3D TIC TAC TOE (4*4*4)

Group 06

Team Members:

Saichandan Reddy Kancharla

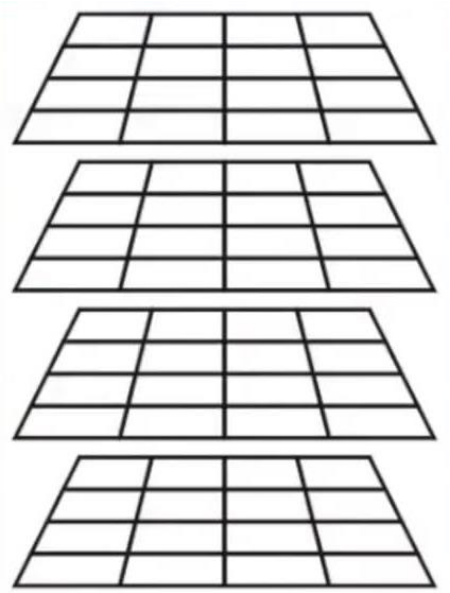
Jaya Devagiri

Sheshi Rekha Guntuka

Rithika Bejjanki

Introduction:

- ➔ 3D tic tac toe of the board of size 4*4*4 is similar to traditional tic tac toe.
- ➔ Win case: Four pieces in a row to win across all levels.
- ➔ The algorithm used in our project is Minimax and alpha-beta pruning.
- ➔ Backend Logic and UI programming are done in Java Native and GUI.



3D Tic Tac Toe View

Project description:

This program creates a three-dimensional version of Tic-Tac-Toe using the minimax approach and alpha-beta pruning. Depending on who wins, scores are updated. Due to the unique characteristics of 3D Tic-Tac-Toe, a tie cannot be achieved. By default, users can choose between "X" and "O" (human by default), and the difficulty level is medium. It has an easy-to-use GUI. Users could play a completely redesigned version of the old game using the system. In addition to maintaining the original game model, complexity, interactivity, and competitiveness were added. One player can participate. A four-by-four-by-four cube is used to play three-dimensional tic-tac-toe.

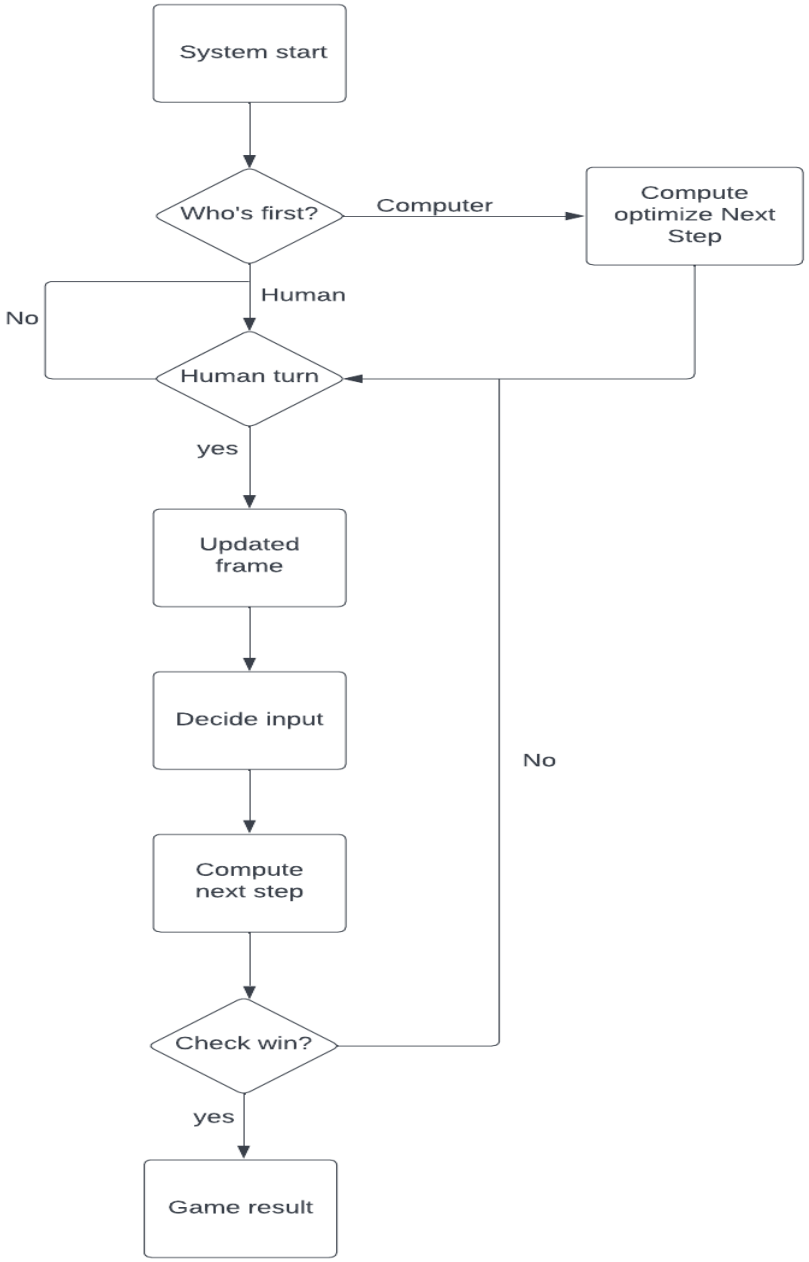
Algorithm:

