NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES (KARACHI CAMPUS)
FAST School of Computing
SPRING 2025

PROJECT REPORT

Project title:
Tactic Table

Members:
Khadija Abbasi 24K-0770
Anum Baig 24K-0843
Arwa Mansoor 24K-0930

Date of submission : 20/4/2025

1.Summary

- **Project overview**

  The *Tactic Table* project is a C++ application developed using the Raylib graphics library, aimed at providing a unified platform to play multiple classic board games including Othello, Checkers, and Battleship. The core purpose of the project is to demonstrate practical application of Object-Oriented Programming (OOP) principles in game development. Each game is implemented using encapsulated game logic ensuring modularity and extensibility. The main tasks involved designing game mechanics, handling GUI interactions with Raylib, and maintaining a consistent user interface for game selection and navigation.

- **Key findings**
  - Successfully implemented three complete and playable board games with unique rules and mechanics.
  - Demonstrated OOP concepts such as inheritance, polymorphism, encapsulation, and abstraction in structuring game elements (e.g., pieces, boards, players).
  - Created an intuitive and responsive user interface using Raylib to support cross-game interaction within a single application.

- Designed a scalable framework where additional board games can be integrated with minimal changes to the core structure.

2.Introduction

- **Background**

  Classic board games offer an excellent foundation for learning and applying Object-Oriented Programming principles due to their rule-based and entity-driven nature. The *Tactic Table* project leverages these aspects to build a multi-game system that highlights OOP best practices by creating objects and classes being incorporated into one game project.

- **Project objectives**
  - Develop a modular C++ application that supports multiple board games with distinct logic.
  - Apply OOP principles to structure the game logic in a maintainable way.
  - Use Raylib to create a visually appealing and interactive graphical interface for gameplay and menu navigation.
  - Build a unified user experience that allows switching between games from a common main menu.
  - Ensure the system is designed with extensibility in mind to allow the addition of future games with ease.

3. PROJECT DESCRIPTION

Included Components:

- A main menu that lets users choose between Othello, Checkers, and Battleship.
- Individual game modules with full gameplay logic, rules enforcement, and visual feedback.

Excluded Components:

- Online multiplayer or AI opponents (unless simple AI is implemented for demonstration).
- User profiles and saving/loading game progress.

• Technical Overview:

- Programming Language: C++
- Game Engine/Library: Raylib – for handling graphics, window management, and user input.
- Development Environment: Visual Studio Code
- Compiler/Build Tools: g++, Makefiles or CMake (depending on system configuration)
- Version Control: Git

4.Methodology

- **Approach**

We began by outlining the general architecture and selecting the board games to include.

- **Roles and Responsibilities**

Checkers board game: Anum Baig

Othello: Arwa Mansoor

Battleship: Khadija Abbasi

Main Menu and Integration: Collaborative effort by all team members

5. Project Implementation

- Design and Structure: The Tactic Table app is structured around a main menu that routes the user to one of three game modules. Each game is encapsulated in its own class hierarchy and has:

- Each game has its own independent classes and objects

  Raylib manages all visual output and input events (mouse clicks, key presses). All game logic is separated from rendering code for modularity.

• Functionalities Developed:

- Interactive main menu with game selection
- Mouse-based input and visual feedback for all games

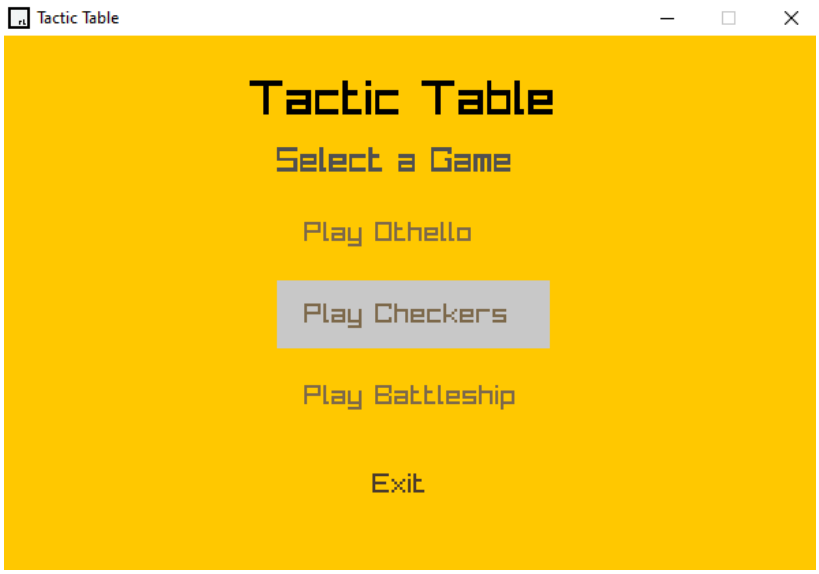
• Challenges Faced:

