SP14 - Red Chess AI - Final Draft

2025 Fall Semester
CS 4850
Prof. Sharon Perry

Carter Barnard
Andy Kenmoe
Bailey Sweeney
Alec Walsh

April 2025

Links:

GitHub - https://github.com/SP14-ChessAI-Red-2025
Website - https://sp14-chessai-red-2025.github.io/website/

STATS AND STATUS	
LOC (Lines of Code)	2370
Tools	Python, C++, Cython, PyGame, React, Flask
Hours Estimate	450
Hours Actual	550
Status	98% working as designed. Missing small, optional features like player timers.

Table of Contents

1. Introduction

This project focuses on the development of a Chess AI game engine using Pygame. The game engine is designed to simulate an engaging chess-playing experience. The engine supports three primary gameplay modes: Player vs Player (PvP) for competition between two human players, Player vs. AI (PvAI) for single-player challenges against an artificial intelligence opponent, and AI vs AI (AI v AI). The AI is designed to analyze and evaluate potential moves efficiently, challenging each player's skill.

A self-play feature is integrated allowing the AI to play against itself to optimize different strategies. The engine ensures compliance with standard chess rules, implementing functionalities such as move validation, stalemate detection, and checkmate detection.

1.1 Scope

The scope of this system is to build a Chess AI app that utilizes Pygame. It will support hosting a secure game between players, including AI players. The system will not support saving games. The game system is designed to simulate an engaging chess-playing experience. The system supports three primary game-play modes: PvAI for single player challenges against an AI opponent, and AIvAI. The AI is designed to analyze and evaluate potential moves efficiently, challenging the skill of each player.

1.2 Definition and Acronyms

- PvAI - Player versus Artificial Intelligence
- AIvAI - Artificial Intelligence vs Artificial Intelligence
- AI - Artificial Intelligence
- IDE - Integrated Development environment
- JS - JavaScript
- CSS - Cascading Style Sheets
- JSON - JavaScript Object Notation

2. Requirements

2.1 Use Cases

UC1. **Select Opponent**

Actor: User

Goal: Choose between playing against an AI or a human player

Priority: High

Trigger: The "Start Game" use case is initialized.

Functional Requirements:

i. The system shall display the menu of available options.
ii. The systems shall allow the user to choose options “AI” or “Online Player”

UC2. **Start Game**

Actor: User

Goal: Start playing game against the selected opponent.

Priority: High

Trigger: An “AI” or “Online Player” was selected as an opponent.

Functional Requirements:

i. The system shall display a board and piece colors (black or white).
ii. The system shall display each player’s piece color and control shifts to the player with the white pieces, and the timers are initialized.

UC3. **Make Move**

Actor: User

Goal: Make a legal move against the opponent.

Priority: Medium

Trigger: The game and timer started.

Functional Requirement:

i. The system shall highlight the players turn.
ii. The system shall let players click and mover chess piece.
iii. The system shall display moved chess pieces.
iv. The system shall allow the opponent to make a move if the game has not ended.

UC4. **Check Move**

Actor: System

Goal: Determine if the move made is valid and determine if the move is “checkmate.”

Priority: High

Trigger: A player has made a move.

Functional Requirements:

i. The system must check game logic to determine if the move is legal.
ii. The system shall display “Checkmate” if the opposing player is in ‘Check’ (state where the king is under attack) and has no legal moves.

UC5. **Start Timers**

Actor: System

Goal: Start the timer of each player to show how long each player has to make a move.

Priority: High

Trigger: The game has started and the player with the white chess piece has made a move.

Functional Requirements:

i. The system shall display a timer counting down it original values.
ii. The system shall display a paused timer when a legal move has been made.

UC6. **Terminate Game**

Actor: User or System

Goal: Determine if the game must be ended.

Priority: Medium **Trigger:**

i. User clicks “Forfeit”
ii. The player’s allotted time runs out.

Functional Requirements:

i. The system shall allow the user to click the “Forfeit” button and immediately end the game.
ii. The system shall display “Stalemate” for both players if players are not in ‘Check‘ and has no legal moves to make.
iii. The system shall allow user to click “Draw”
iv. The system shall allow the opponent to click “Accept Draw” or “Reject Draw”
v. The system shall display the board and continue the game if “Reject Draw” was selected.
vi. The system shall return user to the “Select Opponent” use case.
vii. The system shall display game results after the game has ended.

2.2 Use Case Matrix

Event Name	Input/Output	Summary
User starts a game	• Bot - Easy (IN) • Bot - Normal (IN) • Bot - Hard(IN) • PvP (IN)	The users clicks on the difficulty levels to enter a game mode. Upon clicking a game mode the chess game will begin.
User moves a piece	• Drag and Drop Piece (IN) • Board (Out)	The player drags and drops a piece on the board, if the move is valid the board is updated and redrawn on screen.
User offers a draw	• Draw Button (IN) • Accept Draw Button (IN/Out) • Reject Draw Button (IN/Out)	The player clicks on the Draw button. The opponent is presented with two buttons "Accept" and "Reject" buttons, if the opponent clicks the "Accept Draw" button the game ends in a draw, if the opponent clicks the "Reject" draw button the game continues.
User forfeits	• Forfeit Button (IN)	The player clicks on the forfeit button, which causes the current player to lose the game and the game ends.

