Design Document

Purpose

The purpose of this document is to provide a description of the implementation plan for a game of three dimensional tic-tac-toe with specific considerations for the C++ programming language, as it will be the programming language in which the game will be implemented.

User Experience

Upon initial execution of the game, the player should be presented with a menu. The player should be presented with the opportunity to play, review the rules of the game, or quit.

Should the player choose to continue and play the game itself, they will be presented with three potential game modes: human vs human, human vs computer, and computer vs computer. If the player chooses to play against an AI opponent, the player will be asked to choose which colour to play as (and incidentally if they would like to go first or second).

The board will be displayed as a simple ASCII printout to standard output, as follows:

```
     |         |         |
 ----|---------|---------|----
|    |         |         |    |
|   (F)   |   (G)   |   (H)   |
|         |         |         |
|         |         |         |
|    |   (D)   |   (E)   |    |
|    |         |         |    |
|    |         |         |    |
|   (A)       (B)       (C)   |
|                             |
 -----------------------------
        R: 0      W: 0
```

where the current scores for both players are displayed below the board. Pieces will be represented as a single capital R (for red) or W (for white).

When it is the players turn to move, they will be prompted for an alphabetical input corresponding to the desired peg on which to place their piece. The game will continue until the board is completely filled with pegs, at which point the player with the highest score is declared the winner. Should the player wish to play again, they will be presented the opportunity to do so.

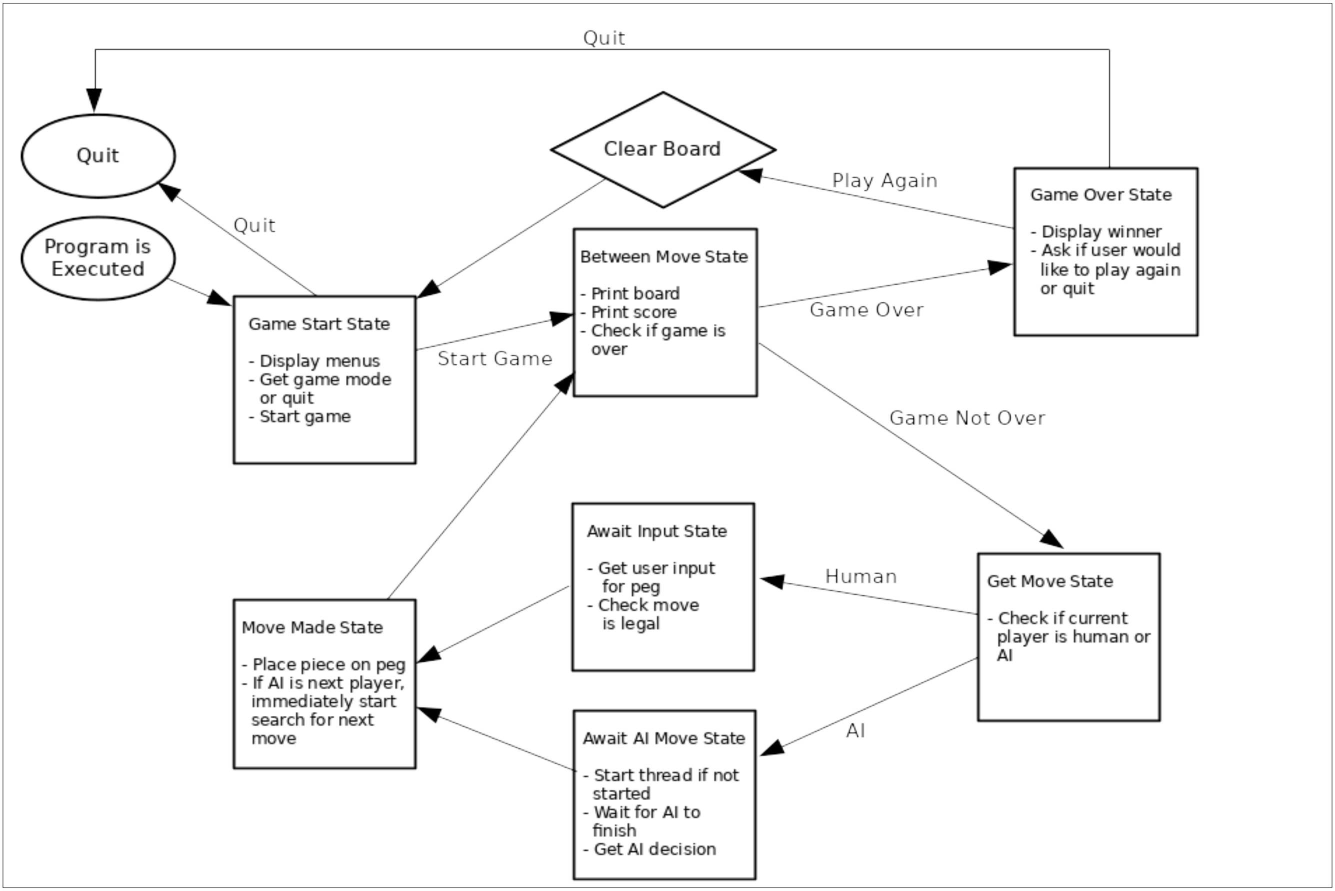
The Game

Specifics such as the player classes and their relevant get move methods are discussed in their relevant section of the design document.