The Minimax algorithm relies on the back-and-forth between the two players to choose the move with the highest score. Each general movement is then scored according to the lowest score for each available activity. Every time a player takes a turn, it determines the scores for the opposite player's actions, and so on, down the move tree to the end. To create alpha-beta pruning, MINIMAX has been altered. This method is used to optimize the MINIMAX algorithm. According to the number of games, the MINIMAX search algorithm should examine trees exponentially with depth. Although we can reduce the exponent to half, we cannot eliminate it.

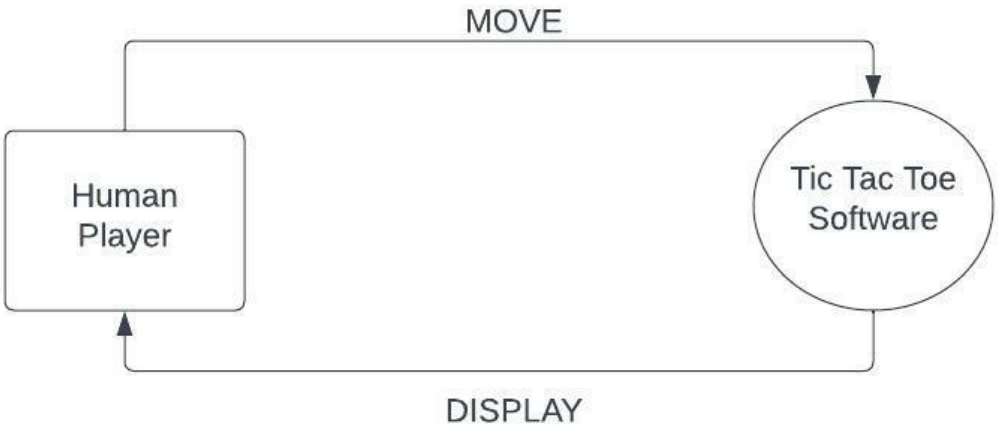
We can compute the correct minimax decision by pruning the game tree without checking each node. The Alpha-Beta Pruning process involves two threshold parameters, Alpha and Beta, for future growth. Alpha-beta pruning is a modified version of the minimax algorithm. It has two parameters and is defined as Alpha: At any point along the Maximizer's path, Alpha is the best (highest value) option. Beta is the most advantageous (lowest value) option at any point along the Minimizer's path. The beta value is initially set to $+\infty$.