Table 1: Event Table

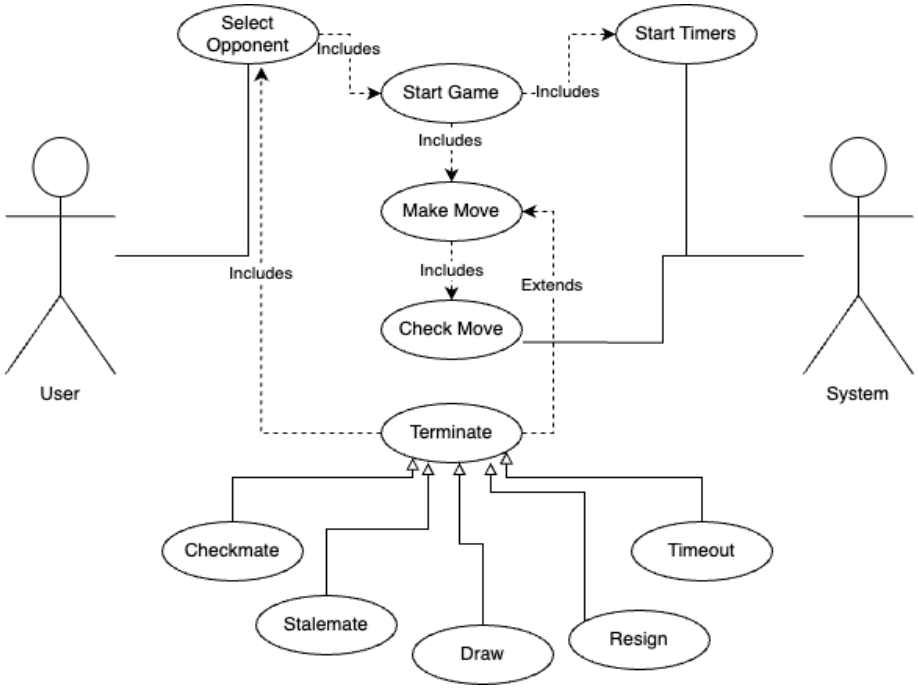


Figure 1: UML Use Case Diagram

3. Non-functional Requirements

3.1 System Attributes

- Compatibility
  - The system shall work seamlessly across different devices and operating systems.
- Usability
  - The system shall have a user-friendly interface for players of all skill levels and age groups.
- Portability
  - The user can access the game from their mobile with internet connection for PvAI match.
- Reliability
  - The system shall save player previous records.
  - The system shall allow all moves to be processed correctly and consistently without errors, ensuring game rules are always followed.
  - The system shall store and manage player game records securely.
- Performance
  - The system shall be able to handle a large number of moves and player actions without slowing down or crashing.
  - The system shall run smoothly without any delays and lag.

4. Design Considerations

4.1 Assumptions and Dependencies

The development of this chess game and engine is based on several key assumptions and dependencies. Python will handle the game interface and user interactions using PyGame, while C++ will implement the chess engine for optimized performance. The project will adhere strictly to standard chess rules without introducing additional variants. Additionally, the system depends on PyGame for Python and standard C++ libraries for execution. Our primary focus is local play, but a secondary objective to the project is to add online play between two players.

4.2 General Constraints

The application will initially be developed for Windows, with cross-platform technologies used to facilitate easy porting to other operating systems. It is designed to support AI-driven gameplay, focusing on AI vs AI and AI vs Player modes.

4.3 Development Methods

An Agile development methodology will be followed, incorporating iterative development cycles and regular team check-ins. Version control will be managed via Git, ensuring efficient collaboration and change tracking.

4.4 Architectural Strategies

The architecture of the chess game and engine will be structured to balance performance and usability. Python will be employed for the graphical user interface, leveraging PyGame's simplicity for handling 2D graphics, while the chess engine will be developed in C++ to enable efficient move calculations and game logic. Optimization techniques such as Minimax and Alpha-Beta pruning will be implemented to enhance AI performance. The communication between Python and C++ will be managed via a Python-C++ API or socket-based communication to ensure seamless data exchange between the two components.

4.5 Use of Python and C++

The chess game will utilize Python for the graphical user interface (GUI) and C++ for the chess engine. This decision is primarily motivated by performance considerations. Python's PyGame library will be used to handle 2D graphics and user interactions, offering a simple and efficient way to manage the game's interface. C++ will power the chess engine, optimizing move calculations and game logic, which are computationally intensive tasks. This division of labor allows the system to take advantage of Python's ease of use for the interface and C++'s performance for processing.

The communication between Python and C++ will be facilitated using a Python-C++ API, such as pybind11 or Python, to ensure efficient data exchange between the GUI and the chess engine. This API will allow for the transfer of board states, move evaluations, and player inputs, enabling smooth communication between the Python-based interface and the C++ based engine.

4.6 Hardware and Software Interface Paradigms

The game will be designed to run on typical personal computers. While optimized for local play, future versions may consider cloud-based architecture for running the chess engine, thus enabling online multiplayer.

4.7 Concurrency and Synchronization

The chess engine will utilize concurrency. When determining its next move, it has to search the game tree. Different branches of the game tree are independent, so they can be searched simultaneously from different threads. Each thread will determine a score for the move it is considering. The main thread will wait for all worker threads to finish, then choose the move with the highest score. An advantage of this approach is that minimal synchronization is required, as each thread operates on a different sub-tree.

4.8 Memory and Resource Management

Memory management is a critical consideration. Memory will be carefully managed by ensuring that dynamically allocated memory is freed before returning data to Python. Functions that allocate memory will also provide corresponding functions for freeing that memory. This explicit management minimizes memory leaks and ensures efficient memory usage, which is particularly important when the game is running on machines with limited resources.

For Python, memory management is handled automatically, but careful handling of large data structures and objects will be necessary to avoid memory bloat.

4.9 Optimization Techniques for AI Performance

To enhance the AI's decision-making process, the Minimax algorithm will be employed, coupled with Alpha-Beta pruning to optimize the tree search. This approach ensures that the system can evaluate potential moves efficiently by pruning branches that do not need to be explored.

4.10 Error Detection and Recovery

Code will use error handling that is idiomatic for the language in which it is written. Python code will primarily use Python exceptions. C++ will use C++ exceptions. The C++ code does not deal with any user input and is purely computational, so it should not be able to fail during normal operation. Running out of memory will not be a recoverable error but should still be handled by gracefully exiting the application, without crashing.

4.11 Distributed Control

While the current design is focused on a local system with no distributed control, future versions may explore distributed systems for online multiplayer. In such a scenario, the system would manage game state synchronization and player connections over a network, likely utilizing a networking protocol like WebSocket for low-latency communication.

5. System Architecture

There are two main parts of our system, the Game Interface Module (the chess game) and the Chess Engine Module. The Game Interface Module, developed in Python, will handle rendering, user input, and display logic. The Chess Engine Module, implemented in C++, will be responsible for processing game logic, validating moves, and maintaining the game state. An Integration Layer will be established to bridge Python and C++ components, ensuring smooth data flow and interaction. We have planned a Networking Module to be incorporated to enable online multiplayer functionality, but this is not a primary concern for the base prototype.

5.1 Memory Management

As C++ is not a garbage collected language, care must be taken when allocating memory. Most allocated memory will be freed before any data is returned to Python. Some functions, however, may dynamically allocate memory, and return a pointer to that memory to Python. These functions will have a corresponding function for freeing the memory.

