Din's Review of Ahmed Amr's Code

file	prof	cons
DiamondXO.h	- Clear class design for DiamondBoard and DiamondUI. - override used for polymorphism. - Doxygen comments are clear. - Simple and understandable AI logic. - Readable and maintainable code.	- Board size is hardcoded to 5×5. - srand(time(0)) inside function may cause repeated results. - No input validation for human players. - Dynamic allocation without smart pointers (risk of memory leaks). - UI tightly coupled to DiamondBoard.
DiamondXO.cpp	- Diamond shape represented clearly using mask. - Modular functions with single responsibilities. - has_k_in_row is flexible for different win lengths. - Well-formatted and readable code	- Board size fixed to 5×5. - is_win logic hardcoded for 3 and 4 in a row. - No handling for invalid move pointers. - has_k_in_row may be slower on larger boards.
MisereXO_Classes.h	- Clear separation between Board and UI. - Simple and reusable design. - Naming of classes and functions is clear. - Readable and maintainable.	- Board size fixed to 3×3 and blank symbol hardcoded. - No virtual destructor. - No validation of Move or Player. - Dynamic allocation without smart pointers.
MisereXO.cpp	- Clear and simple logic for update_board, is_lose, and is_draw. - Lambda all_equal reduces code repetition. - Clear distinction between human and AI moves. - Maintainable and easy to understand	- Board size fixed to 3×3. - No validation of human input. - Random AI uses rand() without seeding. - Dynamic allocation without smart pointers. - No error handling (e.g., nullptr player).

Ahmed Amr's Review of Seif El-Din's Code

File	Prof	Cons
SUS.h	- Clear class design (SUS_Move, SUS_Board). - Virtual destructors for safety. - Engine-compatible overrides defined. - Static play() function for interactive use. - Score tracking (scoreS, scoreU).	- Hardcoded board size 3×3. - Minimal comments for private functions. - No input validation here. - Dynamic allocation risk (Move pointers). - Tight coupling with engine interface.
SUS.cpp	- Clear game logic for placing symbols and counting sequences. - Engine-compatible functions implemented. - play() supports Human/Computer players. - Random AI uses proper mt19937 RNG. - ASCII art and colored output improve UX.	- Hardcoded 3×3 board. - Dynamic allocation without smart pointers. - Tight coupling with engine symbols (X/O → S/U). - countSUSAtCell logic is verbose, could be refactored. - Partial input validation for human moves.
FourInARow.h	- Clear class design (FourInARow_Move, FourInARow_Board). - Virtual destructors for safety. - Engine-compatible overrides defined. - Static play() for interactive use. - Tracks last move for efficient win checking.	- Hardcoded board size 6×7. - Tight coupling with engine symbols (X/O). - Dynamic allocation risk (Move pointers). - No input validation in header. - Some helper functions may duplicate logic.
FourInARow.cpp	- Clear game logic for drop mechanics and win checking. - Efficient win checking using last move. - Engine-compatible functions implemented. - Random AI with proper RNG. - Static play() supports Human/Computer players. - ASCII art and colored output enhance UX.	- Hardcoded 6×7 board. - Dynamic allocation without smart pointers. - Tight coupling with engine symbols. - Partial input validation for human moves. - Repeated display code (disp()) could be improved.

Islam Ezzat’s review about Islam Dawood’s code:

File	Prof	Cons
FourInRow_Board.cpp	- Logic for "gravity" (tokens dropping to the bottom) is implemented correctly. - is_win checks all directions accurately. - The board display is clear and uses colors. - Code is clean and easy to read.	- The AI is purely random; it doesn't try to win or block the opponent. - update_board could use better validation for full columns. - Dynamic allocation used without smart pointers. - Hardcoded board dimensions (6x7).
Infinity_Board.cpp	- The idea of using a queue to track moves is very smart and efficient. - Inheritance from the parent Board class is done correctly. - Grid updates happen smoothly without errors. - Minimal code used to achieve the game rules.	- Computer player has no strategy (Random moves only). - No visual indicator for which move will be removed next. - is_win logic repeats some checks from the standard XO game. - Lacks comments explaining the "Infinity" mechanics.

Islam Dawood's review about Islam Ezzat's code

File	Prof	Cons
TicTacToe4x4.h	- Excellent use of Templates, making the class flexible. - Movement logic for adjacent tokens is calculated efficiently. - Win checking logic (Rows, Cols, Diagonals) is well-organized. - Variable names are expressive and clear (e.g., from_row, to_row).	- cout statements inside update_board violate the Separation of Concerns principle (Logic mixed with UI). - Heavy reliance on Hardcoded Indices in is_win, reducing scalability. - Implementation should ideally be separated into a .cpp file instead of being fully in the Header.
Pyramid_X_O.cpp	- Correct usage of Inheritance from the base Board class. - update_board includes strong validation to prevent moves in invalid/empty pyramid spaces. - The code is simple, readable, and straightforward.	- is_win relies on manual hardcoded checks for every possible winning line, making the code verbose and hard to modify. - Missing const correctness for read-only functions like is_win. - Using using namespace std; in headers (if applied) is considered bad practice.