Faculty of Engineering
Department of Electronics and Electrical Communications
ELC2080 Project

Advanced Tic-Tac-Toe
Performance Measurement Document

Name	ID
Ziad Mostafa	9231055
Ziad Emad	9230401
Ziad Abdel Hafeez	9230399
Sandra Atef	9230428
Robir Tamer	9230378

Under the supervision of:

Dr. Omar Ahmed Nasr

Table of Contents

1 Document Release History

#	Version No.	Release Date	Prepared By	Reviewed By	Preparation Comments
1	V1.0	17/6/2024	Robir Tamer	Sandra Atef	Initial Publication

2 Authors, Reviewers and Stakeholders’

Client Stakeholders Register

#	Name	Role in the project	Influence	Contact	Email
1	Dr. Omar Ahmed Nasr	Program Manager	SPOC	01143941832	omaranasr.phone@gmail.com

3 Introduction

This document outlines the key aspects of the Professional Tic-Tac-Toe application, targeting a broad audience including:

- **Developers**
- **Project Managers**
- **Users**

It aims to provide a comprehensive understanding of the application's design, functionality, and underlying principles.

3.1 Background

The classic Tic-Tac-Toe game, traditionally played with pen and paper, has evolved into a digital format to meet the growing demand for convenient and accessible entertainment. This project addresses this need by developing a modern and engaging digital Tic-Tac-Toe experience.

3.2 Purpose

The primary purpose of this application is to offer an advanced Tic-Tac-Toe experience that enhances user engagement. It allows players to create accounts, track game history, and challenge a strategic AI opponent. The features detailed within this document guide the development towards a user-friendly and captivating gameplay experience.

3.3 Scope

The scope of the Professional Tic-Tac-Toe project encompasses the creation of an interactive game application with a user-friendly Graphical User Interface (GUI), secure user management, and integrated Artificial Intelligence (AI) capabilities. The application is designed to be robust and scalable, providing a seamless experience for all users.

3.4 Definitions, Acronyms, and Abbreviations

To ensure clarity and consistency throughout this document, the following terms and acronyms are defined:

Term	**Definition**
AI	Artificial Intelligence
GUI	Graphical User Interface
CI/CD	Continuous Integration/Continuous Deployment
UX /UI	User Experience/ User Interface
OS	Operating System

3.5 Overview

This project aims to provide users with a modern and user-friendly platform to experience the enduring strategy and challenge of Tic-Tac-Toe. The application offers a captivating and convenient way to test skills, challenge friends, family, or an AI opponent, and enjoy the timeless fun of Tic-Tac-Toe.
Key Features:

- **Classic Gameplay:** Enjoy the timeless appeal of Tic-Tac-Toe with a familiar 3x3 grid and intuitive turn-based mechanics.
- **Multiple Playstyles:** Challenge others in head-to-head battles or test strategic prowess against a sophisticated AI opponent.
- **Streamlined Interface:** The intuitive and clean interface fosters a smooth user experience for players of all ages and skill levels.
- **Personalized experience:** Depending on specific product requirements, features like user profile creation and personalized game history tracking can be included to enhance the overall experience.

This Professional Tic-Tac-Toe application is designed to provide a stimulating and convenient way to enjoy the classic game. It encourages strategic thinking and offers an engaging platform for both casual and competitive play.

4 Methodology

Delivering a smooth and responsive user experience is paramount for the success of the Professional Tic-Tac-Toe application. This section outlines a comprehensive methodology for optimizing game performance, ensuring players enjoy a seamless and engaging gameplay experience.

4.1 Performance Metrics Measurement

The initial step involves establishing a performance baseline by identifying and measuring key metrics. These metrics serve as benchmarks for evaluating the effectiveness of optimization efforts. Here's a breakdown of this crucial phase:

4.1.1 Identifying Key Metrics

For the Professional Tic-Tac-Toe application, the following key metrics are crucial for performance evaluation:

- **CPU Utilization:** This metric measures the percentage of processing power utilized by the game at any given moment. High CPU usage can lead to lag and stuttering during gameplay, especially during AI decision-making or complex UI updates.
- **Memory Consumption:** This metric tracks the amount of RAM occupied by the
- game. Excessive memory usage can impact overall system performance, particularly during long gaming sessions or when handling multiple game states (e.g., game history).
- **Response Time:** This metric measures the time it takes for the game to respond to user actions, such as clicking on a board cell or the AI making a move. Slow response times can hinder user enjoyment and make the game feel unresponsive.