5.2 Thread Management

The program will leverage parallelization to optimize performance, particularly for tasks that can be executed concurrently, such as AI calculations and move evaluations in the chess engine. By distributing these tasks across multiple threads, the system can take advantage of multi-core processors, potentially providing significant speedups on suitable hardware. Proper synchronization will be implemented to maintain the integrity of the game state while ensuring parallel processing maximizes the utilization of available resources.

5.3 Interface/Exports

The system will facilitate interaction through well-defined interfaces and exports. A Python-C++ API will be used, such as the ctypes library in the Python standard library, to enable seamless data exchange between the GUI and chess engine, allowing the game to send board states and receive valid move computations. The graphical user interface will incorporate elements such as buttons, a chessboard grid, and game status indicators to provide a smooth and interactive user experience.

For the development of more complex and interactive GUI elements, libraries like Tkinter or PyQt could be considered. These frameworks provide robust tools for creating standard UI components such as buttons, sliders, menus, and text fields, which are essential for handling user input and managing game settings.

5.4 Screen Mock-ups

The game will feature three distinct screens. There are variants to these screens depending on user preference and game win condition. A collapsible menu (sidebar) will be integrated to enhance navigation and accessibility. The entire flow is below.

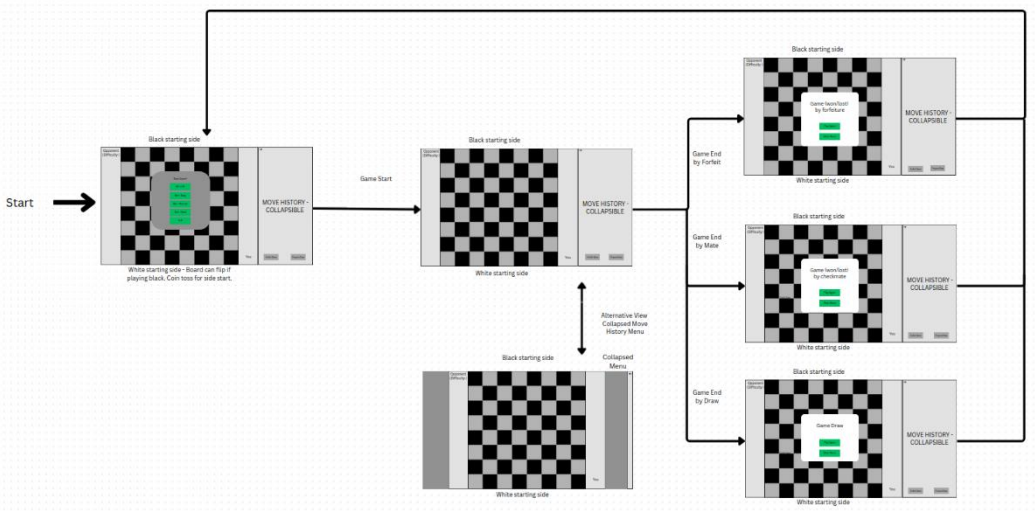


Figure 2: System Flow

5.5 Individual Screens

Main Menu Screen

Displays options to start the game. The user may choose any difficulty or mode. The options are planned to be:

- **AI vs AI**
- **Bot - Easy**
- **Bot - Normal**
- **Bot - Hard**

- **PvP**

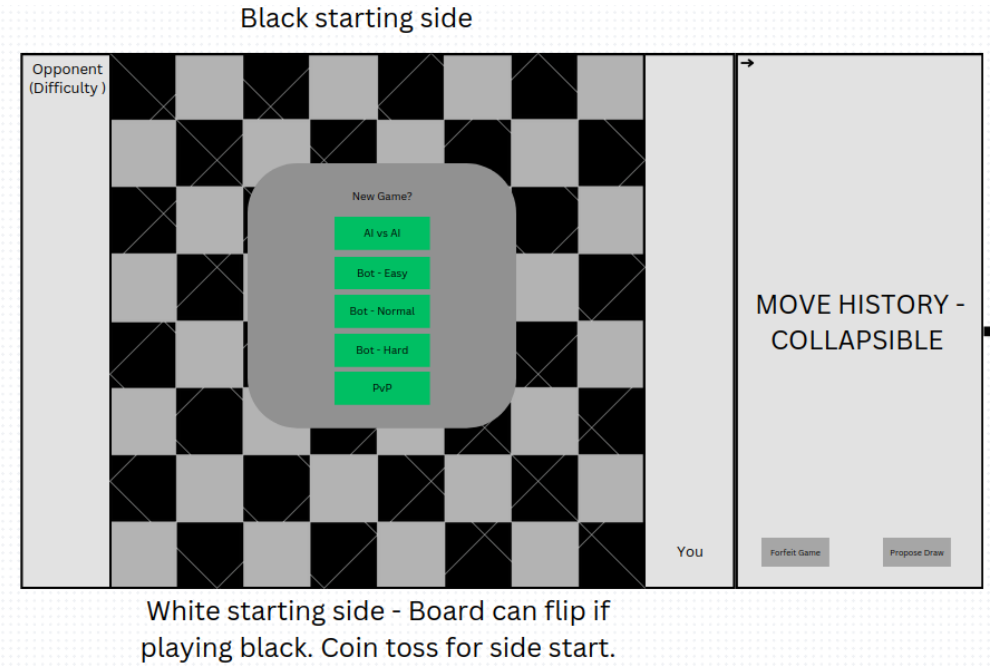


Figure 2: Main Menu

Game Screen

Showcases the chessboard, player information, and move history. This is the same screen that will show during AI vs AI matches. The menu on the right-hand side is collapsible. The menu contains buttons to forfeit or propose a draw.

