# Software Design Specification (SDS)

# 1. Introduction

## 1.1 Purpose

This document describes the software design specification for the Tic-Tac-Toe game project. The goal of this project is to develop a desktop application that allows users to play Tic-Tac-Toe against an AI opponent or their friends, track their game history, and view statistics for each user.

## 1.2 Scope

The scope of this document includes the detailed design and architecture of the Tic-Tac-Toe game, including user interface design, game logic, AI implementation, and database management.

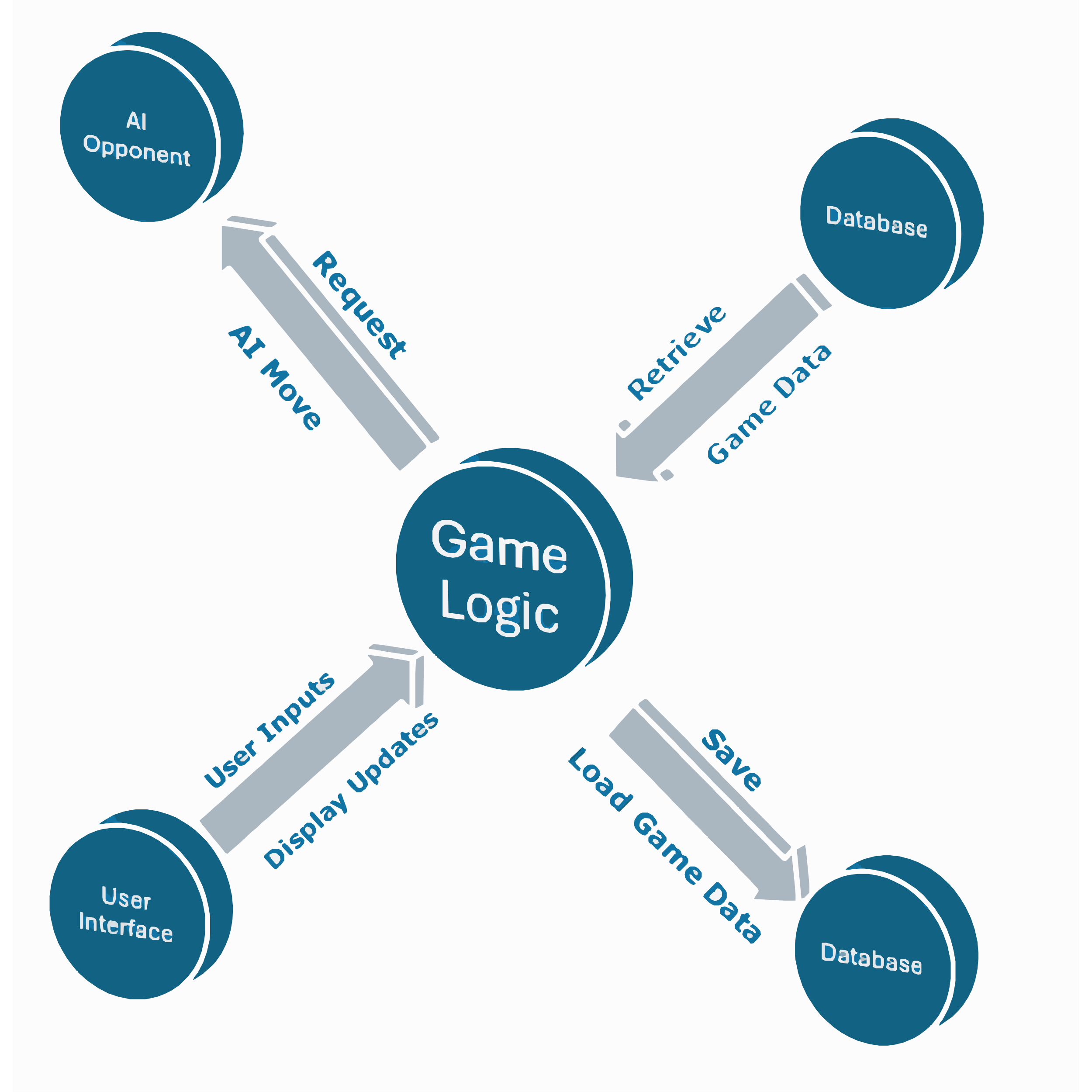
## 1.3 References

* Project requirements document
* Qt Tutorial Videos
* SQLite Tutorial Videos
* Chat GPT