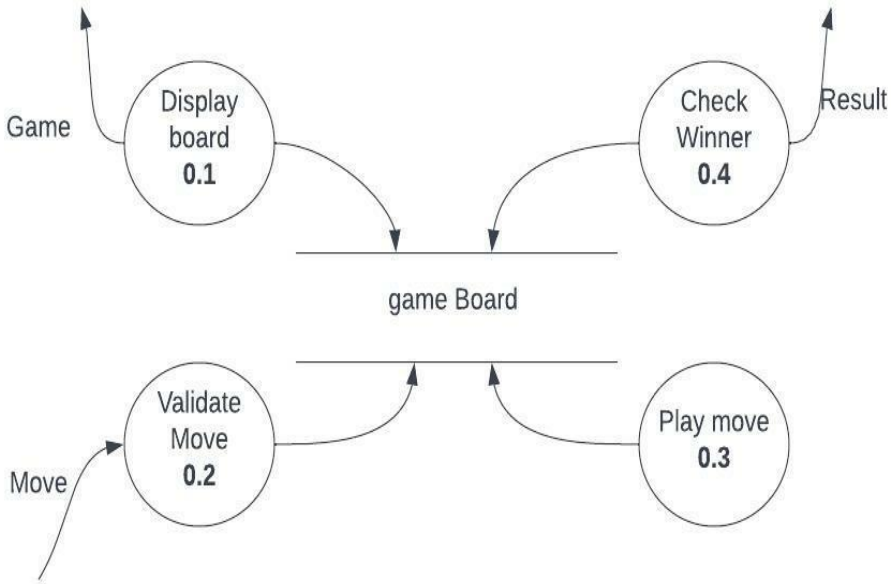
Alpha-beta pruning is the same as when the algorithm is applied to a standard minimax algorithm. In addition, it eliminates all the nodes that merely slow down the algorithm without affecting its outcome. By pruning these nodes, the algorithm becomes faster. The value of Alpha, initially set to -650001 (one less than the possible move, one in which the opponent wins), keeps track of the most effective action the opponent has allowed us to take. Since the min move is aware that the player can perform better than alpha, it can terminate immediately if it encounters a move worse than alpha. Beta, initially set to 600001, keeps track of the worst activity the opponent can make the player do (one more than the value of a win).

System Flow Diagram:



Data Flow Diagram:





Input Design:

A system's input design requires specific consideration as part of its overall design. Input data design aims to make data entry easy and error-free. Input forms were built using JAVA controls.

Input design involves translating user input into a computer-based format. Clicking any point on the board will send the corresponding axis coordinates to the backend, and the X or O symbol will appear on the Board screen.

The new Game Button resets the game, and Easy, medium, and hard radio buttons change difficulty levels. Users or CPUs can make the first move on the board by pressing the CPU or Human First radio buttons.

Program steps:

Board_Panelting is the method that builds the GUI.

Listener_For_BoardPanel is an ActionListener that determines the initial player depending on player input. By clicking it, the board is cleared, the starting player is chosen, the title text is changed, and the difficulty level is checked. For a more competitive game, the computer will play in a random position if it is chosen to go first, and the difficulty level is not high.

If the radio button for the CPU to go first is selected, the **NewButtonListener** ActionListener will go first and empty the board, set the text, and begin a new game.

Listener_For_Movements is an ActionListener that modifies human and digital piece variables in response to user input. After that, it resets the game and clears the board.

With the help of the ActionListener **Listener_for_Levels**, users can control how aggressive or intelligent the machine will be by manipulating the difficulty value. To begin a new game, the class modifies the global difficulty setting.

The listener for each button on the GUI is called **actionPerformed**. If the user clicks on a space, the program reads that information and writes it to the internal and GUI boards before simply determining whether the move was successful. In that case, declare victory and show the winning move or message. Call **random_movement_by_cpu** if it isn't.

Score_BoardPanel() is used to update the score panel with the correct score when a win has occurred

When the difficulty setting is set to easy or medium, and the computer is chosen to go first, the method **movement_by_cpu()** is used. This is done because, using the minimax strategy, allowing the machine to move first makes it nearly impossible for a human to prevail. Placing the first move randomly makes the game more exciting and competitive. The difficulty setting is easy or medium, suggesting that the player may want to win. When the difficulty setting is challenging, this function is not invoked, allowing for highly aggressive play, as the difficulty setting would suggest.

The primary method utilized in this game's artificial intelligence implementation is **random_movement_by_cpu ().** It goes through every move that is currently accessible on the game board, uses **lookAhead()** to construct branches off of those moves, determines what the

player might do in reaction to each conceivable move, and then chooses the move that is most likely to succeed in achieving the desired result.

In response to a potential move made by the computer in **random_movement_by_cpu**, **lookAhead()** creates every conceivable move in the open spaces depending on the current board (). The **heuristic()** function is used to determine the heuristic value returned by this method. Given that the search tree might get somewhat vast when playing on hard difficulty, this function uses the alpha-beta pruning strategy.