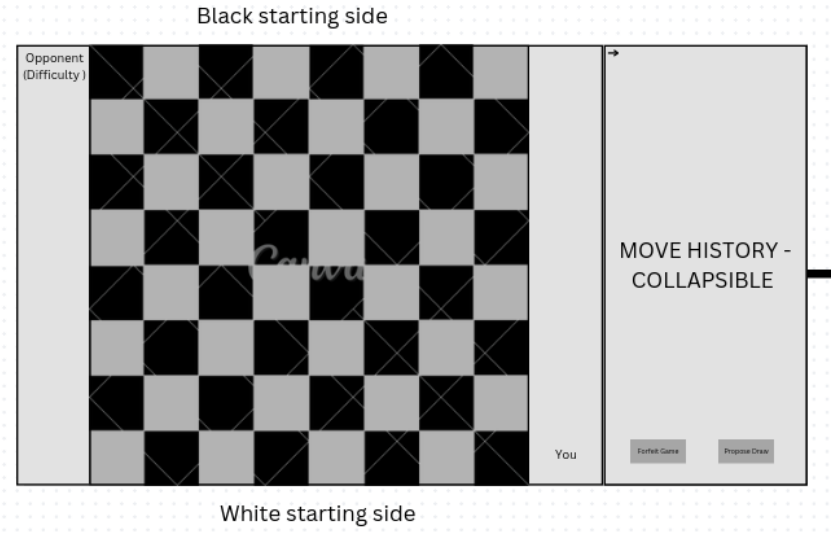


Figure 3: Game Starting Position

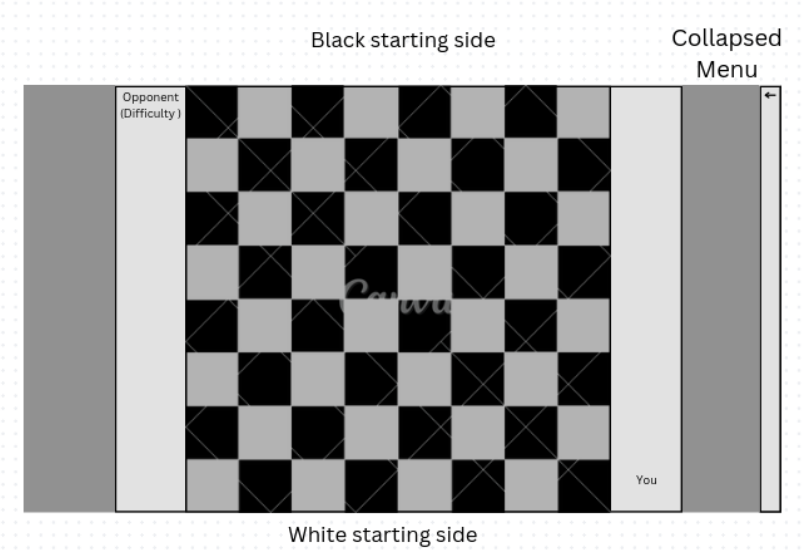


Figure 4: Menu Collapsed

Game End

Three types of screens will display depending on the win condition (checkmate, draw, forfeit).