- Coordinating event handling for different games in a single application loop
- Managing turn switching and input without freezing or misinterpreting clicks
- Ensuring consistent screen scaling and layout across all games using Raylib

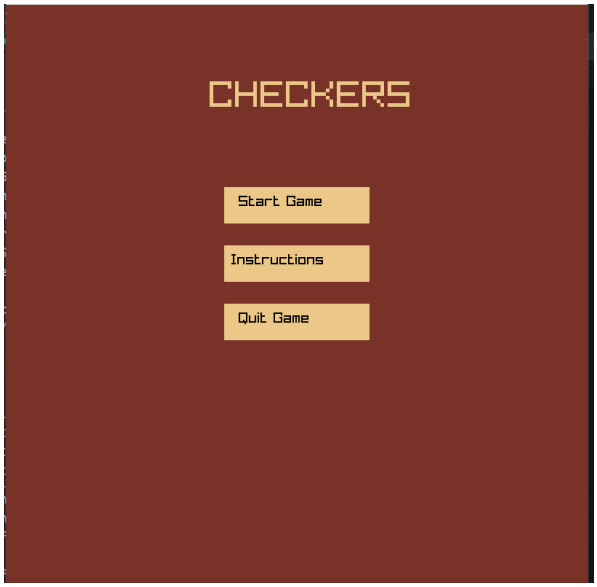
Project Outcomes: The project successfully delivered an interactive application where users can select and play Othello, Checkers, or Battleship. Each game is functional, adheres to its original rules, and provides a smooth graphical experience. The modular structure allows new games to be added in the future with minimal change to the core system.

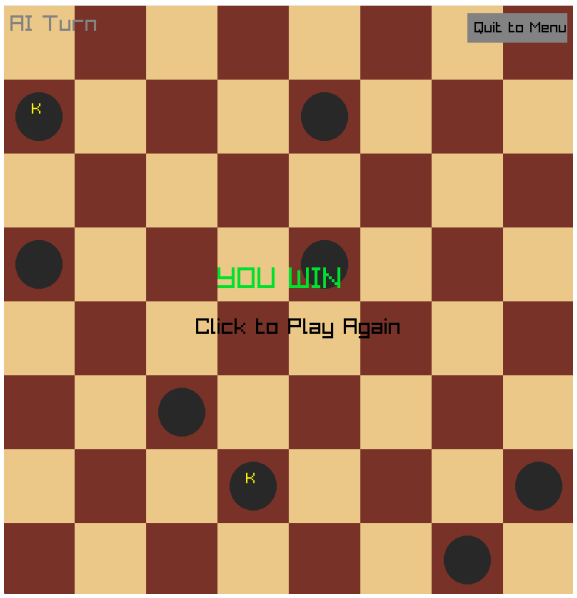
• Screenshots and Illustrations:

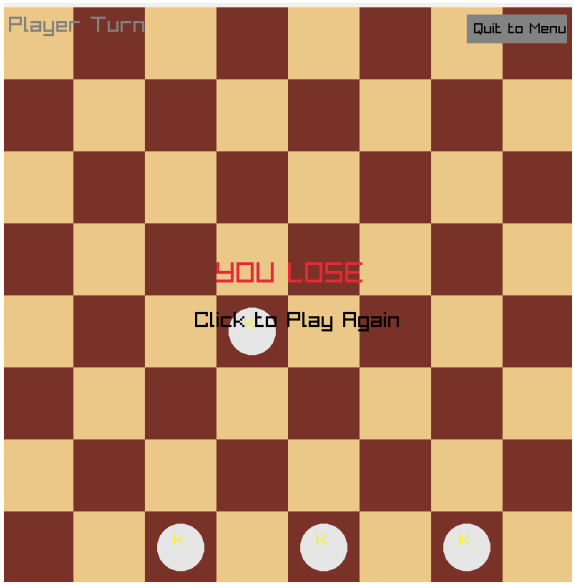
Main Menu



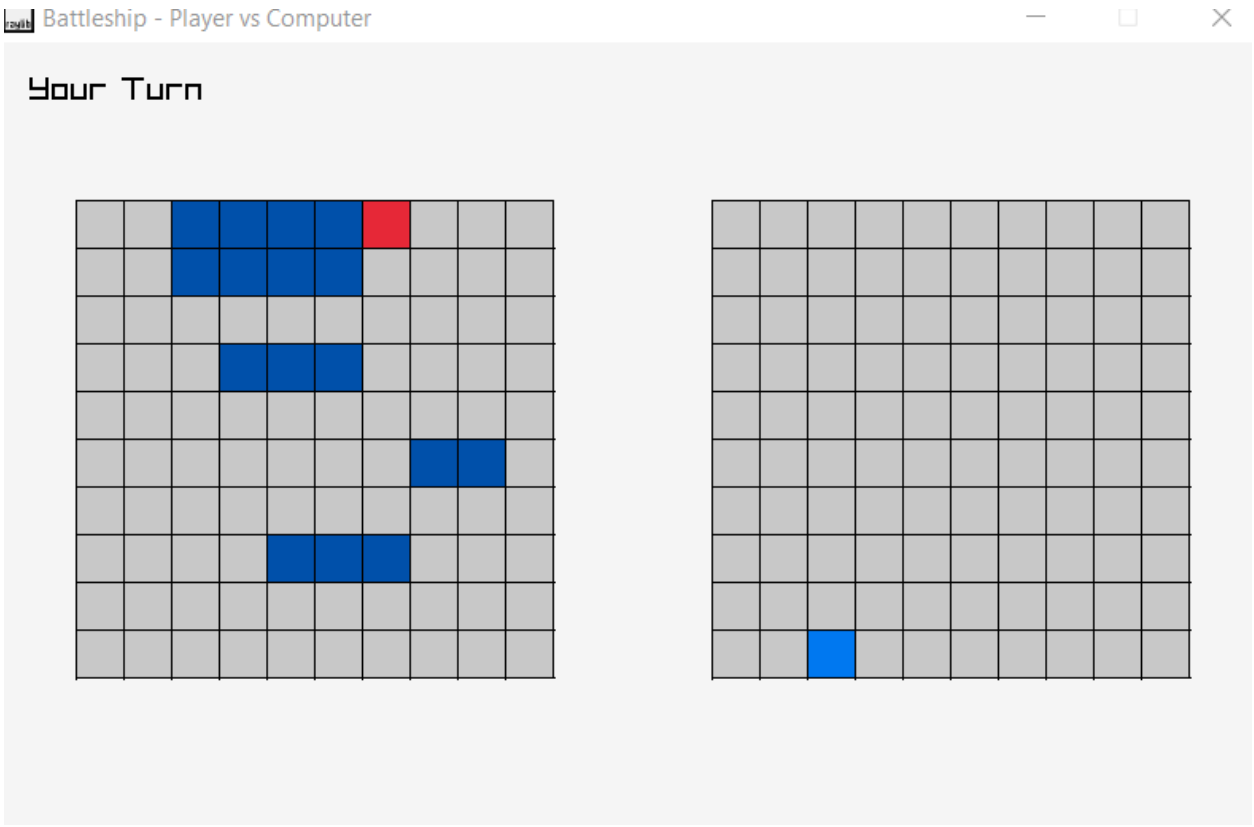
- **Checkers**

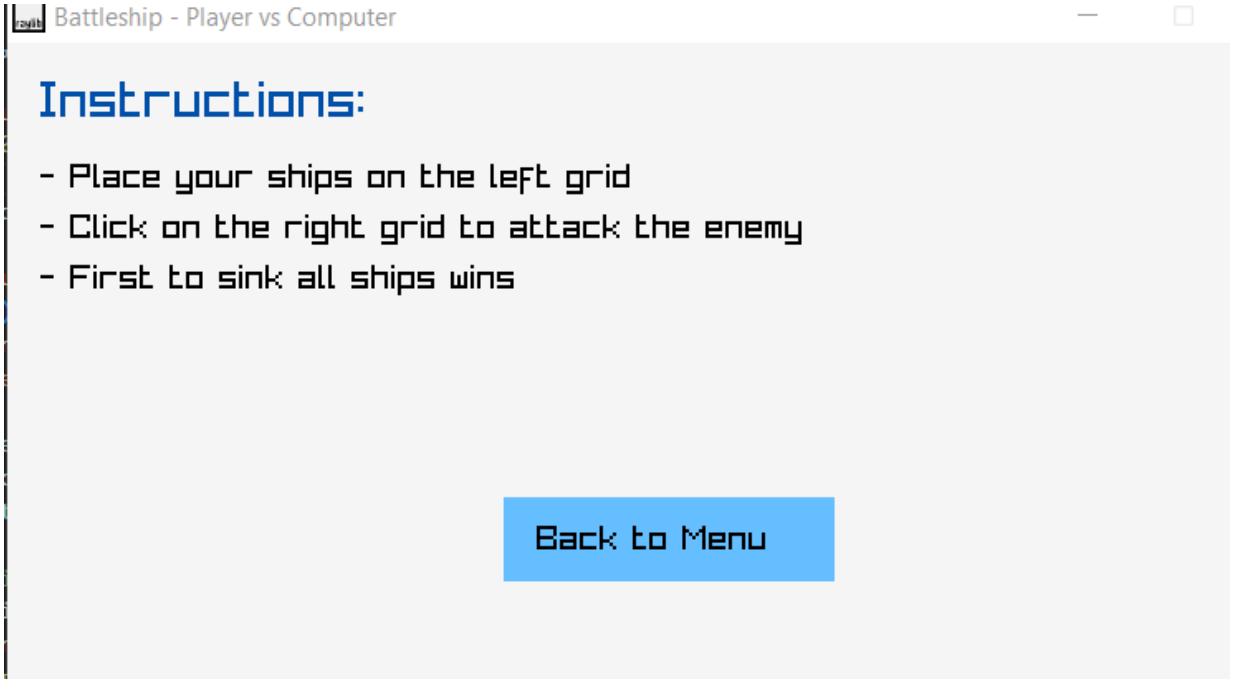






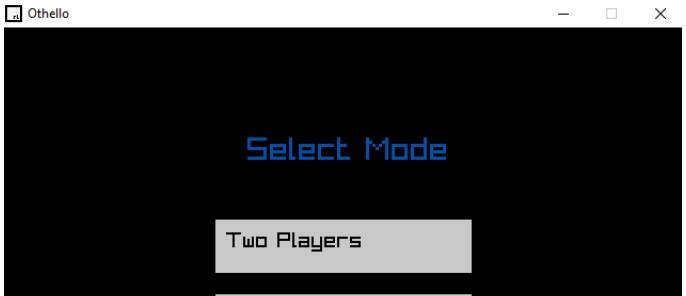
- **Battleship**


Battleship - Player vs Computer
