Mancala Game

1. Team Members.

Eslam Khaled Asfour	1600254
Ahmed Khaled Sayed	1600073
Omar Hany Mohamed	1600921
Rasha Mahmoud Ahmed	1600558
Ahmed Gamal Abdelhamid	1600050

2. Game Description.

Mancala is a family of two-player turn-based strategy board games played with small stones and rows of holes or pits, a board or other playing surface. The objective is usually to capture more stones than the opponent's pieces.

3. Implemented Features.

- GUI.
- Implemented along with an AI-player using the minimax algorithm with alpha-beta pruning.
- Support 2 modes : single player , Multi Players
- Player 1 always starts the game.
- Support various difficulty levels.
- Every player is adjusted to respond with a move in 20 seconds.
- Support game saving and loading.
- Support Background music in the beginning of the game.

4. Detailed Description of code.

Our code is implemented as OOP so it consists of Classes

1) Class Board

a) Attributes

- **Playing**: A Boolean variable to indicate the game status which is true or false and decide whether to loop on the board and ask for input or not.
- **Player**: An int variable that indicates which player turn is it. Value of 1 indicates player 1 turn and 2 indicates player 2 turn.
- **Stealing**: A Boolean variable that selects the mode of the game which can be with stealing or not.
- **Wrong_turn**: An int variable that indicates that player 1 played a pile index from the opposite side of the board – player 2 piles – and vice versa. Value of 1 for indicates player 1 wrong input, and 2 indicates player 2 wrong input, and value of 0 indicates right input.
- **Wrong_turn**: An int variable that indicates that player 1 played a pile index from the opposite side of the board – player 2 piles – and vice versa. Value of 1 for indicates player 1 wrong input, and 2 indicates player 2 wrong input, and value of 0 indicates right input.
- **SaveLoad**: A character variable that indicates whether to save the current game – if the user was already in a game - or load a previously saved game in the main screen of the game. It can take 's' to indicate saving a game and 'l' to indicate loading a game.

- **Winning_player**: An int variables that decides whether to stay playing or declare a winner. This variable can take 0 as an indicator for continue playing, 1 for player 1 winning, 2 for player 2 winning, or 3 for draw.
- **Player1_store**: An int variable represents player 1 store index which is 6 for a more readable code.
- **Player2_store**: An int variable represents player 2 store index which is 13 for a more readable code.
- **Piles**: A list that contains board piles representation as indices. From index 0 to 5 are player 1 piles and 6 is player 1 store, and from 7 to 12 are player 2 piles and 13 is player 2 store.
- **pilesDict**: A dictionary that maps each index "regardless of the stores indices" to the opposite index on the board. This dictionary is useful in the stealing function.

b) Functions

- **toggleGameStatus(self, player)**: this function toggles the playing attribute and chooses the starting player.
- **togglePlayer(self)**: this function toggles the player turn.
- **prepMove(self)**: this function detects whether the index that the player clicked is right or wrong, and it does not continue if it is wrong , it also defines the depth of recursion for playMove() function, and after the player move is done it checks whether the game is finished or not.

- **playMove(self, depth)**: this is a recursive function takes a depth "number of the stones of the clicked pile" and loops recursively till the depth is 0. The last clicked_index would be the ending index,
  If the depth is equal to 1, The function checks whether the player last index landed on his/her store for an extra turn, and also checks for whether the game should end or not. If it landed in the other player store it is moved to the next pile. Also, if the clicked_index exceeded 13 it will be set to zero,
  If we are in stealing mode, at depth 1 the function will check if the last index matched an empty pile of the current player piles, and the opposite pile in the other player side of the board is not empty, if the previous statement is true, the opponent stones are stolen and added to the player store then the players turn is toggled,
  At the end of this function, we call it again with a depth of depth – 1. Also, this function adds 1 for every pile it passes through "excluding the opponent store".
- **toggleIfDone(self)**: this function checks if the current player has any possible moves, if not it toggles the player. This function is called inside playMove() function.
- **printPiles(self)**: this function was made for testing purposes. It prints the current board in console after each turn. This function gets called in the main loop in main script.

- checkToEnd(self): this function checks each turn if the sum of both players stores are equal to 48 or not. If the sum is 48, we check for the winning player and set winning_player with the winning_player or set it with 3 in case of draw. If the sum is not 48 then the game did not end and set the winning_player to zero.
- **extraTurn(self)**: this function is called after the current player had an extra turn, if the game did not end, if the player has any possible moves, he/she gets the extra turn otherwise the player is toggled.
- **saveGame(self)**: this function is used to save the current board status "the piles list", and the current player turn, and the game mode, which is stealing or not, and the chosen game difficulty.
- **loadGame(self)**: this function is used to load a saved game and continue playing

2) Class Player inside Minimax Script.

a. Functions

- **Get children**:
  - **Inputs**: list of Board Piles, Boolean that refers to player 1 or 2, Boolean that refers to using stealing or not.
  - **Output**: List of children, which is the output board for each available play move along with the index of this move and flag that refers to having extra play or not.
  - **Description**: The function checks first about stealing mode to define how the output of the board will look like then it checks about the player to define the range if available playing moves

then it loops among these indexes and spread the pile stones among the board if it’s a non-zero pile, then it checks if the last stone is dropped in the store then this player has extra moves so it puts the flag to True, after this it appends the board, index and the flag to the children list and after finishing looping it returns this list of children.

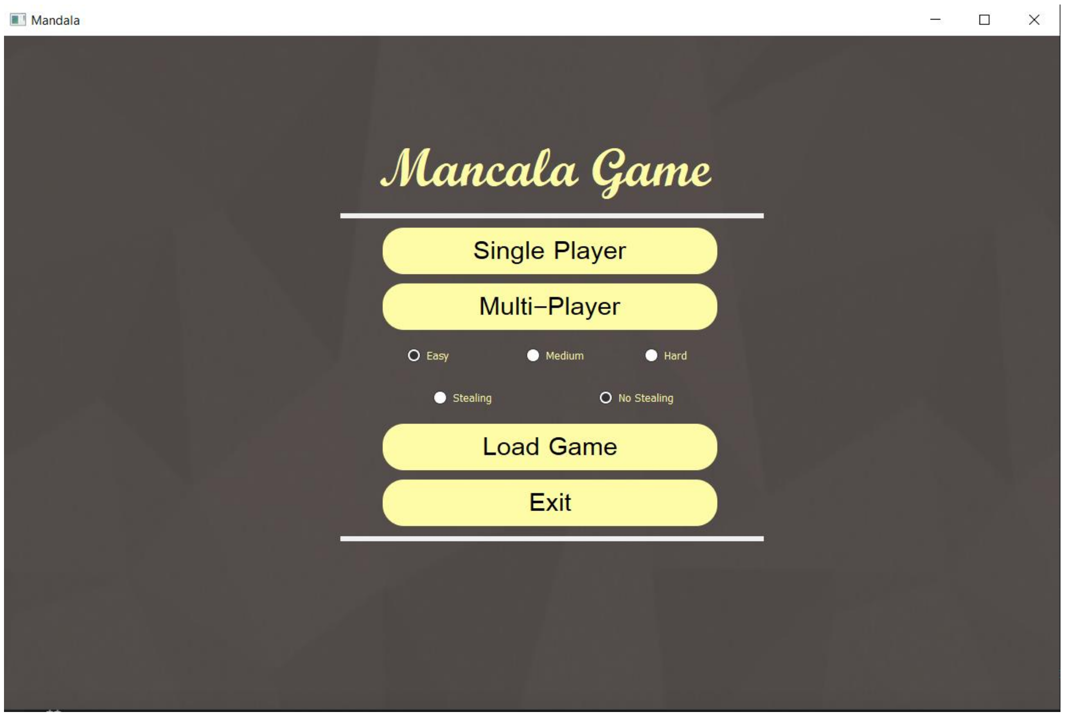
- **Heuristic**:
    - **Inputs**: list of Board Piles, Boolean that refers to player 1 or 2.
    - **Output**: Heuristic value
    - **Description**: the function checks which player is calling the function and then it returns his store value minus the opponent store value.
- **MiniMax:**
    - **Inputs**: list of Board Piles, Boolean that refers to player 1 or 2, Boolean that refers to using stealing or not, alpha and beta initial values.
    - **Output**: index of the best move to play
    - **Inside Functions**:
        - Game is over function: it takes the board piles as input and checks if all piles are already in stores, then it returns True as the game is over unless that it returns False.
        - Adjust Function: Function to make sure that never choose a zero pile as play move.
        - MiniMax Helper function: Function that is used for the recursion, it starts by checking if the game is over in the

sent board, or if the depth reached zero it returns the heuristic value and index of -1 (that will be changed in the future), if the game didn't end it sets the best score value with initial value, then it calls the get children function to get children of the current board then it loops among these children, first it checks that the played index isn't zero pile play, then it checks if this play makes extra play, in this case it calls the minimax helper function again for the same player with depth reduced by one , if it doesn't have extra play it will call the minimax function for the other player with depth reduced by one and this will go on until it reaches depth zero or game is over. After the function return it checks if the return heuristic value is better than best value it replaces it and replace the best index and if alpha>beta it breaks the loop as it does not matter for the other values. At last, it returns the best score and best value.

- **Description**: it calls the minimax helper function with the sent board, chosen player and maximizing mode to get best move and then it returns the index of this move.

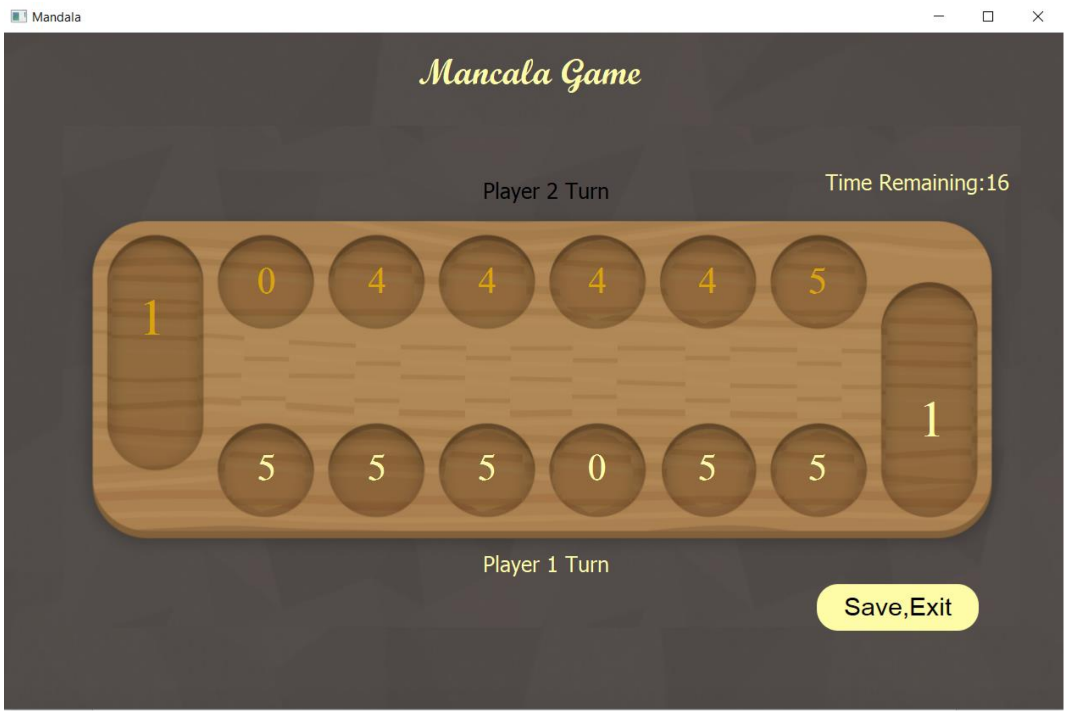
5. User guide with snapshots.

- Home page:



- If user chooses **(Single Player-Multi-Player)** he must first select the Game Mode **(Stealing/NoStealing)** and Game Difficulty **(Easy/Medium/Hard).**

  - 1st case: User Choose **Single Player** after Selecting **Game-Mode** and **Difficulty.**

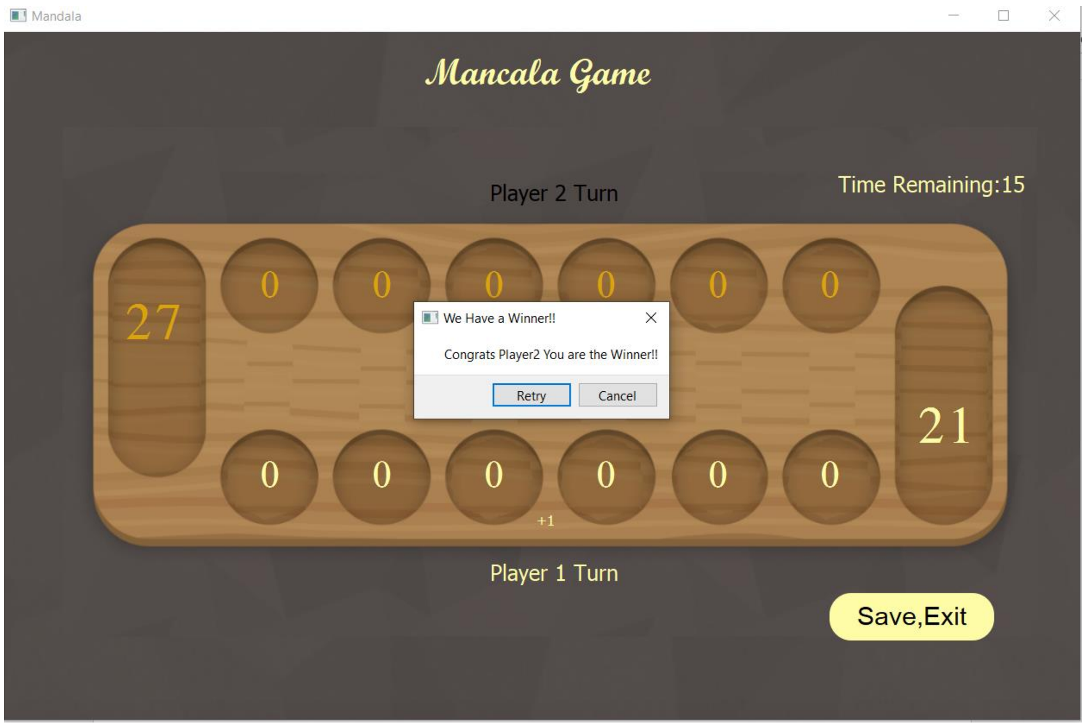


- Player 1 starts with **20sec** to choose his play, in case he didn't play the game will automatically play the best play for him.

- Then Player 2 **(AI)** will play and so on.

- Any Time while playing the player can **Save,Exit** the game and can load if anytime to continue.

- Until the game ends with a message of the winner.

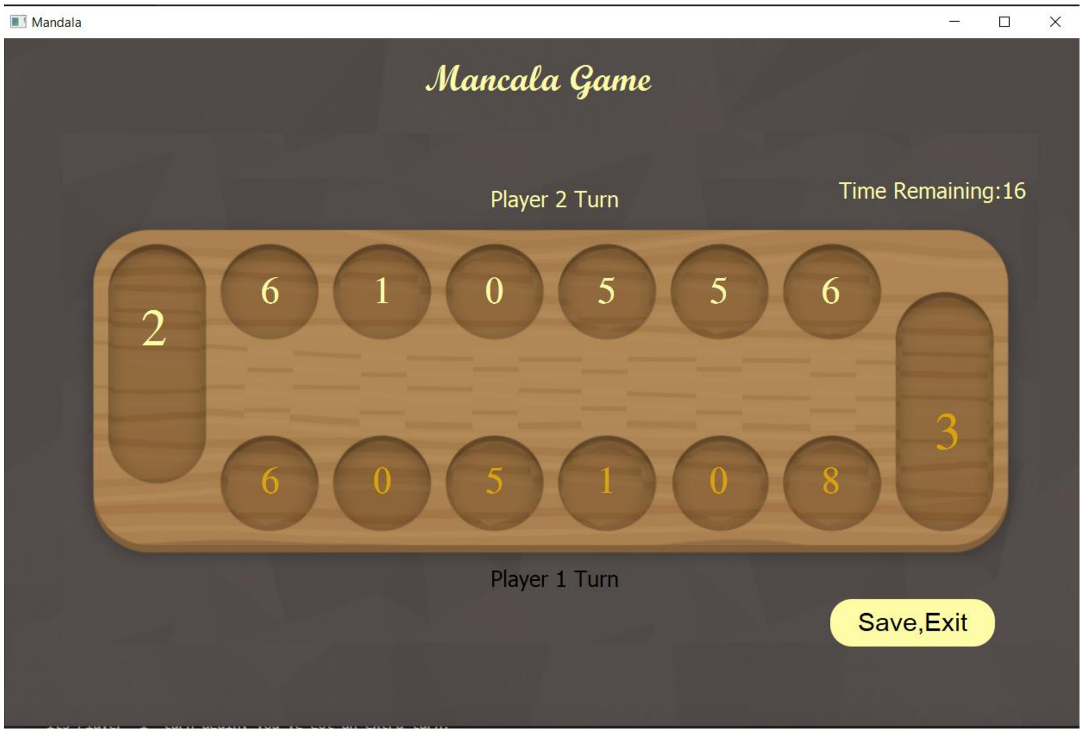


We Have a Winner!!

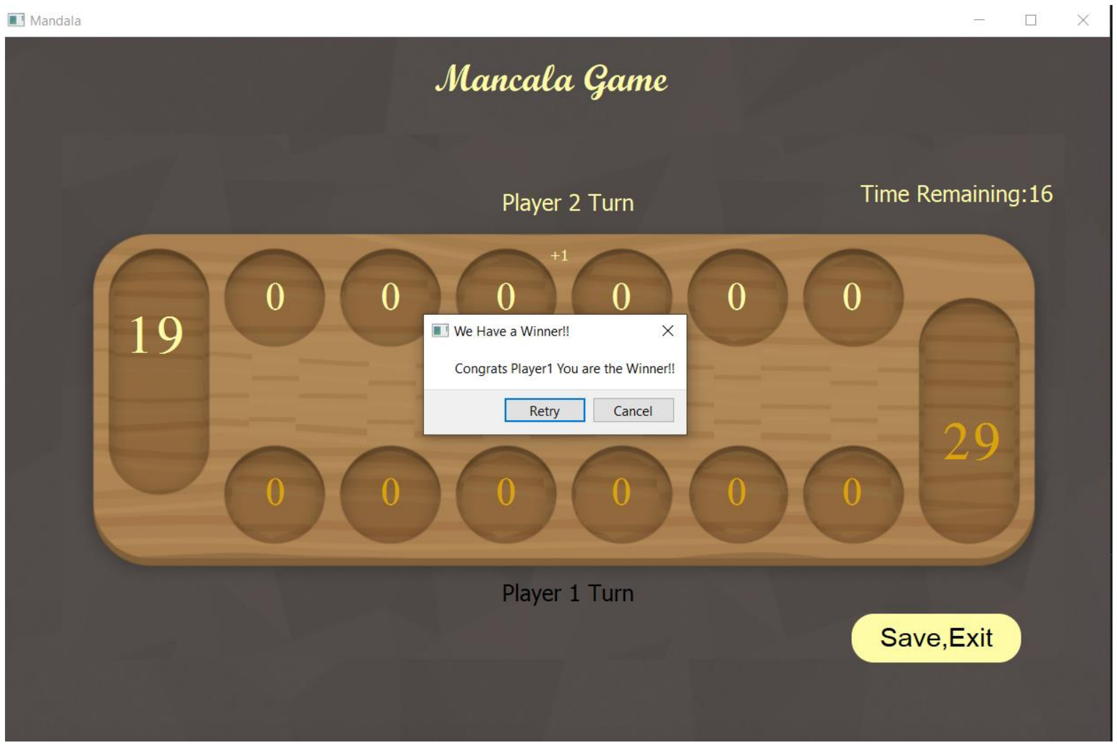
Congrats Player2 You are the Winner!!

- The player can select **Retry** to try again with the same settings or **Cancel.**

- 2nd case: **Multiplayer** mode After Selecting **Game-Mode** and **Difficulty.**



- Each Player has **20sec** to play or the **(AI)** will play the best move possible.
- Anytime the players can select **Save,Exit.**



We Have a Winner!!

Congrats Player1 You are the Winner!!

- **3rd Case**: Load Game
  - If a game was already save the game will load with the same scores and player turn and mode.

- **4th Case**: Exit Game

6. Team Participation.

A. Class Board: Rasha Mahmoud , Ahmed khaled.

B. Class Player: Omar Hany , Ahmed Gamal.

C. GUI: Eslam Khaled.

D. Report: All.