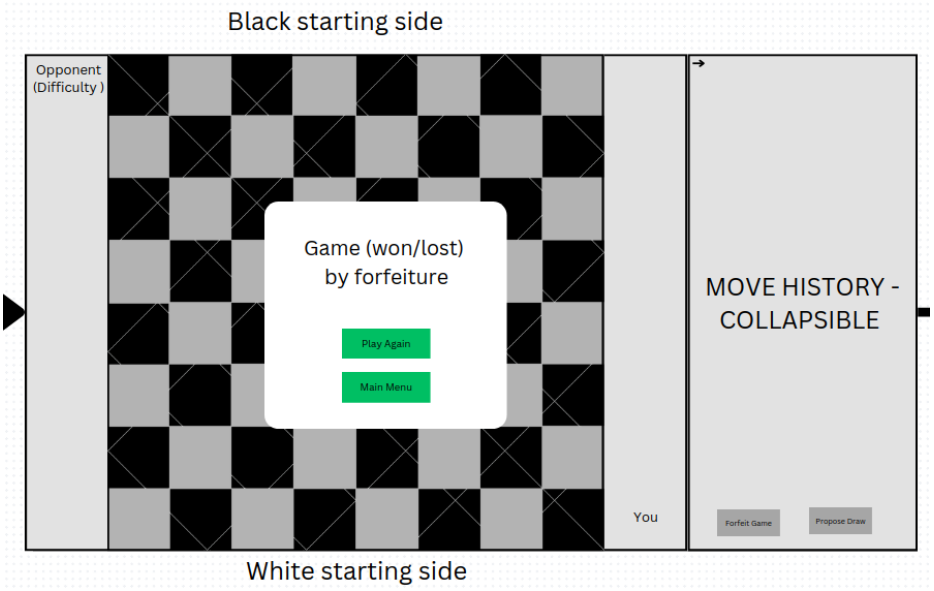


Figure 5: Game Forfeit

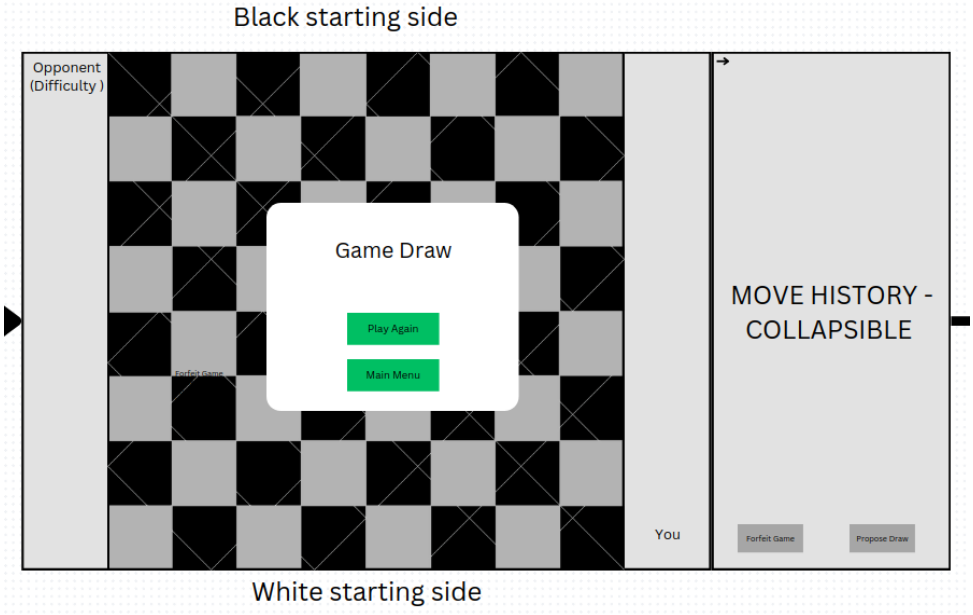


Figure 6: Game Draw

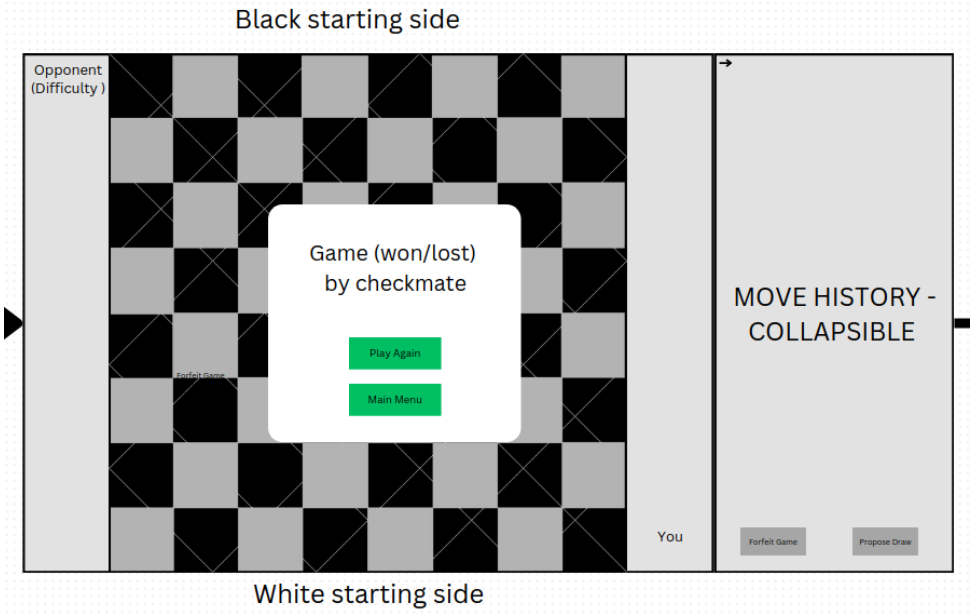


Figure 7: Game Checkmate

6. Test Plan and Report

6.1 Functional Requirements Tests

Select Opponent

1. **FR1-T**

   Type: Manual, Dynamic, Structural

Initial State: A user is presented with options of playing against an AI opponent or online player.

Input: User selects an online player.

Output: A game will start with two players.

Test will be performed: A team member will choose to play against another team player and verify that the game starts with the team player.

2. **FR2-T**

Type: Functional, Dynamic, Structural

Initial State: A user is presented with options of playing against an AI opponent or online player. Input: User selects an AI player.

Output: The system attempts to start a game against an AI opponent.

Test will be performed: A team member will choose to play against an AI player and verifies that the system attempts to start a game.

Start Game

1. **FR3-T**

Type: Manual, Functional, Dynamic

Initial State: The system presents the players with a view of the board.

Input: The game will begin to call the "Start Game" use case.

Output: The system initializes the game after calling the "Start Game" use case, and both players may now play.

Test will be performed: Two members of the team will join a game, and system shall check if the "Start Game" use case has been called.

Make Move

Await Opponent Move

1. **FR4-T**

Type: Manual, Functional, Dynamic

Initial State: The board and pieces have been initialized.

Input: The system has given exclusive control to the player with the white pieces and has started that player's timer.

Output: The system shall not allow the player to make a move until their opponent has made a move, with the exception of the first move of the game.

Test will be performed: One member of the team will start a game and check to see if they are allowed to make a move before their opponent if they have the black pieces.

Move Piece

2. **FR5-T**

Type: Manual, Functional, Dynamic

Initial State: The system has given exclusive control to the player with the white pieces and has started that player's timer.

Input: The system shall not allow the player to make a move until their opponent has played a move, with the exception of the first move of the game.

Output: If a player clicks a piece, holds the mouse down, drags the piece to a square, and releases the mouse, the system shall register an attempted move.

Test will be performed: One member of the team will start a game and check to see if they are allowed to drag and drop a piece to make a move on the board.

Check Legal Move

3. **FR6-T**

Type: Manual, Functional, Dynamic

Initial State: The system shall not allow the player to make a move until their opponent has played a move, with the exception of the first move of the game.

Input: If a player clicks a piece, holds the mouse down, drags the piece to a square, and releases the mouse, the system shall register an attempted move.

Output: The system shall confirm that the move is legal as defined by the rules of the game.

Test will be performed: One member of the team will start a game and check to see if the "Check legal Move" use case is triggered after moving a piece.

Legal Move

4. **FR7-T**

Type: Manual, Functional, Dynamic.

Initial State: If a player clicks a piece, holds the mouse down, drags the piece to a square, and release the mouse, the system shall register an attempted move.

Input: The system shall confirm that the move is legal as defined by the rules of the game.

Output: If the attempted move is legal, the move shall be sent to the server and the opponent player shall be given control.

Test will be performed: One member of the team will make a legal move and ensure that it is reflected in the board state and that they are unable to make a second move with the same piece.

Illegal Move

5. **FR8-T**

Type: Manual, Functional, Dynamic

Initial State: If a player clicks a piece, holds the mouse down, drags the piece to a square, and releases the mouse, the system shall register an attempted move.

Input: The system shall confirm that the move is legal as defined by the rules of the game.

Output: If the attempted move is not legal, the move shall not be sent to the server and the opponent player shall retain control.

Test will be performed: One member of the team will make an illegal move and ensure that it is not reflected in the board state and that they are not able to attempt another move with the same piece.

Check "Terminate Game" use case

6. **FR9-T**

Type: Manual, Functional, Dynamic

Initial State: If a player clicks a piece, holds the mouse down, drags the piece to a square, and release the mouse, the system shall register an attempted move.

Input: The system shall confirm that the move is legal as defined by the rules of the game.

Output: The 'Terminate Game" use case shall be checked after every move is made.

Test will be performed: One member of the team will check if the "Terminated Game" use case is triggered after moving a piece.

Resume Game Timer

7. **FR10-T**

Type: Manual, Functional, Dynamic

Initial State: The system shall confirm that the move is legal as defined by the rules of the game.

Input: The "Terminate Game" use case shall be checked after every move is made.

Output: If the game is not terminated, the timer for the other player shall resume.

Test will be performed: One member of the team will check to see if the timers swap successfully while the game is not terminated.

Verify Move

1. **FR11-T**

   Type: Manual, Dynamic

   Initial State: A user has made a move

   Input: A valid move based on the current game state.