# 2. System Overview

## 2.1 System Architecture

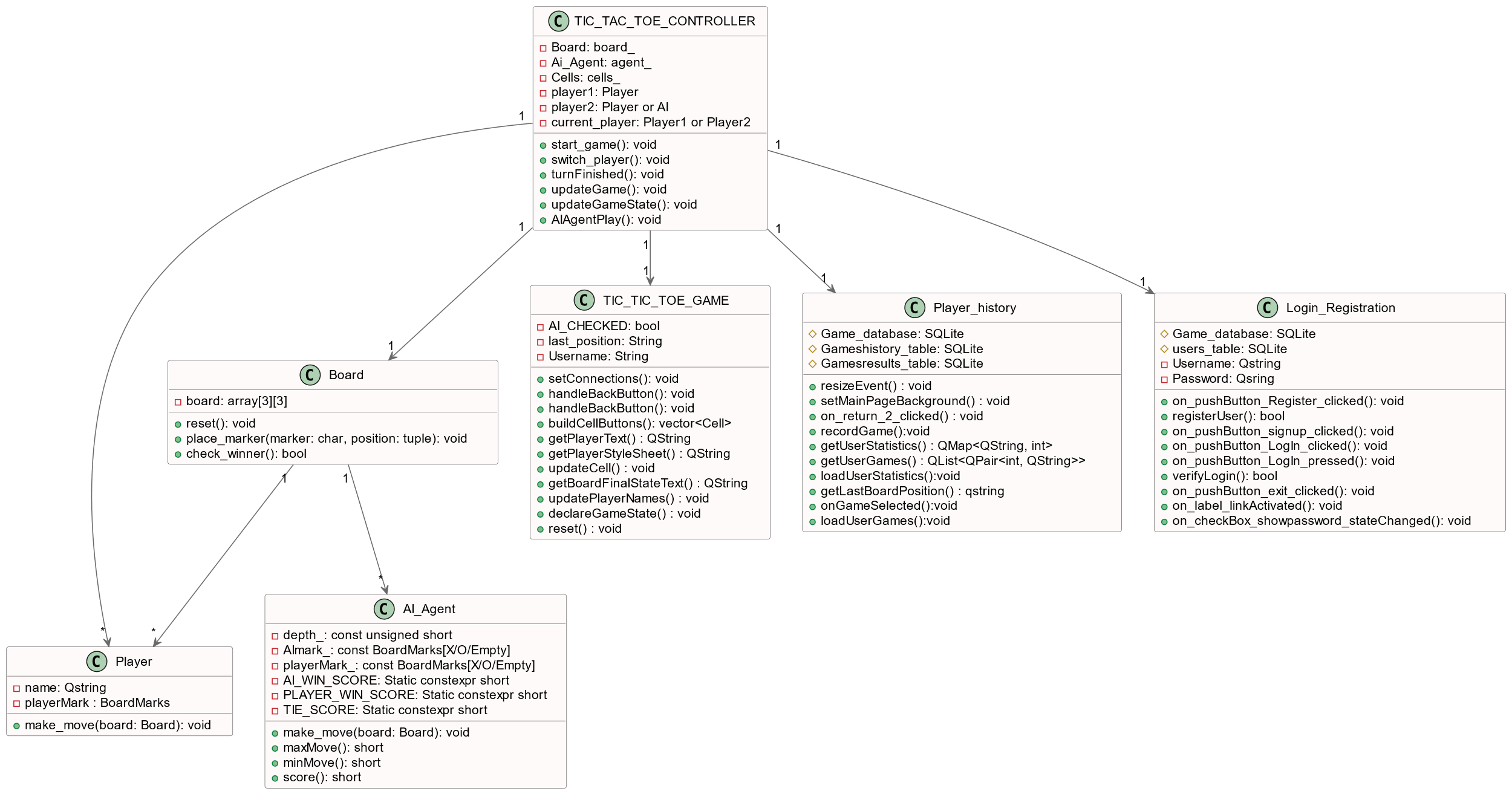
The system is composed of the following components:

* User Interface: Provides an interface for users to interact with the game.
* Game Logic: Manages the rules and state of the game.
* AI Opponent: Implements the AI to play against the user.
* Database: Stores user information, game history, and statistics.

