Ludo like ucsc
Manuja yasas
1st September 2024

1. The Structures Used to Represent the Board and Pieces

In the updated C program, three main structures are used to represent the game components: `Pieces`, `Player`, and `Borad`. These structures are designed to encapsulate various attributes and states associated with the game pieces, players, and the board.

1. `Pieces` Structure:
  - `piecesCaptured`: Tracks whether a piece has been captured.
  - `piecesPosition`: Stores the current position of the piece on the board.
  - `startgame`: Indicates whether the piece has started moving in the game.
  - `piecesDirection`: Represents the direction in which the piece is moving.
  - Other attributes include `pastHomeHowTimes`, `pastpiecesDirection`, `piecesBlock`, `pieceInHomeCell`, `mystryCellTurns`, `movementModifier`, `turnSkkiped`, and `mysteryCell` to handle various piece states and behaviors.

2. `Player` Structure:
  - `pieces[4]`: An array of `Pieces` representing the four pieces each player controls.
  - `playercolor`: A character indicating the color assigned to the player.
  - `startGamePieces`, `piecesBlock`, `finishPieces`: Attributes that manage the player's overall game state.

3. `Borad` Structure:
  - `piecesName`, `piecesBlock`, `mysteryCell`: Attributes that represent specific states of the board, such as naming pieces, blocking pieces, or mystery cells.

2. Justification for the Used Structures

The choice of structures in the program is justified by the need for clear, organized, and modular management of game elements. Each structure is carefully designed to encapsulate related data, promoting code readability and maintainability. For example, the `Pieces` structure groups all attributes related to a single piece, allowing easy manipulation and tracking of its state. Similarly, the `Player` structure brings together all pieces and relevant player-specific data, facilitating straightforward player management. Finally, the `Borad` structure is used to handle board-specific states, which are crucial for gameplay mechanics.

3. Discussion of the Efficiency of Your Program with Justification

The program is designed with efficiency in mind, both in terms of memory usage and performance. The use of structures allows related data to be stored compactly, reducing the memory footprint. This design also enhances performance by minimizing the need for repetitive calculations and making the code more organized and faster to execute.

For instance, the program keeps track of each piece's position, direction, and state within the `Pieces` structure. This eliminates the need for complex lookups or recalculations, as all necessary data is readily accessible. Furthermore, by grouping player-specific data within the `Player` structure, the program can efficiently manage multiple players, each with their own set of pieces.

The scalability of the program is another strong point. By using these structures, the program can easily be extended to accommodate more players or additional rules, without requiring significant changes to the core logic. This modular approach ensures that the program remains manageable and adaptable.

4. Functions: Purpose, Inputs, and Outputs

roll()

Purpose: This function simulates rolling a six-sided dice.
Input: None.
Output: An integer between 1 and 6 representing the result of the dice roll.

coinTos()

Purpose: This function simulates a coin toss.
Input: None.
Output: An integer, either 0 or 1, representing the result of the coin toss.

playermove()

Purpose: This function handles the movement of a player's pieces based on the roll of the dice.
Input: Player *player, Borad *borad, int *playerNum, int piecesNum[3], int rollNum, int *piecesToHome, int howManyPiecesStart.
Output: None, but it updates the positions and states of the pieces based on the game logic.

playerstart()

Purpose: Determines which player starts the game by rolling the dice for each player and resolving ties.
Input: int *startPlayer.
Output: Returns the index of the starting player.

playerInstall()

Purpose: Initializes the player structures with default values before the game starts.
Input: Player *player.
Output: None, but it sets up each player with initial states, like setting their pieces to the base position.

Ludo like ucsc
Manuja yasas
1st September 2024

boradconst()

Purpose: Initializes the board structure with default values, setting up the initial state of the board.
Input: Player *player, Borad *borad.
Output: None, but it prepares the board for the start of the game.

countEnemyPiecesAtPosition()

Purpose: Counts the number of enemy pieces at a given board position and identifies which player they belong to.
Input: int position, int currentPlayer, int *enemyIndex, int (*enemyPieces)[3].
Output: Returns the number of enemy pieces found at the position.

countAllyPiecesAtPosition()

Purpose: Counts the number of ally pieces (belonging to the current player) at a given board position.
Input: int position, int currentPlayer, int (*allayPieces)[3], int *piecesNum.
Output: Returns the number of ally pieces found at the position.

blockDirectionChecker()

Purpose: Determines the direction of a block (a group of pieces moving together) and updates their movement directions.
Input: Player *player, Borad *borad, int cureentPiecesDirection, int newPiecesDirection, int newSecondPiecesDirection, int piecesNum, int *allayPieces.
Output: None, but it updates the direction of the moving pieces.

piecesCaptured()

Purpose: Handles the logic when a player's pieces are captured, sending them back to the base.
Input: int enemyPlayer, int firstpieces, int secondpieces, int thirdpieces, Player *player, Borad *borad.
Output: None, but it resets the captured pieces' states and positions.

blockBreaker()

Purpose: Resets the direction of pieces in a block after a block is broken.
Input: Player *player, Borad *borad, int allayplayer, int pieces1, int pieces2, int pieces3.
Output: None, but it updates the direction and block status of the involved pieces.

mystryCell()

Purpose: Handles the behavior of pieces that land on a mystery cell, which can have various effects.
Input: Player *player, Borad *borad, int firstPiece, int secondPiece, int thirdPiece, int cureentplayer.
Output: None, but it applies the mystery cell effects to the pieces.

Ludo like ucsc
Manuja yasas
1st September 2024

greenplayer()

Purpose: Manages the moves and decisions for the green player based on the roll of the dice.
Input: Player *player, Borad *borad, int playerNum, int rollNum, int *piecesToHome, int (*piecesNum[3]).
Output: None, but it determines which pieces the green player should move and how.

This will go on greenplayer redplayer yellowplayer and blueplayer

On their behaviors oder will be change.