Language: Racket programming language

The battleship game includes a user interface to select the number of players and ships, place ships on a grid, and play the game by taking turns to guess the location of opponent ships. The game is designed to support both single-player (Player vs. AI) and two-player modes. The game logic involves placing ships, making guesses, tracking hits and misses, and determining the game-over state. This documentation will explain the structure and logic of the code to ensure easy future development.

**Code Structure Overview**

1. Initialization: Constants and variables are defined to set up the game states, board sizes, and control flow.

2. Game States: Defined using constants (home, game-mode-selection, etc.). These states represent different stages of the game.

3. Game Boards: The function createBoard initializes the game boards, setting up a 2D grid represented by nested vectors.

4. Ship Placement: Functions check for valid ship placement and handle adding or removing ships from the board.

5. Gameplay: The main gameplay involves a player making guesses to hit the opponent's ships. The game loops through states and updates the board based on player actions.

6. Drawing Functions: UI drawing functions handle rendering different game states to the screen, such as grids, text, and buttons.

7. Game Loop: The function game-loop coordinates the game's state transitions bycalling update and draw functions in a loop.

**Initialization**

- The game states (home, game-mode-selection, etc.) are defined as constants, using numbers for easier state management.
- Variables such as boardSize, cellSize, x-offset, y-offset, button-width, and button-height are declared to control the layout of the game UI.
- currentState is initialized to home, indicating the starting point of the game.
- playerTurn, game-mode, and ship-related variables (num-ships, ships-placed, ship-sizes, etc.) are declared to manage game flow.

**Game States**

- The game transitions between different states using the currentState variable. Each state corresponds to a part of the game (e.g., main menu, ship placement, in-play).
- The states are used to control what part of the game logic is executed in the update and draw functions.

**Game Boards**

- createBoard: This function creates a 2D vector (board) to represent the game grid. The size of the board is defined by boardSize.
- initialBoard, opponentBoard, player1-board, and player2-board are created using createBoard, setting up the grids for both players.

**Ship Placement**

- can-place-ship?: Checks if a ship can be placed on the board without overlapping another ship or going out of bounds. It considers the size and orientation (horizontal or vertical) of the ship.
- place-ship: Marks cells on the board to indicate where a ship has been placed. It uses set! to update the ships-placed-locations list with the ship's details.
- remove-ship: Removes the most recently placed ship, allowing players to revert their placement.

- place-opponent-ships: Randomly places ships on the opponent's board in single-player mode (against AI).

**Gameplay**

- player-guess: Handles the player's attempt to hit an opponent's ship. It converts the mouse click to board coordinates, checks the cell status, and updates the board (hit or miss).
- opponent-guess: Simulates the AI's random guessing in single-player mode.
- check-game-over: Iterates over the board to count ship parts and checks if all parts have been hit, indicating the end of the game.

**Drawing Functions**

- draw-grid: Draws the board's grid lines and fills cells based on their status (hit, miss, ship present). This function is called for both players' boards during the game.
- mouse-in?: Checks if a mouse click occurred within a specified rectangular area.
- mouse-to-board: Converts the mouse position into grid coordinates for ship placement and guesses.
- center-text: Approximates the horizontal centering of text within a given area.
- draw: Draws the game screen based on currentState. It includes drawing buttons, text, and game grids.

**Game Loop**

- game-loop: The central loop of the game that runs continuously, calling update to process game state changes and draw to update the visuals.
- run: Starts the game loop with specified window dimensions (800x900) and frame rate (60 FPS).

**User Interaction**

- Mouse clicks are used to select game modes, place ships, and make guesses. The functions mouse-in? and mouse-to-board handle the conversion of mouse events to game actions.
- The game supports toggling ship orientation using the LEFT arrow key, which is handled during the

ship placement state.

**Future Considerations**

- Game Modes: Add different AI difficulty levels or online multiplayer functionality.