## 2.2 High-level Architecture Diagram

# 3. Detailed Design

## 3.1 Class Diagram



## 3.2 Sequence Diagram

A diagram of a computer

Description automatically generated

## 3.3 User Flow Diagram

# 4. Component Specifications

## 4.1 Game Logic

Tic-Tac-Toe is a classic two-player game played on a 3x3 grid. The game begins with an empty board where players take turns marking cells with their respective symbols: 'X' for the first player and 'O' for the second player. The objective is to be the first to align three of their symbols horizontally, vertically, or diagonally.

## 4.2 AI Opponent

The AI opponent in the Tic-Tac-Toe game utilizes the **Minimax algorithm** improved by alpha-beta optimization to ensure optimal decisions. This method ensures that the AI evaluates all possible moves, balancing between maximizing its chances of winning and minimizing the player's chances of winning.

**Key Components:**

1. **Minimax Algorithm:**
   * The maxMove function aims to maximize the AI's score, while the minMove function seeks to minimize the player's score. This is implemented through recursive calls, evaluating all possible game states.

short MiniMaxAgent::maxMove(Board &board, unsigned short depth, short alpha, short beta) const

{

BoardState state = board.evaluateBoard();

if (depth == 0 || state != BoardState::NoWinner)

return score(state);

short bestScore = INT\_MIN;

for (size\_t row = 0; row < board.size(); ++row) {

for (size\_t col = 0; col < board.size(); ++col) {

if (board.at(row, col) == BoardMarks::Empty) {

board.setPlayerInput(row, col, AImark\_);

short score = minMove(board, depth - 1, alpha, beta);

board.resetCell(row, col);

bestScore = std::max(bestScore, score);

alpha = std::max(alpha, score);

if (beta <= alpha)

break;

}

}

}

return bestScore; }

short MiniMaxAgent::minMove(Board &board, unsigned short depth, short alpha, short beta) const

{

BoardState state = board.evaluateBoard();

if (depth == 0 || state != BoardState::NoWinner)

return score(state);

short bestScore = INT\_MAX;

for (size\_t row = 0; row < board.size(); ++row) {

for (size\_t col = 0; col < board.size(); ++col) {

if (board.at(row, col) == BoardMarks::Empty) {

board.setPlayerInput(row, col, playerMark\_);

short score = maxMove(board, depth - 1, alpha, beta);

board.resetCell(row, col);

bestScore = std::min(bestScore, score);

beta = std::min(beta, score);

if (beta <= alpha)

break;

}

}

}

return bestScore; }

1. **Score Evaluation:**
   * The score function assigns values based on the game state, with higher scores for wins and lower scores for losses.

short MiniMaxAgent::score(const BoardState state) const

{

if (AImark\_ == BoardMarks::X && state == BoardState::XWins)

return AI\_WIN\_SCORE;

if (AImark\_ == BoardMarks::O && state == BoardState::OWins)

return AI\_WIN\_SCORE;

if (playerMark\_ == BoardMarks::X && state == BoardState::XWins)

return PLAYER\_WIN\_SCORE;

if (playerMark\_ == BoardMarks::O && state == BoardState::OWins)

return PLAYER\_WIN\_SCORE;

return TIE\_SCORE;

}

1. **AI Decision Making (play Function):**
   * The play function orchestrates the AI's move selection by evaluating all possible moves and choosing the one with the best score.

int MiniMaxAgent::play(Board &board)

{ int bestScore = INT\_MIN;

QPair<size\_t, size\_t> bestEntry;

for (size\_t row = 0; row < board.size(); ++row) {

for (size\_t col = 0; col < board.size(); ++col) {

if (board.at(row, col) == BoardMarks::Empty) {

board.setPlayerInput(row, col, AImark\_);

int moveScore = minMove(board, depth\_ - 1, SHRT\_MIN, SHRT\_MAX);

if (moveScore > bestScore) {

bestScore = moveScore;

bestEntry.first = row;