   Output: A positive signal will be returned to signify that the game state should be updated to reflect the input move.

   Test will be performed: The system will input a valid move into the check legal move and verify that a positive signal is returned.

2. **FR12-T**

   Type: Manual, Dynamic

   Initial State: A user has made a move.

   Input: An invalid move based on the current game state.

   Output: A negative signal will be returned to signify that the game state should not be updated.

   Test will be performed: The system will input an invalid move into the check legal move module and verify that a negative signal is returned.

Update Board State

3. **FR13-T**

   Type: Manual, Functional, Dynamic

   Initial State: A move has been made and a signal has been passed returned.

   Input: An inputted move and a positive or negative signal representing if the move was legal.

   Output: An update to the board state data representing a legal move made.

   Test will be performed: The system will input a valid move and verify that the same move is reflected on the board state.

4. **FR14-T**

   Type: Manual, Functional, Dynamic

   Initial State: A move has been made and a signal has been passed returned.

Input: An inputted move and a positive or negative signal representing if the move was legal.

Output: An update to the board state data representing a legal move made.

Test will be performed: The system will input a invalid move and verify that the same move is not reflected on the board state.

Lock Player Control

5. **FR15-T**

Type: Manual, Functional, Dynamic

Initial State: A legal move has been made and the board state has been updated.

Input: Both players attempted to make a move simultaneously.

Output: Only one move will occur.

Test will be performed: The system will input a valid move from both players simultaneously and verify that only the move of the player will control is reflected on the board state,

Start Timers

Start Timer On Game Start

1. **FR16-T**

Type: Manual, Functional, Dynamic, Structural Initial State: A game has been started between two players.

Input: None.

Output: One timer has been initialized for each player. The timer for the player with the white pieces has started counting down.

Test will be performed: Two members will start a game and verify that they each have a timer and that the timer for the player with the white pieces has started counting down.

Initialize Timer Values

2. **FR17-T**

Type: Functional, Dynamic

Initial State: A game has been started between two players

Input: None

Output: Timers have been initialized for each player with 10:00 minutes of total time.

Test will be performed: Two team members will start a game and verify that they each have a timer with 10:00 minutes of total time.

Pause and Resume Timers

3. **FR18-T**

   Type: Manual, Functional, Dynamic

   Initial State: A game has been started between two players.

   Input: Both players input valid moves in valid sequence.

   Output: After each valid move, the player who made the move will have their timer paused and the other timer will resume.

   Test will be performed: Two team members will start a game with each other and make valid moves in a valid sequence. They will verify that the timers are pausing and resuming after each valid move.

4. **FR19-T**

   Type: Manual, Functional, Dynamic

   Initial State: A game has been started between two players.

   Input: Both players input valid moves in valid sequence.

   Output: After each valid move, the player who made the move will still have their timer going, and the other player will have their timer paused.

   Test will be performed: Two team members will start a game with each other and make invalid moves. They will verify that the timers are not pausing and resuming after any invalid moves.

Terminate Game

Establish System Checks

1. **FR20-T**

   Type: Manual, Functional, Dynamic

   Initial State: The 'Terminate Game' use case shall be checked after every move is made.

   Input: The system will have checks to establish if the game shall be terminated with the appropriate use case possibilities.

   Output: These checks include the Checkmate, Stalemate, Draw, Resignation, and Timeout use cases.

   Test will be performed: One member of the team will start a game and check to see if the "Terminate Game" use case is triggered.

Check if a Checkmate has occurred

2. **FR21-T**

Type: Manual, Functional, Dynamic

Initial State: The 'Terminate Game' use case shall be checked after every move is made.

Input: The system shall determine if the opposing player is in 'Check' (state where the King is under attack) and has no legal moves for the King.

Output: If true, this shall result in a victory with 'Checkmate' for the current player.

Test will be performed: One member of the team will play a game and check to see if they have achieved a 'Checkmate'.

6.2 Nonfunctional Requirements Tests

Usability and Humanity

Ease of Use

1. **UH1-T**

Type: Manual, Functional (Usability), Dynamic

Initial State: The game is started and presented to a 14-year-old.

Input/Condition: The children are presented with the interface to play the game. The children should be new to chess.

Output/Result: around 90% of the test subjects should be able to play a move within five minutes of starting the presented program without any assistance.

Test will be performed: The test will be performed by presenting the game to a 14-year-old.

6.3 Traceability Between Test Cases and Requirements

Requirements-driven test cases named with the following naming convention: Req#-Test. T stands for test, Req# represents specific requirements, such as FR1 or UH2. E.g. FR2-T represents a test for functional requirement with ID FR2.

Traceability Matrix

Requirement ID	Description	Test ID
FR1	Select Opponent	FR1-T
FR2	Select Opponent	FR2-T

FR3	Start Game	FR3-T
FR4	Await Opponent Move	FR4-T
FR5	Move Piece	FR5-T
FR6	Check Legal Move	FR6-T
FR7	Legal Move	FR7-T
FR8	Illegal Move	FR8-T
FR9	Check “Terminate Game” use case	FR9-T
FR10	Resume Game Timer	FR10-T
FR11	Verify Move	FR11-T
FR12	Verify Move	FR12-T
FR13	Update Board State	FR13-T
FR14	Update Board State	FR14-T
FR15	Lock Player Control	FR15-T
FR16	Start Timer on Game Start	FR16-T
FR17	Initialize Timer Values	FR17-T
FR18	Pause and Resume Timers	FR18-T
FR19	Pause and Resume Timers	FR19-T
FR20	Establish System Checks	FR20-T
FR21	Check if Checkmate has occurred	FR21-T
UH1	Ease of Use	UH1-T
PR1	System Response Time	PR1-T

Table 1: **Traceability Matrix**

6.4 Player Control Validation

Player Control

1. **PC1-T**

   Type: Functional, Dynamic

   Initial State: Both player boards have been initialized and the game is in play.

   Input: The current player has the first move, or the opponent has made a move.

   Output: The current player is given complete control to the board and can choose to make any legal move they want. The opponent may not make a move during this period.

   Test will be performed: The system will make the first move for the player with the white pieces. The test is deemed successful if the player is able to do so.

2. **PC2-T**

   Type: Functional, Dynamic, Manual

   Initial State: Both player boards have been initialized and the game is in play.