Your Turn

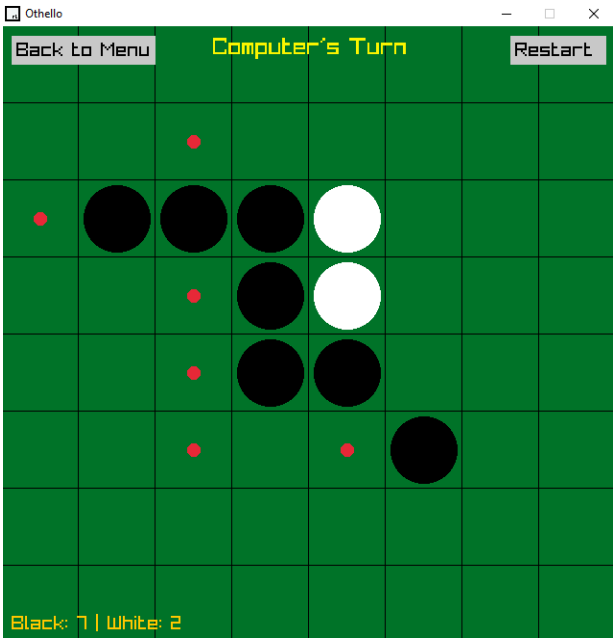

Battleship - Player vs Computer
Instructions:
- Place your ships on the left grid
- Click on the right grid to attack the enemy
- First to sink all ships wins
Back to Menu