- Enhanced UI: Additional visual feedback during gameplay (e.g., highlighting possible ship placement, animating hits/misses).

- Game State Management: Refactor game states into a more modular structure, possibly using a state pattern or event-driven model.

- Scoring and Victory Conditions: Implement a detailed scoring system and more robust victory conditions.

**Detailed Function Descriptions createBoard(size)**

Creates a game board represented by a 2D vector of size `size`. Each cell in the board is initialized to #f, indicating an empty cell. The function returns this vector that represents the board.

**draw-grid(x-offset, y-offset, board, showShips)**

Draws the grid on the screen based on the board's current state. It iterates through each cell of the board to draw grid lines. If `showShips` is true, it fills cells with ships, represented by #t. It uses different colors to represent hits, misses, and ships.

**mouse-in?(mx, my, x, y, width, height)**

Checks if a mouse click (given by `mx`, `my`) is within a specified rectangular area (`x`, `y`, `width`, `height`). Returns true if the click is within the area, otherwise false. Used to detect button clicks in the game.

**mouse-to-board(mx, my, x-offset, y-offset)**

Converts the mouse position (`mx`, `my`) to board coordinates based on the provided `x-offset` and `y-offset`. Returns a pair (row, col) representing the board coordinates or #f if the coordinates are

out of bounds.

**can-place-ship?(board, row, col, size, orientation)**

Checks if a ship of a given `size` can be placed on the board at the specified `row`, `col`, without overlapping other ships or going out of bounds. The ship can be oriented either horizontally or vertically. Returns true if placement is valid, otherwise false.

**place-ship(board, row, col, size, orientation)**

Places a ship of the specified `size` on the board at the given `row`, `col`. The `orientation` determines if the ship is placed horizontally or vertically. Updates the board to mark the cells occupied by the ship.

**place-opponent-ships(ship-sizes)**

Automatically places ships on the opponent's board randomly in single-player mode (Player vs AI). It tries to place each ship in the `ship-sizes` list until a valid placement is found.

**remove-ship(board, ship)**

Removes the most recently added ship from the board. Uses the `ship` information (row, col, size, orientation) to unmark the cells occupied by the ship. Also updates the list of placed ships (`ships-placed-locations`).

**coinToss()**

Randomly selects which player starts the game first. For two-player mode, it randomly chooses between Player 1 and Player 2. For Player vs AI mode, it randomly assigns the first turn to either the player or the AI.

**print-board(board)**

Prints the current state of the board to the console for debugging purposes. Displays each row of the board, which can show hits, misses, and ships.

**player-guess(board, mouseX, mouseY)**

Handles the player's guess on the opponent's board. Converts the mouse click to board coordinates and checks if the guessed cell is a hit or miss. Marks the cell as 'hit' or 'miss' on the board.

**opponent-guess(board)**

Simulates the AI's guess in Player vs AI mode. It randomly selects a cell on the player's board and marks it as 'hit' or 'miss', ensuring the cell has not been guessed before.

**check-game-over(board)**

Checks if all the ship parts on the board have been hit, indicating the end of the game. It counts the total number of ship parts and the number of hit parts to determine the game-over condition.

**update(state)**

The core function that handles the game logic based on the current state (`currentState`). It checks for user inputs, such as mouse clicks, and updates the game state accordingly. This includes navigating menus, placing ships, making guesses, and handling turn switching during gameplay.

**center-text(x, y, width, text-str)**

Calculates the position to draw text centered within a given width. This is used to display text messages, such as button labels, in a visually centered manner.

**draw(state)**

Renders the game's visual elements based on the current state (`currentState`). It includes drawing menus, grids, and text on the screen, as well as displaying buttons and game instructions. This function ensures the game is displayed correctly at each stage.

**game-loop()**

The main loop of the game that is executed continuously. It calls the `update` function to process game state changes and the `draw` function to render the game visuals. Runs at a specified frame rate (60 FPS) to maintain smooth gameplay.