The game is the main thread of execution. It will be responsible for displaying the previously discussed menus, the board and the score, and essentially driving the game forward. When it is a given players turn to move, the game will call the get move method of the current player.

When the move is made, if the next player to make a move is an AI player the game must immediately start the AI players get move method on a new thread. The game will then display the updated board, calculate the new scores, and then finally wait for the AI to finish it’s search. This will shorten the time that a player spends waiting for the AI to make a move.

This will continue until the game is over, at which point the winner will be declared. The user will then be asked if they would like to play again, and if not the game will quit.

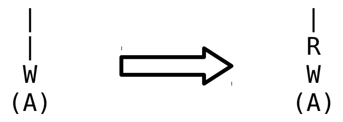


High Level State Diagram of the Game

The Board

The board and all of it’s components will be wrapped in a class. This board object will keep track of all previous moves and add pieces as requested through a method.

The board will have eight pegs, arranged as above. The pegs should behave similar to a stack in the sense that a piece will be placed on the bottom most unused position on the peg. For example, playing a red piece on a peg which already has a white piece on it will behave as follows:



An enumerative type will be created to represent the possible states a peg position may have: Red, White, and None, for red pieces, white pieces, and no piece, respectively. A peg can contain at most three pieces, and players should not be able to place a piece on a full peg. The board will then be represented as eight 1x3 arrays of the aforementioned enumerative type, where peg A is array 0, peg B is array 1, …, H is array 7.

Methods which must be implemented by the Board class:

- Add Piece:
  - `void addPiece(int, Colour);`
  - Add a piece of a given colour to a given peg being careful to place it on the lowest available position, or at the lowest unoccupied index of the corresponding piece array..
- Get Pieces
  - `int getNumPieces(int);`
  - Get the number of pieces on a given peg. A simple iteration over the corresponding 1x3 array, counting the elements which are not “None”.
- Get Move
  - `int getMoveNum();`
  - Get the current move number (how many moves have been played). A private variable will be used to keep track of the move number as the game progresses, this function will simply need to return it.
- Valid Move
  - `bool validMove(int);`
  - Check if the given possible move is a legal move with respect to the current game state. Should simply just need to check that the given peg is, in fact, a real peg, and that there are less than three pieces on that peg.
- Print Board
  - `void printBoard();`
  - Output the ASCII representation of the board as above.
- Evaluate Board
  - `int evalBoard(Colour);`
  - Calculate the score of the player with the given colours on the current board. This will have to be a brute force approach checking all 34 possible scoring positions.

The Players

Players, in general, should extend a generic “Player” class to aid in the programming of the game loop. This will allow the derived, polymorphic, player classes to be set to the current player as needed with great ease.

Methods which will be implemented by the Player class:

- Get Colour
  - `Colour getColour();`
  - Return the player’s piece colour. The player’s colour will be initialized by the constructor.
- Get Score
  - `int getScore();`
  - Get the player’s score. This will be called by the game before every move to retrieve the current score to display.
- Set Score
  - `void setScore(int);`
  - Set the player’s score. This will be called by the game after every move the player makes.
- Get Move
  - `virtual int getMove(Board) = 0;`
  - Get the desired move from the player.

The get move method will be implemented as a pure virtual method, in that both the derived human and AI classes must provide their own implementation of it.

The human class will prompt the user for input, where the AI class will perform a search on the game tree in order to determine the best move possible. To do this, the AI class needs to be passed a copy of the board.

The specific search algorithm to be implemented is discussed in the next section.

The AI

Tic-Tac-Toe is a perfect fit for the MiniMax algorithm. A good move for one player is a bad move for the opponent. Thus to accomplish the search for the best move, the AI player will use a simplification of the MiniMax algorithm called “NegaMax”, which exploits the fact that $\max(a, b) = -\min(-a, -b)$.

The depth of the search in the game tree will be determined through testing after the game has been written to ensure a balance between the amount of time a player will wait for an AI to decide on a move, and the difficulty of the game.

The pseudocode for the NegaMax algorithm is as follows:

```
function negamax(board, depth, alpha, beta, colour)
    if depth = 0 or no moves left
        return colour*board
    bestValue := −1000
    foreach possible move
        value := −negamax(child, depth−1, −beta, −alpha, -colour)
        bestValue := max(bestValue, value)
        alpha := max(alpha, value)
        if alpha >= beta
            break
    return bestValue
```

The AI player should determine it’s move as soon as possible to enhance the player’s experience. Since the AI player’s move may depend on the move it’s opponent makes, this will happen immediately following it’s opponent’s move (before the score is calculated, the board is printed, etc.), and must be initiated by the game.

This should be performed asynchronously – on a new thread of execution – from the main game playing thread. This will be implemented through the use of POSIX threads from the C++ standard library, and as such the game will only run on a POSIX compliant operating system (i.e. not Windows).

The heuristic score of the board will simply be the number of points a given player has in a given board state. This will cause the AI to play aggressively and focus on scoring points.