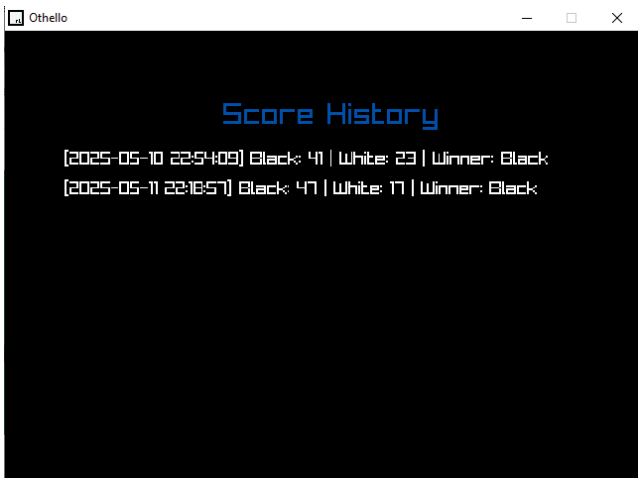
- **Othello**

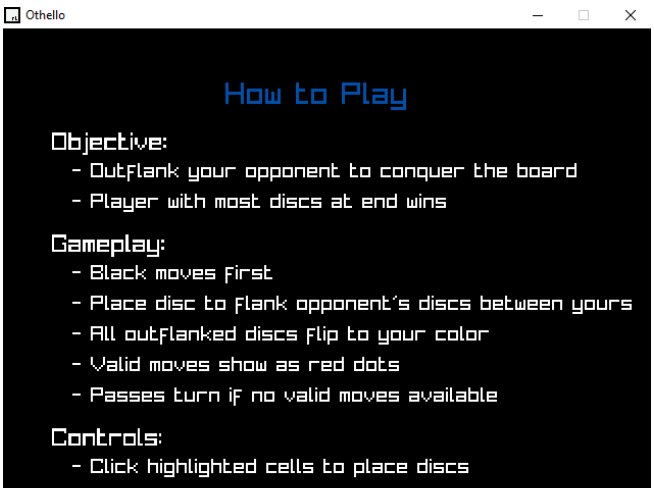

Othello
OTHELLO
A Minute To Learn... A Lifetime To Master!
Play


Othello
Select Mode
Two Players

- **Testing and Validation:**







- **Checkers**

  The Checkers game has been successfully completed. All core features, including GUI enhancements, have been fully implemented.

  Initial Testing

  The testing process began with running the code in the Visual Studio IDE. This helped in identifying and resolving syntax errors and runtime issues early in development. Visual Studio's compiler and debugger were instrumental in spotting incorrect declarations, missing headers, and memory-related issues.

  Functional and Logical Testing

  To validate the game's core logic and rule enforcement, the compiled `.exe` file was executed multiple times. This allowed thorough testing of in-game behavior and interaction rules. Several specific checks were performed:

- Piece Movement Rules: Confirmed that regular pieces could only move forward diagonally, while king pieces were allowed to move both forward and backward diagonally.

- Forced Captures: Tested that whenever a capture move was available, the player (or AI) was restricted to perform it and could not make a normal move instead.

- Multi-Capture Sequences: Verified that after a capture, if the same piece could capture again, the game allowed for a subsequent capture within the same turn.

- Turn Switching: Checked that the turn alternated correctly between the player and AI, especially after valid moves and capture sequences.

1. Encapsulation

- **Definition**: Wrapping data (attributes) and methods (functions) together inside a class, restricting direct access to internal details.

- **Implementation**:

    - Classes like `PieceBase`, `HumanPiece`, `AIPiece`, and `Board` encapsulate their internal data:

        - E.g., `PieceBase` has `row`, `col`, `isKing`, `isAI` as private/protected, only modifiable through methods like `MoveTo`, `MakeKing`, etc.

        - Board management is encapsulated within the `Board` class; external code doesn't access `board[r][c]` directly—it uses methods like `GetPiece()`, `MovePiece()`, etc.

2. Abstraction

- **Definition**: Hiding complex implementation details and showing only the necessary features of an object.

- **Implementation**:

    - The `PieceBase` class acts as an abstract base class with a pure virtual function `Draw()`. This allows treating all pieces uniformly (as `PieceBase*`) while hiding the details of how they are drawn.

    - Users of the `Board` class don't need to know how moves, captures, or king logic is implemented—they just call `MovePiece()`, `HasMoves()`, etc.

3. Inheritance

- **Definition**: A mechanism where a class derives properties and behavior from another class.

- **Implementation**:

`HumanPiece` and `AIPiece` both inherit from `PieceBase`.

```
class HumanPiece : public PieceBase

class AIPiece : public PieceBase
```

    - This allows both types of pieces to share common functionality (like movement, king status) from `PieceBase`, while also customizing their behavior (e.g., different `Draw()` colors).

4. Polymorphism

- **Definition**: The ability to use derived class objects through a base class pointer or reference, and have the correct method called.

- **Implementation**:

    - `Draw()` is a **pure virtual** function in `PieceBase`, and is **overridden** in both `HumanPiece` and `AIPiece`.