bestEntry.second = col; }

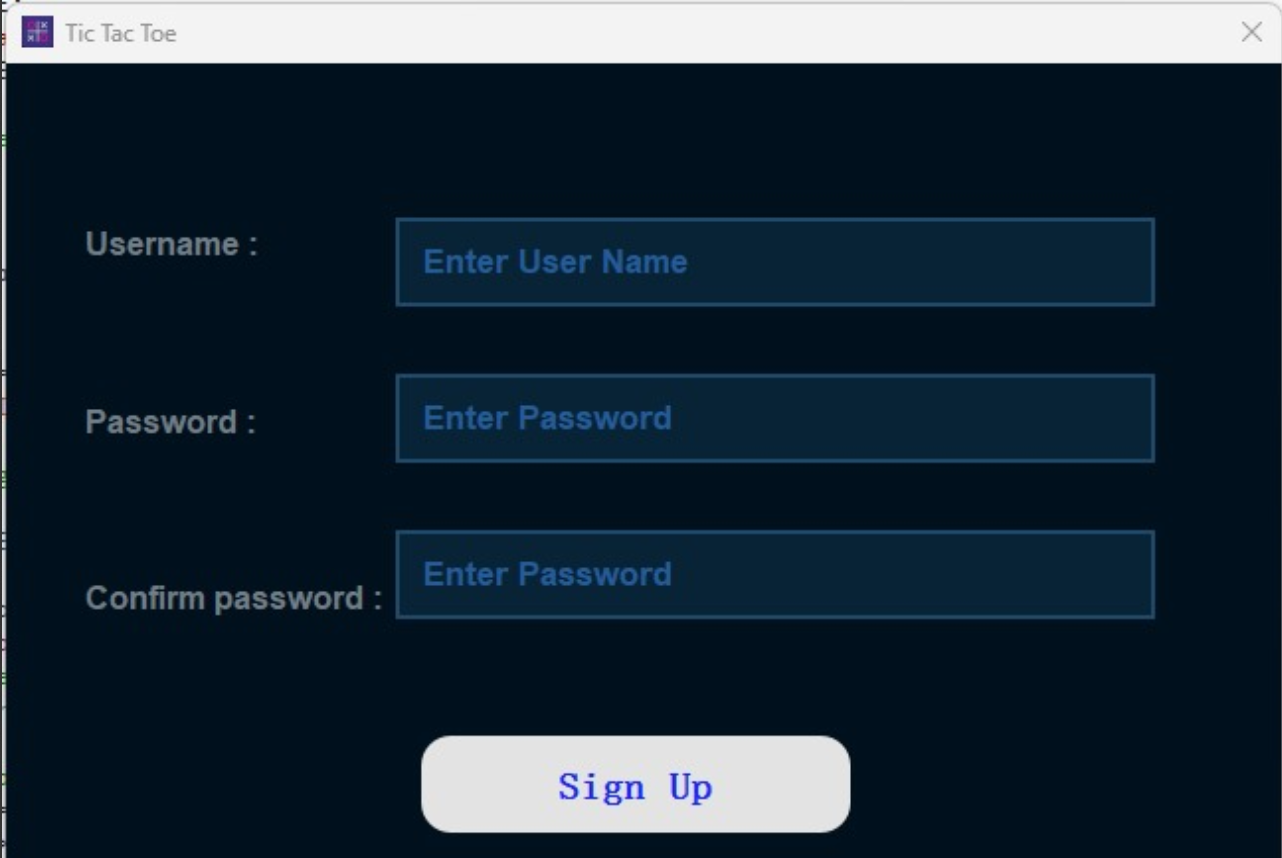
board.resetCell(row, col);

}}}

board.setPlayerInput(bestEntry.first, bestEntry.second, AImark\_);