4.1.2 Baseline Measurement

Tools like system-level task managers (e.g., top or htop on Linux, Task Manager on Windows) or more advanced profiling suites (e.g., Valgrind for memory profiling, gprof for CPU profiling in C++ applications) can be employed to gather baseline performance data. This data serves as a reference point for evaluating the impact of optimization efforts. Measurements should be conducted during various gameplay scenarios, encompassing single-player (vs. AI) and multiplayer modes, different AI difficulty levels, and during transitions between different screens (e.g., login, game, statistics).

4.1.3 Data Collection

Continuously monitoring the selected metrics throughout different stages of gameplay helps us understand resource usage patterns and pinpoint potential bottlenecks for further analysis and optimization. Automated testing frameworks can be integrated to collect performance data systematically over time, allowing for trend analysis and regression detection.

4.2 Code Optimization Techniques

Once performance bottlenecks are identified through careful measurement, targeted code optimizations can be implemented to address them. This phase involves a combination of algorithmic improvements and resource management strategies, specifically tailored to the Professional Tic-Tac-Toe application:

4.2.1 Algorithm Optimization

The core of the AI opponent in Tic-Tac-Toe often relies on search algorithms. The provided code structure suggests the use of an AIOpponent class, which likely implements a variant of the Minimax algorithm

4.2.1.1 Minimax Algorithm

The Minimax algorithm, commonly employed by AI opponents in games like Tic-Tac-Toe, explores all possible game states to determine the most optimal move. For this application, optimization of the Minimax algorithm can involve:

- **Memorization Techniques**: Storing previously calculated game states and their corresponding optimal moves (e.g., using a transposition table or hash map) can avoid redundant computations, especially in games with repetitive states. This is particularly effective for the GameLogic and AIOpponent classes.
- **Heuristics**: Employing heuristics to prioritize promising moves and reduce the search space can significantly speed up decision-making, potentially sacrificing some optimality for faster response times. This would involve evaluating board states more efficiently without exploring every single branch..

4.2.1.2 Alpha-Beta Pruning

Alpha-Beta Pruning is a powerful optimization technique for Minimax. It eliminates branches in the Minimax decision tree that are demonstrably suboptimal, significantly reducing the search space and accelerating AI move selection. Implementing this within the AIOpponent class would drastically improve AI performance, especially at higher difficulty levels.

4.2.2 Resource Management

Efficient resource management is crucial for maintaining low memory consumption and CPU utilization.

- **Memory Management:** Memory leaks can occur when allocated memory is not properly released after use. Implementing memory management best practices, such as employing appropriate data structures and adhering to proper memory allocation and deallocation routines, ensures efficient memory utilization. For C++ applications like this, careful use of smart pointers and understanding object lifetimes within classes like Authentication , GameHistory , and Database is essential.
- **Data Structure Selection:** The choice of data structures for storing and manipulating game data significantly impacts performance. Carefully selecting efficient data structures, such as QVector for dynamic arrays, QHash for fast lookups (e.g., for user authentication or game history), or QList for linked lists, can minimize processing overhead and improve access times. For instance, the GameLogic board (QVector<Player> m_board) is a good choice for a fixed size grid, but for more complex data, other structures might be more suitable.

4.3 Profiling and Testing:

Profiling tools are valuable allies in the performance optimization process. These tools help identify sections of code that consume a disproportionate amount of resources, allowing us to focus optimization efforts on these specific areas. Here's how profiling and testing contribute to achieving optimal performance:

4.3.1 Profiling Tools

Employing profiling tools during gameplay allows us to pinpoint performance hotspots within the codebase. For C++ and Qt applications, tools like Qt Creator's built-in profiler, Valgrind (for memory and CPU profiling), or perf (Linux performance counter) can be used. By focusing optimization efforts on these specific areas, we can maximize their impact and ensure the most significant performance gains. For example, profiling might reveal that the Database interactions or GameHistory logging are consuming more resources than expected.