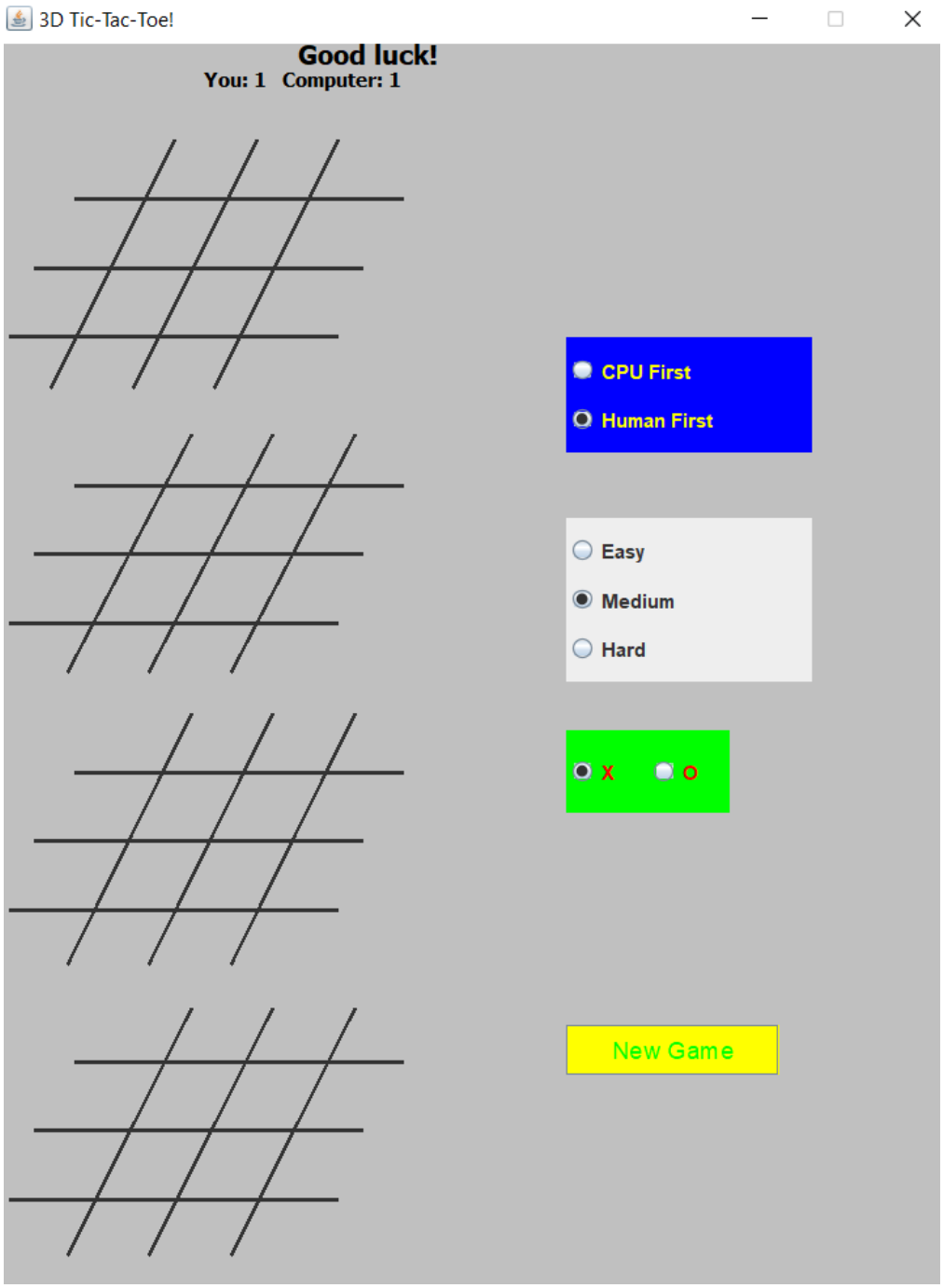
heuristic() merely subtracts both the computer's **Positions_for_BoardPanel** method and the human's **Positions_for_BoardPanel** method from the current board, with a greater value being more promising for the computer.

Win_Meth() accepts a character to be checked for a win and a move to be checked for a win. It employs a 1-dimensional array to represent all the spaces on the game board and a 2-dimensional array to hold every winning combination that could be made.