    - Wherever a `PieceBase*` is used (like in `board[r][c]->Draw()`), the correct draw function is automatically called depending on the actual piece type (human or AI).

      ```
      board[r][c]->Draw(); // Calls HumanPiece::Draw() or AIPiece::Draw() based on the object
      ```

Battleship

1. Encapsulation

- Data and behavior are grouped into classes (`Ship`, `Battleship`, `Submarine`, `Board`).
- Class members like ship position and grid status are kept private to each object.
- Methods such as `PlaceShip()`, `Attack()`, and `UpdateShips()` operate on internal data without exposing it directly.

---

2. Inheritance

- `Battleship` and `Submarine` **inherit** from the base class `Ship`.
- They reuse and extend the base class's functionality.
- Constructors call the base `Ship(int s)` constructor to set size and other shared attributes.

---

3. Polymorphism

- The `Ship` class defines a **virtual** function `Type()` which is overridden by `Battleship` and `Submarine`.
- Polymorphic array `Ship* ships[SHIP_COUNT]` holds different derived objects (Battleship or Submarine).
- This allows calling `ship->Type()` at runtime to identify the actual ship type.

---

4. Abstraction

- Complex operations like ship placement (`PlaceShipsRandomly()`), grid updates (`UpdateShips()`), and attack logic (`Attack()`) are hidden behind clear method interfaces.

- The main logic uses these methods without needing to know internal details like how a ship is stored or how the grid updates.

Othello

Four pillars of OOP

1. Encapsulation:
    - Classes like `Board`, `Game`, and `Player` encapsulate data and logic.
        - `Board` hides grid details (`board[8][8]`) and exposes methods like `PlacePiece()`.
        - `Game` manages game state privately (`vsAI`, `gameOver`).
    - Example: `private` members (e.g., `flipProgress`) ensure controlled access.
2. Abstraction:
    - The `Player` base class defines abstract methods (`MakeMove()`, `ShowScore()`), hiding implementation details.
    - `Board::CanPlace()` abstracts complex move-validation logic.
3. Inheritance:
    - `HumanPlayer` and `AIPlayer` inherit from `Player`, sharing a common interface.
    - Example: `AIPlayer` overrides `MakeMove()` with AI-specific logic.
4. Polymorphism:
    - Runtime binding of `HumanPlayer`/`AIPlayer` via `Player` pointers in `Game`

File Handling

- Score Saving:
    - `Game::SaveScore()` writes results to `scores.txt` using `ofstream`.
    - Format: `[Timestamp] Black: X | White: Y | Winner: Z`.
- Score Loading:
    - Score history screen reads `scores.txt` with `ifstream`.

Exception Handling

- File I/O Errors:
    - `try-catch` blocks handle file operations (e.g., failed file access).

AI Algorithm

Minimax with Alpha-Beta Pruning

- Algorithm:
    - Recursively evaluates future moves up to depth 2 (performance trade-off).
    - Alpha-Beta Pruning discards non-optimal branches to reduce computation.
- Heuristic Evaluation:
    - Uses a weighted board matrix to prioritize positions

To ensure the functionality and usability of the Othello game, we conducted thorough testing through both manual gameplay and test cases for key components. The game was tested in multiple rounds under the following conditions:

- Player vs Player Mode: Verified correct turn alternation, valid move handling, score updates, and endgame detection.

- Player vs AI Mode: Tested the Minimax algorithm to ensure the AI selects the optimal moves.

- Edge Case Handling: Ensured the game skips turns when a player has no valid moves and correctly identifies a game over when no further moves are possible for either player.

- Score Tracking: Validated that the score counter updates in real-time during gameplay.
- Score History: Saving which player won and when in the text file and displaying in the Score History screen.

- Game Over Screen: Verified proper display of the winning message, final scores, and replay options.

- Graphical Interface: Tested GUI elements using raylib to confirm correct rendering of discs, animations, transitions, and messages such as "Computer’s

Turn".

- Usability Testing: Informal user testing was done by peers, who played the game and provided feedback on UI clarity, turn visibility, and smoothness of gameplay flow.

Overall, the game was found to be stable and user-friendly, with all core functionalities working as expected across different modes of play.

7. Conclusion

• **Summary of Findings:** Tactic Table achieved its goal of building a unified multi-board-game platform using object-oriented principles. We managed to create scalable game logic. The Raylib GUI helped deliver a visually interactive experience across all games.