4.3.2 Automated Testing

Implementing a suite of automated performance tests is crucial for ongoing performance monitoring. These tests should be designed to regularly assess the impact of code changes on performance metrics. This ensures that optimizations are maintained, and regressions are identified swiftly. Unit tests (e.g., using Qt Test framework, as suggested by the tests directory in the provided code) for Game Logic , AI Opponent , Authentication, and Database classes can include performance assertions. Integration tests can simulate full game scenarios and measure end-to-end response times and resource usage.

5 Detailed Analysis

This section provides a detailed analysis of the expected performance characteristics of the Professional Tic-Tac-Toe application, focusing on memory usage, CPU utilization, and response time. Since actual performance data is not available at this stage, the analysis is based on typical behaviors of Qt-based applications and the logical structure of the provided C++ code

5.1 Memory Usage and CPU utilization

5.1.1 Results

Event	Memory Usage	CPU Utilization
Just started	14.3 MB	0.45%
After opened	50.2 MB	1.3%
While playing	115.7 MB	3.8%
Changing the frame (login, choosing mode, history, etc.)	190.4 MB	5.8%
After saving	240.6 MB	2.0%

Note: These figures are estimates and can vary significantly based on the operating system, hardware, and specific Qt version.

5.1.2 Discussion of Results

Upon application startup, a baseline memory footprint is established, primarily due to the Qt framework, application resources (images, stylesheets), and initial object instantiations (e.g., MainWindow , Authentication , GameLogic). The CPU utilization is minimal during idle states.

When the application is fully opened and the main window is displayed, memory usage will increase as all UI elements are loaded and initialized. CPU usage remains low unless there are active animations or background processes.

During gameplay, especially in Player vs. AI mode, CPU utilization is expected to rise. The AIOpponent class, particularly if implementing a complex Minimax algorithm with Alpha-Beta Pruning, will consume CPU cycles during its decision-making process. The complexity of the AI algorithm directly impacts CPU usage; higher difficulty levels (e.g.,

Expert) will likely involve deeper search trees and thus higher CPU usage. Memory usage will also increase slightly to store game states, history, and potentially memoization tables for the AI.

Changing between different screens (e.g., login, game, statistics) will cause minor fluctuations in memory as UI elements are shown/hidden or new widgets are loaded. Brief CPU spikes are expected during these transitions as the UI is redrawn and layouts are recalculated.

5.2 Response time

5.2.1 Results

Event	Expected Response Time
UI Element Click	< 50 ms
AI Move (Easy)	< 100 ms
AI Move (Medium)	100-300 ms
AI Move (Hard)	300-800 ms
AI Move (Expert)	800-2000 ms

Note: These figures are estimates and can vary significantly based on the operating system, hardware, and specific Qt version.

5.2.2 Discussion of results

User interface (UI) element clicks (e.g., on board cells, buttons) are expected to have very low response times, typically under 50 milliseconds, ensuring a fluid and responsive user experience. This is critical for a game where immediate feedback is expected.

AI move response time is highly dependent on the chosen difficulty level. For

easy and medium difficulties, the AI should respond almost instantaneously. However, for hard and expert difficulties, where the AI performs deeper searches (Minimax with Alpha-Beta Pruning), the response time will increase. The goal is to keep these response times within acceptable limits (e.g., under 2 seconds for expert AI) to avoid frustrating the user.

Login and registration operations, handled by the Authentication class, should be fast, typically under 200 milliseconds. This includes hashing passwords and potentially interacting with a local user database. Similarly, database save and load operations, managed by the Database class, should also be efficient, ideally under 500 milliseconds, to ensure quick persistence and retrieval of user data and game history.

Overall, the response times are critical for user satisfaction. Optimizations in the AIOpponent and Database classes will be key to achieving the desired performance targets.

6 Optimization

This section details various optimization techniques that can be applied to the Professional Tic-Tac-Toe application to enhance its performance, drawing inspiration from general software optimization principles and specific strategies applicable to game development and C++/Qt environments:

6.1 Memorization Techniques

Memorization, or memoization, is a powerful optimization technique used to speed up computer programs by caching the results of expensive function calls and returning the cached result when the same inputs occur again. In the context of the Tic-Tac-Toe AI, this is particularly relevant for the AIOpponent and GameLogic classes.

For the Minimax algorithm, which often re-evaluates the same game states multiple times during its search, storing the computed optimal move and score for each visited board state can significantly reduce redundant calculations. A QHash or std::map could be used to store (board_state, optimal_score) pairs. Before initiating a new search for a given board state, the AI would first check if the result is already in the cache. This approach is highly effective for games with a finite and relatively small number of reachable states, like Tic-Tac-Toe.

6.2 Heuristics

Heuristics are problem-solving approaches that employ a practical method not guaranteed to be optimal or perfect, but sufficient for reaching an immediate, short-

term goal. In AI game development, heuristics are used to guide the search process and make decisions more quickly, especially when a full search of the game tree is computationally infeasible.

For the AIOpponent , heuristics can be used to evaluate the

desirability of a board state without fully exploring all subsequent moves. For example, a simple heuristic could assign scores based on the number of two-in-a-row opportunities, blocking opponent wins, or controlling the center square. While this might sacrifice some optimality compared to a full Minimax search, it can drastically reduce the AI's response time, making the game feel more fluid, especially for lower difficulty settings.

6.3 Alpha-Beta Pruning Branch Elimination

Alpha-Beta Pruning is a search algorithm optimization technique for Minimax. It reduces the number of nodes evaluated in the search tree by eliminating branches that cannot possibly influence the final decision. This is a crucial optimization for the AIOpponent class, particularly for higher difficulty levels.

By maintaining two values, alpha (the best value that the maximizer currently can guarantee at that level or above) and beta (the best value that the minimizer currently can guarantee at that level or above), the algorithm can

prune branches when the current node's value is worse than the alpha value (for minimizer) or beta value (for maximizer). This significantly reduces the computational cost without affecting the final outcome of the Minimax search.

6.4 Data Structures

The choice of data structures can profoundly impact the performance of an application, especially in terms of memory usage and access speed. For the Professional Tic-Tac-Toe application, optimizing data structure usage is key.

- **Game Board Representation:** The QVector<Player> m_board in GameLogic is a suitable choice for a 3x3 grid. However, for larger boards or more complex game states, a more compact representation (e.g., bitboards) could be considered to reduce memory footprint and enable faster bitwise operations for checking win conditions.
- **User and Game History Storage:** The Authentication and GameHistory classes likely use QVector<User*> and QListWidget respectively. For efficient lookup and storage of user data and game records, especially if the number of users or games grows, consider using QHash or QMap for user management (mapping usernames to User objects) and a more optimized database schema for GameHistory if a persistent database is used.

6.5 UI Optimization

User Interface (UI) performance is critical for a smooth user experience. Qt provides various mechanisms for optimizing UI responsiveness.

- **Efficient Painting:** Minimize redundant repaints. Use update() instead of repaint() when possible, as update() schedules a paint event, allowing Qt to optimize multiple updates into a single paint operation. Custom painting in LeaderboardItemDelegate and HistoryItemDelegate should be optimized to draw only what is necessary.
- **Layout Management:** Complex layouts can be computationally expensive. Ensure that layouts are not excessively nested and that sizeHint() and minimumSizeHint() are correctly implemented for custom widgets to help Qt's layout engine work efficiently.
- **Animations:** While animations enhance user experience, they can consume CPU resources. The setupTitleAnimation function should use QPropertyAnimation or QVariantAnimation for smooth, efficient animations that leverage Qt's animation framework.
- **Resource Loading:** Loading resources like images (spinner.gif) and stylesheets (main.qss) should be done efficiently. Using Qt's resource system (.qrc files) helps embed resources directly into the executable, improving loading times compared to external files.

6.6 Database Interaction

The Database class handles persistence of user data and game history. Optimizing database interactions is crucial for application responsiveness.

- **Asynchronous Operations:** For potentially long-running database operations (e.g., loading all game history, saving large amounts of data), consider performing these operations in a separate thread using QThread or QtConcurrent to prevent blocking the UI thread. This ensures the application remains responsive during data access.
- **Batch Operations:** Instead of performing individual insert/update operations in a loop, use batch operations (e.g., INSERT ... VALUES (...), (...);) to reduce the overhead of multiple database calls.
- **Indexing:** Ensure appropriate indexes are created on database tables (e.g., on username for User table, on user IDs and timestamps for GameHistory) to speed up data retrieval queries.
- **Connection Management:** Efficiently manage database connections. Open connections when needed and close them when no longer in use, or use a connection pool for frequently accessed databases.

6.7 Resource Management

Beyond memory management for data structures, overall resource management is vital.

- **File I/O:** Minimize file I/O operations, especially synchronous ones, as they can block the UI. For user data and game history, consider in-memory caching for frequently accessed data to reduce disk access.
- **Network Operations:** If the application were to expand to include online multiplayer, network operations would need careful optimization, including asynchronous communication and efficient data serialization.

6.8 Event Handling

Qt's event system is central to its operation. Efficient event handling contributes to a responsive application.

- **Signal and Slot Connections:** Ensure that signal-slot connections are efficient. Direct connections are faster but can block the sender's thread; queued connections are safer for cross-thread communication but introduce overhead. Use the appropriate connection type based on the context.
- **Minimize Event Processing:** Avoid computationally intensive operations directly within slots connected to frequent signals (e.g., onCellClicked). Delegate heavy processing to worker threads if necessary

.

6.9 Code Refactoring

Regular code refactoring can improve maintainability, readability, and performance.

- **Modularity:** Ensure classes like Authentication , GameLogic , GameHistory , AIOpponent , and Database are highly modular and have clear responsibilities. This facilitates independent optimization and testing.
- **Code Duplication:** Eliminate redundant code to reduce codebase size and improve maintainability. For example, common utility functions could be extracted.
- **Algorithmic Complexity:** Continuously review algorithms for their time and space complexity. Even small changes in algorithmic choice can lead to significant performance improvements for larger inputs or more complex scenarios.

- **Profiling-Driven Refactoring:** Use profiling results to guide refactoring efforts. Focus on optimizing the identified hotspots rather than prematurely optimizing less critical parts of the code.

7 Appendix

7.1 Refrenced Analysis Main Graphs

This section would typically include visual representations of the performance analysis, such as graphs for memory usage, CPU utilization, and response times. These graphs provide a clear and concise overview of the application's performance characteristics under various conditions.

7.1.1 Memory Usage

consumption (in MB) over time, possibly segmented by different application states (e.g., idle, playing vs. AI, playing vs. Player, navigating menus). This graph would illustrate the memory footprint and identify any potential memory leaks or spikes.

7.1.2 CPU Utilization

Placeholder for CPU Utilization Graph: A line graph depicting CPU utilization (in percentage) over time, highlighting periods of high computational activity, such as AI decision-making or complex UI rendering. This graph would help pinpoint CPU bound operations.

7.2 Detailed References

[1] C. Dr. Omar Nasr, "Advanced Tic Tac Toe Game," *Detailed Project Description,* April 2024.

[2] "Github," [Online]. Available: https://google.github.io/googletest/.

[3] "Qt_studio_GUI_index," [Online]. Available: https://doc.qt.io/qt-6/qtgui-index.html.

[4] "Finding optimal move in Tic-Tac-Toe using Minimax Algorithm in Game Theory," 20 Feb 2023. [Online]. Available: https://www.geeksforgeeks.org/finding-optimal-move-in-tic-tac-toe-using-minimax-algorithm-in-game-theory/.

[5] "SQLite Official Documentation," [Online]. Available: https://www.sqlite.org/docs.html.

[6] "Secure Hashing algorithms," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Cryptographic_hash_function.

[7] "User Interface Design Basics," [Online]. Available: https://www.interaction-design.org/literature/topics/ui-design.