   Input: The opponent player is to make the first move, or the current user has made their move.

   Output: The current player has been restricted all control to the board, and cannot make any legal move on the board.

   Test will be performed: The system will make the opponent start with the white pieces. The test is deemed successful if the current player, with the black pieces, is not able to make a move on the board before the opponent makes a move.

7. Existing Implementation Comparison

A test that compares the system implementation against the existing implementation is as follows:

Board State Comparison

1. **EI1-T**

   Type: Functional, Dynamic, Manual

   Initial State: A game has been started on the system as well as on the existing implementation.

   Input: The same set of valid moves is input into both systems.

   Output: Both systems have the same state response after each input.

   Test will be performed: Two team members will start a game on the system as well as the existing implementation. The team members will play the same moves on both systems and ensure the same response occurs on both systems.

Description	Pass/Fail	Severity
Select Opponent	Pass	
Start Game	Pass	
Await Opponent Move	Pass	
Move Piece	Pass	
Check Legal Move	Pass	
Legal Move	Pass	
Illegal Move	Pass	

Check “Terminate Game” use case	Pass	
Resume Game Timer	Fail	Low
Verify Move	Pass	
Update Board State	Pass	
Lock Player Control	Pass	
Start Timer on Game Start	Fail	Low
Initialize Timer Values	Fail	Low
Pause and Resume Timers	Fail	Low
Establish System Checks	Pass	
Check if Checkmate has occurred	Pass	
50 move rule supported	Pass	
Threefold repetition rule	Fail (Only implemented in React frontend)	Low
Threading support	Fail (Implementation not finished)	Low

8. Version Control

Managed through GitHub. We utilized different branches for different changes for different people. When committed, we would merge the changes into a main branch that we'd push commits to. We used separate repositories for the frontend and backend.

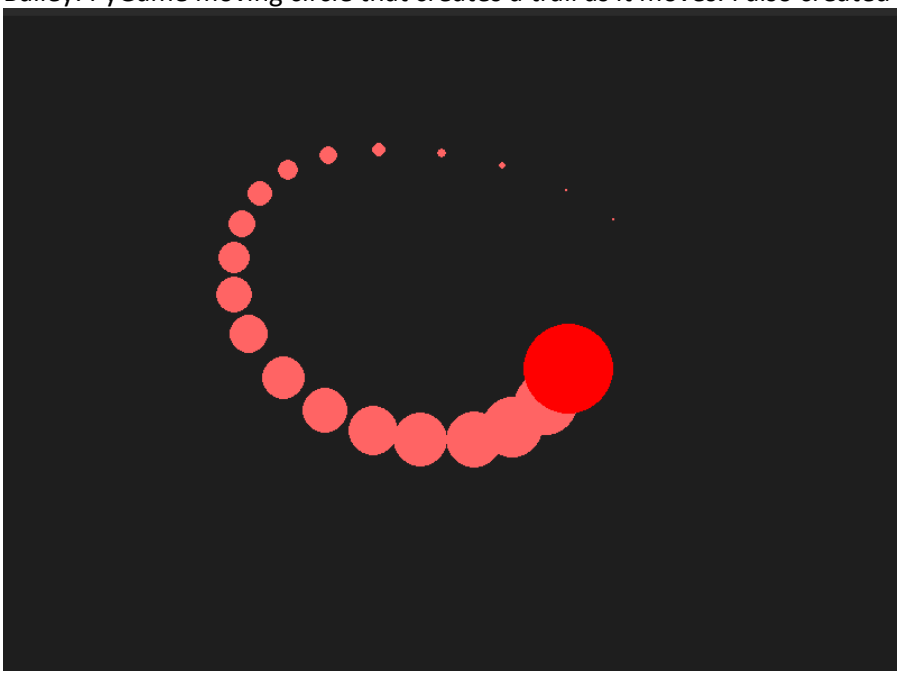
There are several partially implemented features that you can find in different branches of the repository. The threading_wip implements threading support for the backend. There are several bugs that we didn't have time to fix, however, and we did not consider it reliable enough to merge into the main branch. The threefold repetition rule was only implemented in the branch that supports the React based frontend.

9. Conclusion/Summary

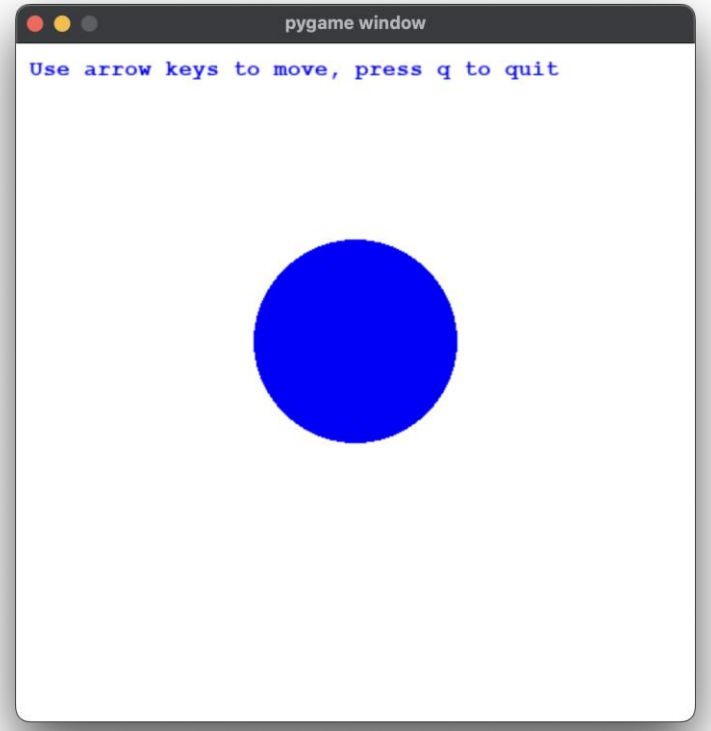
This project successfully developed a Chess AI game engine utilizing Python with Pygame for the user interface and C++ for the core engine logic, aiming to balance usability and performance. Key features include multiple gameplay modes (PvP, PvAI, AIvAI), adherence to standard chess rules, and an AI opponent employing Minimax with Alpha-Beta pruning and concurrency for optimized decision-making. The architectural design separated concerns between the interface and engine, communicating via a Python-C++ API. Development followed an Agile methodology with version control managed via Git. Comprehensive testing validated most functional requirements, including core game logic and AI behavior, although issues were identified with opponent selection and timer functionality, indicating specific areas for further refinement. Overall, the project established a robust foundation for a chess game, demonstrating the effective integration of Python and C++ for computationally intensive tasks while providing an engaging user experience, with clear paths for future enhancements such as resolving outstanding bugs and exploring online multiplayer capabilities.

10. Appendix – Tutorials

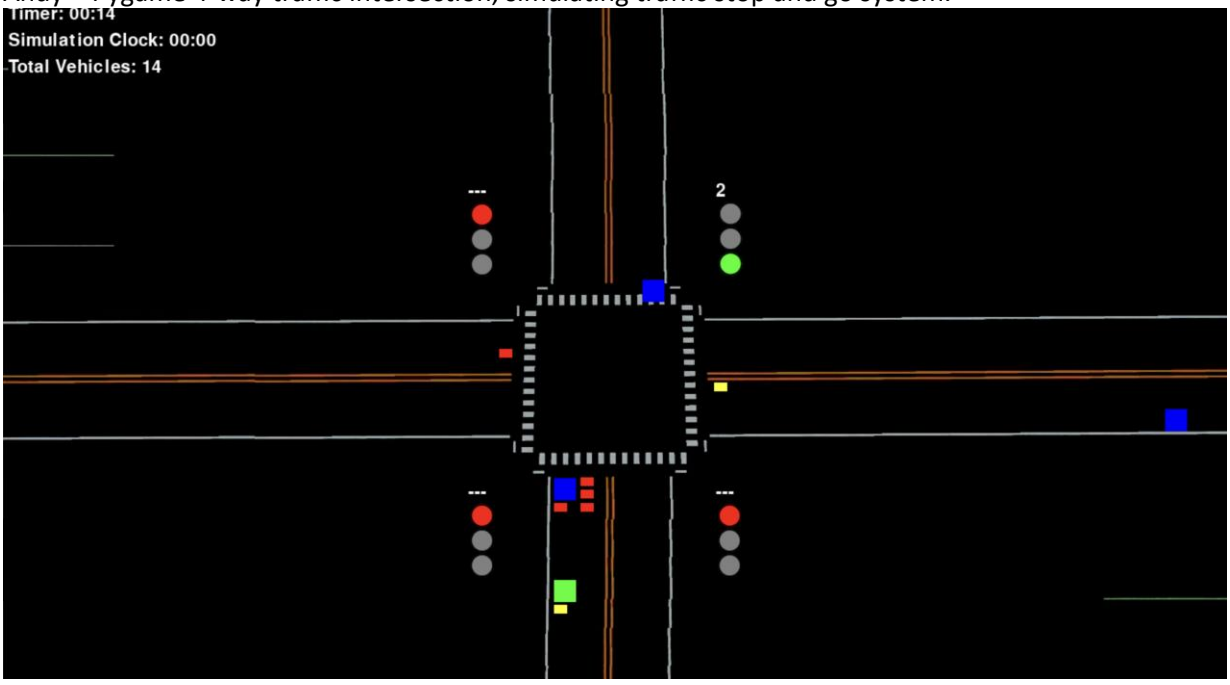
Bailey: PyGame moving circle that creates a trail as it moves. I also created the React website and game.



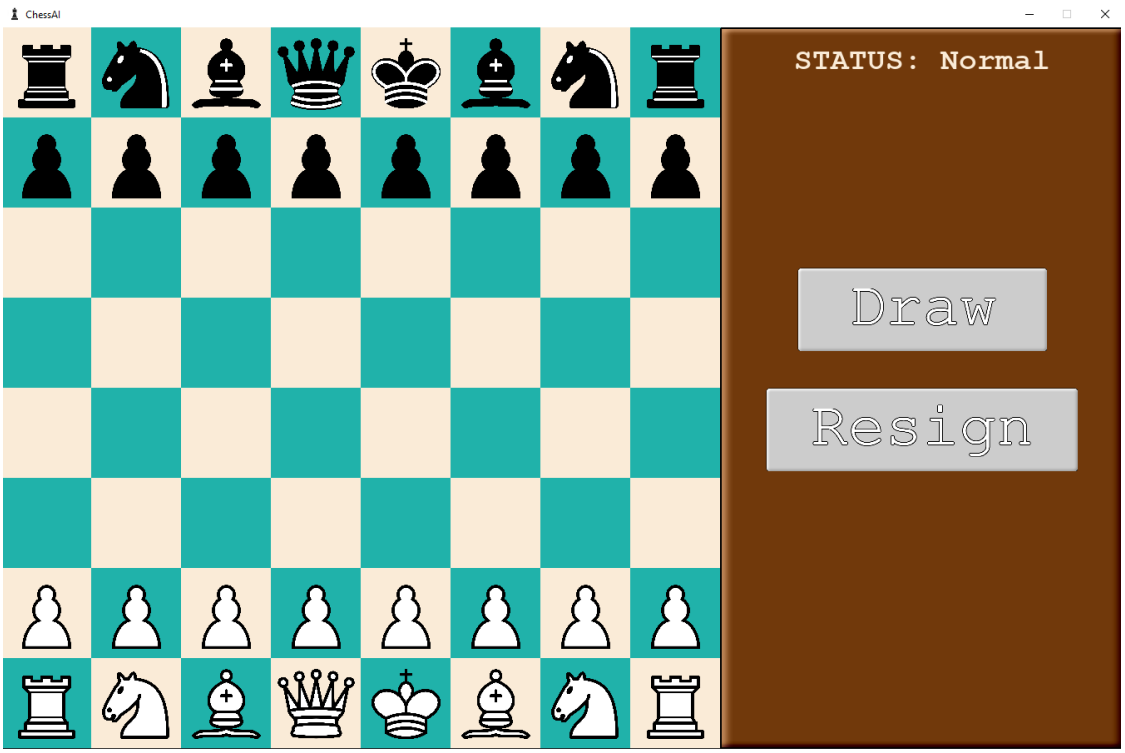
Alec
I made this basic Pygame application that draws a shape and lets you move it around with keyboard input. I also contributed to the chess UI, particularly the input handling



Andy – Pygame 4-way traffic intersection, simulating traffic stop and go system.



Carter - Pygame Chess game User Interface, allowing player to interact with game board


ChessAI
STATUS: Normal
Draw
Resign