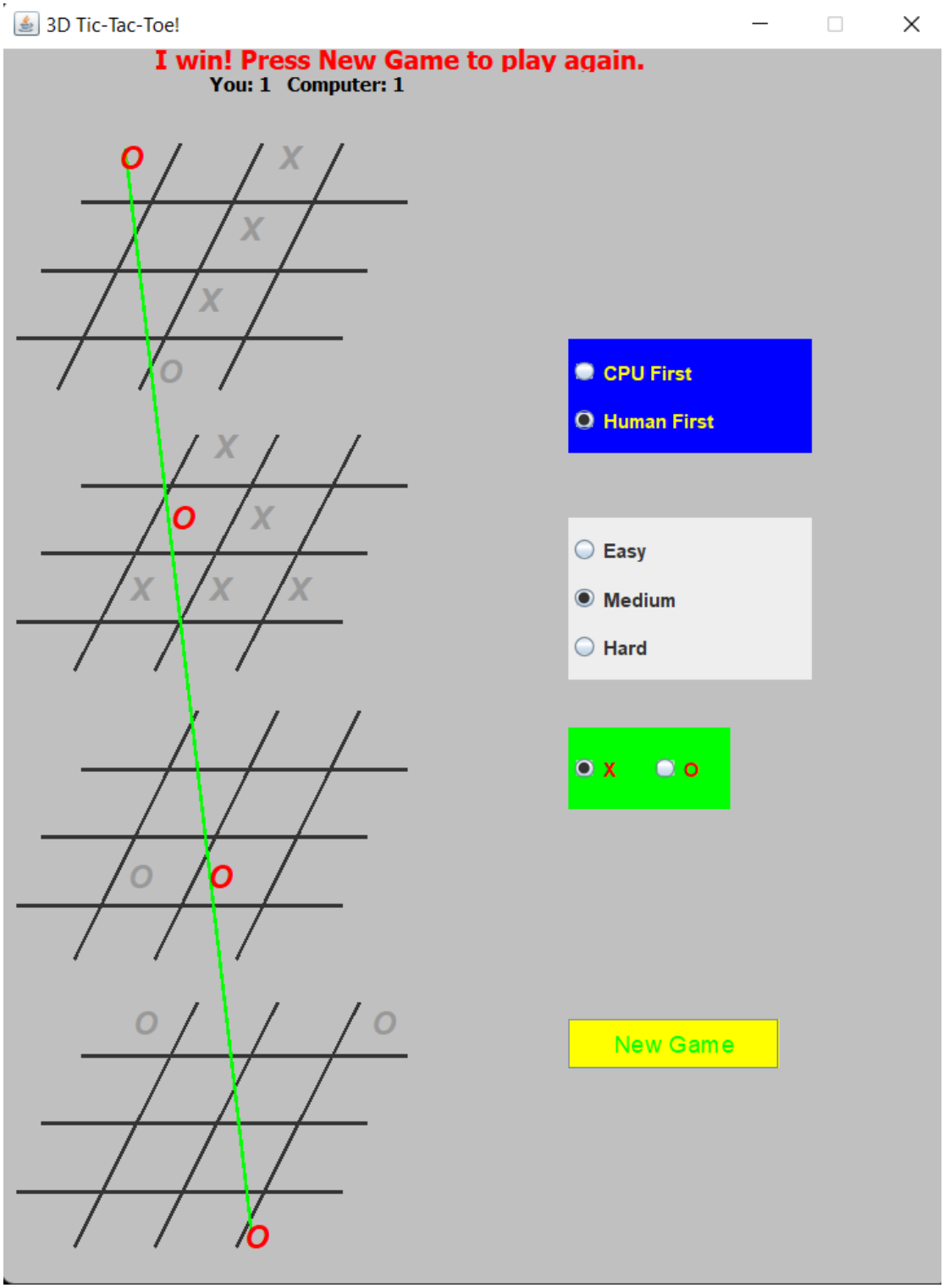
The **Positions_for_BoardPanel** function is quite similar to the **Win_Meth** () method. Still, instead of returning a Boolean indicating whether or not the input move is a win, this method returns an int indicating the number of wins that are now feasible for input char c ('X' or 'O').

Output Design:

After the Successful Run of our code, our application opens:



Computer Won after the Game:



Human Won after the Game: