Tic Tac Toe Game on Ethereum

Introduction

We are going to implement the popular Tic Tac Toe game on the blockchain. For more information on this game, please visit the link: https://en.wikipedia.org/wiki/Tic-tac-toe

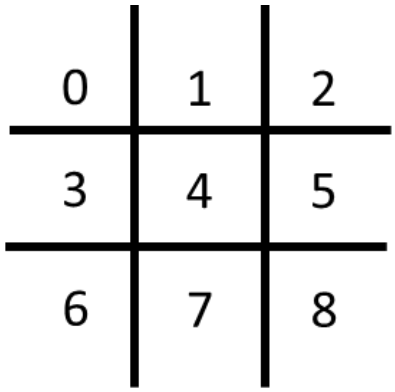
Housekeeping points

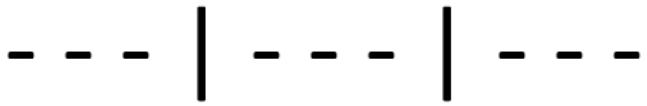
- This is a minimal example and may not follow some standard practices
- We'll leave out some obvious tasks like allowing multiple games between two players, ensuring different blocks between proof and choice storage, etc.

Program Structure and Organization

We will implement this game by writing a contract **TicTacToeGame.** This contract will store the individual moves on the game board played between two players.
Two players will play this game. **playerOne** and **playerTwo**. One of the players who starts the game, by deploying the contract, will be the **playerOne.** For playerTwo, the details (address) will be passed to the contract while starting the game.
Either of the players can start with the first move.

The implementation of the game board in this contract is done using a **one-dimensional** array of addresses. Each element of the array will store one of the three values. address(0), this will be the default value of all the elements. address(playerOne) or address(playerTwo), this depends on the move each player makes while playing the game.





The above left side figure shows the actual game board with index values.
The right side figure is the one-dimensional array representation of the game board in this project. The first 3 dashes represent the index (0,1,2). The 3 dashes in the middle represent indexes (3,4,5) and the last 3 dashes represent the index (6, 7, 8)

Following the flow of the game.

1. The game starts by providing the address of playerTwo. The person who deploys the contract is the playerOne in this game.
2. Either player can start the game by making the first move. A player mentios one of the indexes to mark the move. While doing so, the address of the player is stored at that index.
3. This contract provides a functionality to show the game board as well. While doing so, instead of showing the addresses, it shows **X** for playerOne and **O** for playerTwo. And **-** where players are yet to play a turn.
   Example. Initially the game board will look like **- - - | - - - | - - -**
   Now playerOne takes his turn and makes a move at index 8. After the move the game board will look like **- - - | - - - | - - X**
   Now playerTwo takes his turn and makes a move at index 0. After the move the game board will look like **O - - | - - - | - - X**
   Once all the 9 moves are made, we either will have a winner or a draw game. Following is an example of winning game by playerTwo
   O O O | X X O | X O X, winning Index (0,1,2)
   O X X | X O X | X O O, winning Index (0,4,8)
   X X O | X O X | O X O, winning Index (2,4,6)

Problem Statement

The program structure has already been defined and as a part of this project you are expected to implement a specific method.
Do read the comments in the code carefully. Especially on the methods that you have to implement.

1. Complete the method **startGame**(). This method accepts the address of the second player as a parameter and initializes the variable playerTwo of type address.

2. Complete the method **isWinner**(), which is a private method. This method accepts a parameter “address” of one of the two players. Here the main idea is to return true if we get the winner by computing the
    a. Equality check in one of rows [0,1,2], [3,4,5], [6,7,8]
    b. Equality check in one of the columns [0,3,6], [1,4,7], [2,5,8]
    c. Equality check in one of the diagonals [0,4,8], [6,4,2]

    While implementing the method **isWinner**() we have to do the following tasks…
    1. Validate, all the 8 combinations of “Equality Check” defined above, have the same address or not. If any of the combinations has the same address. This defines a winner of the game. And the function returns **True**.
    2. This function returns **False** when no winner is identified.

3. Complete the method **placeMove**() which will take an index as the parameter. Valid indexes are from 0 to 8. While implementing the method do the following checks
    a. Checks before a player is placing a move
        i. Is the game over? Check the flag **isGameOver** and **_winner**. When **isGameOver** is set to **True** or **_winner** stores the address of one of the players it indicates that the game is over. If **isGameOver** is False or **_winner** has the address 0, the game continues
        ii. Is the game a draw? Check the flag **isGameOver**. When **isGameOver** is set to **True**, it indicates that the game is a draw
        iii. The players (playerOne and playerTwo) must be the valid players. No other player should be able to play the game.
        iv. A player should only be able to play a move on one of the blank index only. For example, when a player plays a move on index 5, then all the previous moves of both the players should not have played on index 5.
        v. It should not allow a player to play two moves in sequence.
        vi. —----------
        vii. Is the game over or not? For this we can check off the isGameOver flag is set to true and the winner is not equal to the address(0)
        viii. Is the game a draw? For this, we can just check the isGameOver flag
        ix. The player which is exercising the move must be from one of the players(playerOne/ playerTwo)

x. If the player is exercising the move on the index 5, then gameBoard[5] should not be equal to the address(0), else we can just print the message that the slot is already taken.
xi. If the player which is exercising the move is not the lastPlayed player, otherwise we can just print the message that your attempt is already done. The next player can exercise the move.

b. When a player makes a move, we do the following activities…
   i. Store the address of the current player in the index of the gameBoard like gameBoard[index] = msg.sender
   ii. Mark the current player as the last player.
   iii. Increment the number of turns taken by 1
   iv. ——————————————————————————————————
   v. Now it's the time to exercise the move by the current player. So we will store the address of the current player in the index of the gameBoard like gameBoard[index] = msg.sender and mark the current player as the last player. Also, we can increase the number of turns taken by 1 unit.
c. Once a move is made, check whether we have a winner or not. If we have a winner update the flag isGameOver with True. Update the variable _winner with the address of the player who played this move.
   In case we do not have a winner, we check for the number of turns taken. If this counts to 9, it means we do not have a winner. The game is a draw.

   Now it's time to check if we have the winner after this move. If yes then we will update the isGameover flag and also the winner variable which will now pose the address of the current player(winner). Also, we will check if the number of turns taken is 9 and we don't have the winner, this means we have a draw.

Evaluation Rubric

Total Project Points: **240**

- Basic compilation without errors (**10%**) : **24 Points**
- Correctness:
  - Problem statement - 1 (**10%**) : **24 Points**
  - Problem statement - 2 (**30%**) : **72 Points**
  - Problem statement - 3.a (**30%**) : **72 Points**
  - Problem statement - 3.b (**10%**) : **24 Points**
  - Problem statement - 3.c (**10%**) : **24 Points**

Program Instructions

1. Go to https://remix.ethereum.org, create a new file in the contracts folder, name it TicTacToeGame.sol and load up the initial code given in the TicTacToeGame.sol file, given along with this doc.
2. Your goal is to modify this file to accomplish solving Problems 1,2 and 3.
3. Use the 0.8.16 solidity compiler version as already specified in the file.
4. Deploy the TicTacToeGame contract.
5. You can use the local 'Javascript VM' itself to try this out. There are multiple accounts already available so you can use them to test the game.
6. Please don't change existing functions unless needed to solve the problems.
7. You might need to concatenate strings to generate appropriate game comments. This is one way to do that in Solidity:
   string(abi.encodePacked(string1, string2, string3))

References

1. https://remix.ethereum.org
2. https://docs.soliditylang.org/en/v0.8.6/