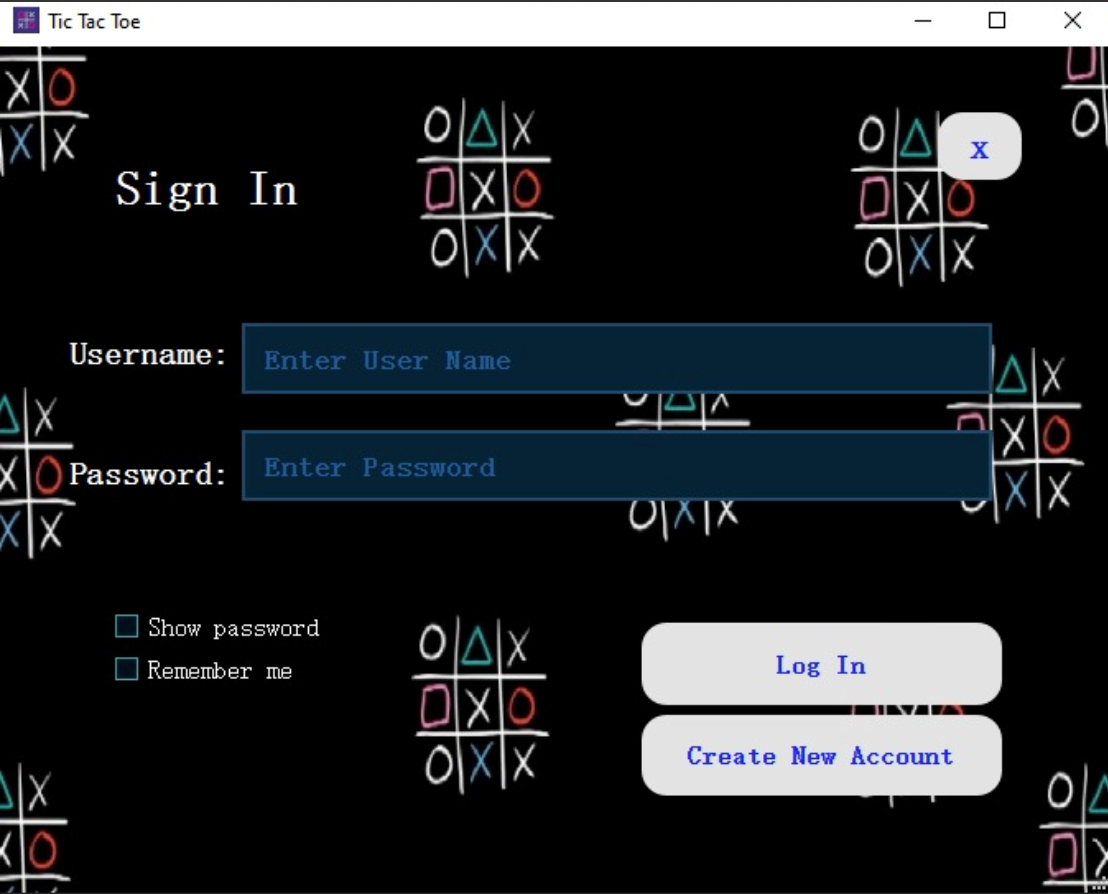
return static\_cast<int>(bestEntry.first \* board.size() + bestEntry.second);}

## 4.3 GUI Design

The graphical user interface (GUI) for the Tic-Tac-Toe project is designed to be intuitive and user-friendly, allowing players to easily navigate through different game modes, view their game history, and play the game. The GUI consists of several main screens: the main menu, the game board, the player history, and the profile page.

**Sign Up Page**

The Sign Up page allows new users to create an account by asking for new username and password and storing them in user's table in the database created.



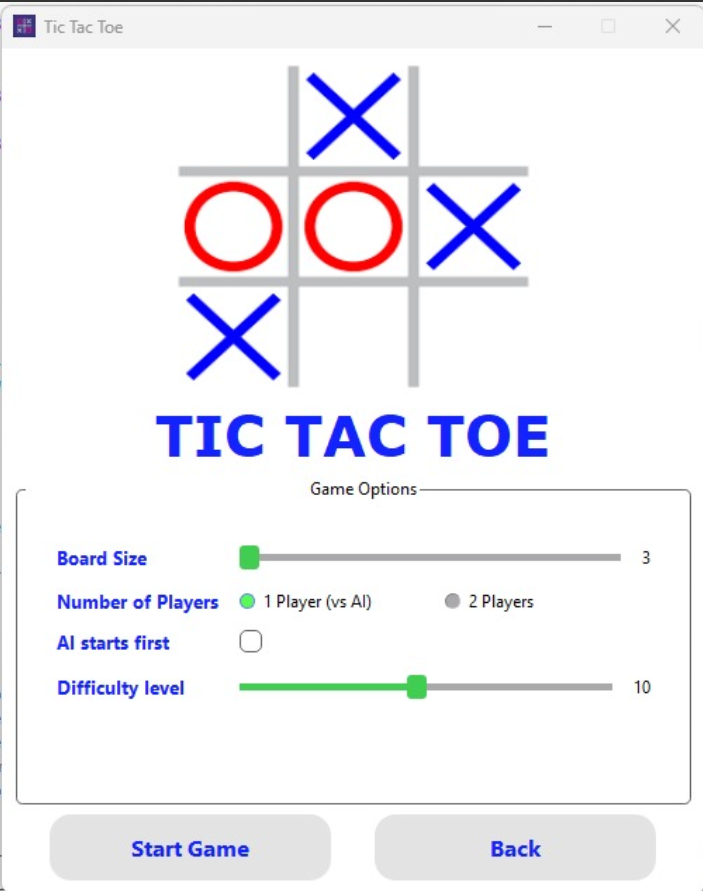
**Login Page**

The Login page is designed for user authentication so that each user has his own username and password and his game play history.



**Profile Page**

Redirects the user to one of three paths whether to the title screen to start playing a new or to the "Player's History" or to "How to play screen"

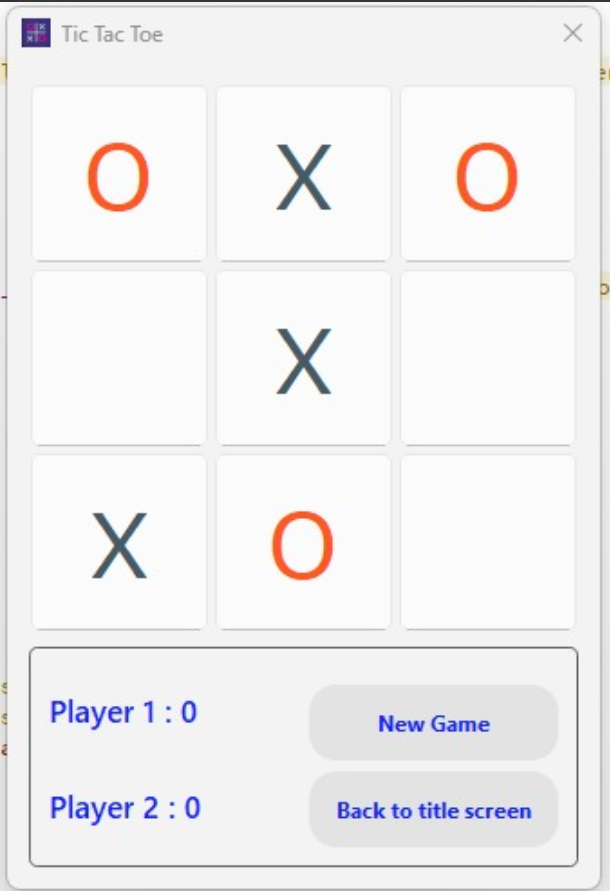


**Title Screen**

The Title Screen allows the user to choose whether to play against AI or one of his friends. It also allows to choose the level you want to play against when facing

the AI. Another option is choosing the board size from 3x3 board to a 10x10 board.

**Tic Tac Toe game board**

The Game Board is the central interface for the Tic Tac Toe game where players interact to play the game.

It is designed with simplicity and clarity to provide an intuitive user experience.



**Player History**

The Player History feature is designed to provide users with a detailed record of their past games.

This functionality allows users to view their game outcomes, review past performances, and analyze their progress over time.

## 4.4 User Login & Registration

**User Registration:**

Registers new users securely by hashing passwords before storing them in the database.

Connects to SQLite, inserts username and hashed password into the 'users' table, and handles errors gracefully.

Code example for User registration in database:

bool signup::registerUser(const QString& username, const QByteArray& passwordHash)

{

if (!m\_db.isOpen()) {

qDebug() << "Database connection is not open.";

return false;

if (username.isEmpty() || passwordHash.isEmpty()) {

qDebug() << "Username or password hash is empty.";

return false;}

QSqlQuery query(m\_db);

if (!m\_db.transaction()) {

qDebug() << "Failed to start database transaction.";

return false;}

query.prepare("INSERT INTO users (username, password) VALUES (:username, :password)");

query.bindValue(":username", username);

query.bindValue(":password", passwordHash);

if (!query.exec()) {

qDebug() << "Error executing insert query:" << query.lastError().text();

m\_db.rollback();

return false;}

if (!m\_db.commit()) {

qDebug() << "Failed to commit database transaction.";

m\_db.rollback();

return false;}

qDebug() << "User registered successfully!";

return true;}

**Login Verification:**

Verifies user login credentials by comparing input password with the hashed password stored in the database.

Connects to SQLite, retrieves hashed password for the provided username, performs comparison, and returns login success or failure.

Ensures secure password handling to protect user credentials.

Code example for User login verification:

bool MainPage::verifyLogin(const QString& username, const QString& password)

{

if (!m\_db.isOpen()) {

qDebug() << "Database connection is not open.";

return false;}

QSqlQuery query(m\_db);

query.prepare("SELECT password FROM users WHERE username = :username");

query.bindValue(":username", username);

if (!query.exec() || !query.next()) {

qDebug() << "Error executing login query:" << query.lastError().text();

return false;}

// Retrieve the hashed password from the database

QByteArray storedPasswordHash = query.value(0).toByteArray();

// Hash the entered password using SHA-256

QByteArray enteredPasswordHash = QCryptographicHash::hash(password.toUtf8(), QCryptographicHash::Sha256).toHex();

// Compare the hashed passwords

if (enteredPasswordHash == storedPasswordHash) {

qDebug() << "Login successful!";

return true;}

else {

qDebug() << "Invalid username or password.";

return false;}}

## 4.5 Game History

The game history feature in our Tic Tac Toe project allows users to review their past games, providing valuable insights into their gameplay over time.

This section details how game histories are stored and managed in the data base, including the format and retrieval of past game data.

Game history is stored in an SQLite database, which ensures efficient and reliable data management. The database includes the following tables:

1. Users: Stores user credentials(username and password).

2. Games: Stores individual username, including the game ID, associated timestamp, game result(Win-lose-tie), and the final board position.

3. Game\_results: Stores cumulative statistics for each user, such as the total number of wins, losses, and draws.

**The following code snippet shows how to create the tables to store the users played matches:**

query.exec("CREATE TABLE IF NOT EXISTS users (username TEXT PRIMARY KEY, password TEXT)");

query.exec("CREATE TABLE IF NOT EXISTS games ("

"game\_id INTEGER PRIMARY KEY AUTOINCREMENT,"

"username TEXT,"

"result TEXT,"

"last\_board\_position TEXT,"

"timestamp DATETIME DEFAULT CURRENT\_TIMESTAMP,"

"FOREIGN KEY (username) REFERENCES users(username))");

query.exec("CREATE TABLE IF NOT EXISTS game\_results ("

"username TEXT PRIMARY KEY,"

"wins INTEGER DEFAULT 0,"

"losses INTEGER DEFAULT 0,"

"draws INTEGER DEFAULT 0,"

"FOREIGN KEY (username) REFERENCES users(username))");

**Data Retrieval**

To retrieve game histories, the application queries the games table based on the username.

The results are then displayed in the user interface, allowing users to select and view detailed information about each game.

**Example Code for Retrieving User Games**

QList<QPair<int, QString>> Playerhistory::getUserGames(QString &username) {

QList<QPair<int, QString>> games;

QSqlQuery query(m\_db);

query.prepare("SELECT game\_id, result FROM games WHERE username = :username");

query.bindValue(":username", username);

if (!query.exec()) {

qDebug() << "Error retrieving user games:" << query.lastError().text();

return games; }

while (query.next())

{

int gameId = query.value("game\_id").toInt();

QString result = query.value("result").toString();

games.append(qMakePair(gameId, result));

}

return games;

}

**Displaying Game History**

Once the game data is retrieved, it is displayed to the user via a combo box.

Selecting a game from this list triggers the retrieval of the game's final board position, which is then displayed on the board.

**Example Code for Displaying Game History**

void Playerhistory::loadUserGames(QString &username) {

ui->gameComboBox->clear();

auto games = getUserGames(username);

for (const auto &game : games) {

ui->gameComboBox->addItem(QString("Game %1 (%2)").arg(game.first).arg(game.second), game.first);}}

void Playerhistory::loadUserStatistics( QString &username) {

auto stats = getUserStatistics(username);

ui->winsLabel->setText(QString::number(stats["wins"]));

ui->lossesLabel->setText(QString::number(stats["losses"]));

ui->drawsLabel->setText(QString::number(stats["draws"]));

}

void Playerhistory::onGameSelected(int index) {

if (index == -1) return; // No selection

int gameId = ui->gameComboBox->currentData().toInt();

QString lastBoardPosition = getLastBoardPosition(gameId);

LastPositionDisplay \*LastPositionDisplay = new LastPositionDisplay();

LastPositionDisplay->show();

LastPositionDisplay->updateBoard(lastBoardPosition);

}

# 5. Database Design

## 5.1 Database Tables

**Users Table:**

A screenshot of a computer

Description automatically generatedStores the username and password (Hashed) so that they are used for login Verification.

**Games results Table:**

Stores for each user using his username the total wins, losses, and draws:

A screenshot of a computer

Description automatically generated**Games Table:**

A screenshot of a computer

Description automatically generatedStores for each game played the Game ID, Username, Result, Last Position, Time Stamp:

## 5.2 Data Retrieval

Retrieval occurs in two distinct contexts:

1. **Login Verification:** User credentials (username and hashed password) are retrieved from the 'users' table using SQL queries based on the provided username. This ensures secure authentication during the login process, validating user access to the game system.

**Code snippet for retrieving the username and password for login verification:**

if (!m\_db.isOpen()) {

qDebug() << "Database connection is not open.";

return false;}

QSqlQuery query(m\_db);

query.prepare("SELECT password FROM users WHERE username = :username");

query.bindValue(":username", username);

if (!query.exec() || !query.next()) {

qDebug() << "Error executing login query:" << query.lastError().text();

return false;}

1. **Game History Display:** Within the player history class, game histories are retrieved from the 'games' and 'game\_results' tables. This involves querying based on the user's ID to fetch specific game sequences, results, and timestamps. This retrieval enables users to review their past game performances and outcomes within the application interface, providing a comprehensive view of their gaming history.

QMap<QString, int> Playerhistory::getUserStatistics(QString &username) {

QMap<QString, int> stats;

QSqlQuery query(m\_db);

query.prepare("SELECT wins, losses, draws FROM game\_results WHERE username = :username");

query.bindValue(":username", username);

if (!query.exec()) {

qDebug() << "Error retrieving user statistics:" << query.lastError().text();

return stats;}

if (query.next()) {

stats["wins"] = query.value("wins").toInt();

stats["losses"] = query.value("losses").toInt();

stats["draws"] = query.value("draws").toInt();}

return stats;}

QList<QPair<int, QString>> Playerhistory::getUserGames(QString &username) {

QList<QPair<int, QString>> games;

QSqlQuery query(m\_db);

query.prepare("SELECT game\_id, result FROM games WHERE username = :username");

query.bindValue(":username", username);

if (!query.exec()) {

qDebug() << "Error retrieving user games:" << query.lastError().text();

return games;}

while (query.next()) {

int gameId = query.value("game\_id").toInt();

QString result = query.value("result").toString();

games.append(qMakePair(gameId, result));}

qDebug() << "Retrieved games:" << games; // Add this line to debug the retrieved games

return games;

}

# 6. Security Considerations

## 6.1 Authentication

* Implement a secure login and registration system where players can create accounts, log in, and manage their profiles.

## 6.2 Data Protection

* Use password hashing for secure storage of credentials and session management for keeping users logged in.
* The system shall use secure communication protocols to protect user data during transmission.

# 7. Performance Considerations

## 7.1 Overview of Performance Metrics

We analyze the performance of our Qt Tic-Tac-Toe game, focusing on CPU usage, memory usage, and move calculation times at various search depths. This data helps us understand the computational demands of our AI algorithms and optimize them for better performance and efficiency.7.2 Metrics

Identify key performance metrics and how they are measured.

## Performance AnalysisA graph with a line Description automatically generated

As the search depth increases, the CPU usage tends to vary, reflecting the increasing computational complexity of the AI's decision-making process.

A graph with a line

Description automatically generated

Memory usage shows a trend with increasing depth, indicating the growing data requirements for deeper AI computations.

A graph with a line

Description automatically generated

Move calculation time generally increases with depth, highlighting the additional time needed for the AI to evaluate more possibilities.

A graph with orange dots

Description automatically generated

A correlation between CPU usage and move calculation time reveals how processing power influences the speed of AI decisions

A graph with numbers and dots

Description automatically generated.

Memory usage against move calculation time provides insights into how memory allocation impacts the speed of AI computations.

# 8. Testing Strategies

* All tests were executed using Google Test framework. The tests passed successfully, indicating that the functionalities of Board, MiniMaxAgent classes, as well as the utility functions, work as expected. The following is a summary of the test results:

| **Test Case** | **Result** |
| --- | --- |
| boardAccessValid | Pass |
| resetBoard | Pass |
| VerticalWins | Pass |
| HorizontalWins | Pass |
| CheckTie | Pass |
| DiagonalWins | Pass |
| PlayerStartAIWin | Pass |
| AIStartAIWin | Pass |
| AIStartAIBlock | Pass |
| PlayerStartAIBlock | Pass |