OSPROJECT-DOCUMENTATION

FUNCTION PLAYER (908 - 955):

SUMMARY

The `player` function is designed to handle the gameplay logic for a single player in a multi-threaded board game simulation. It takes a pointer to an integer as an argument, which represents the player number (`p_no`). The function operates in an infinite loop, suggesting it continues until externally terminated, likely when the game ends.

Here's a breakdown of the function's operations:

1. **Semaphore Lock for Dice Rolling**: The function begins by acquiring a semaphore lock (`dicesem`) to ensure exclusive access to the dice rolling process. This prevents other threads (players) from rolling the dice simultaneously.
2. **Dice Rolling**: It calls the `rollDice` function to simulate the rolling of dice. This function updates global variables `dice` , `dice2` , and `dice3` , which represent the results of dice rolls.
3. **Semaphore Release**: After rolling the dice, it releases the semaphore (`dicesem`), allowing other players to roll the dice.
4. **Sleep**: The thread sleeps for 1 second, likely to simulate time taken for a real player to make a move.
5. **Token Movement Semaphore**: It then acquires another semaphore (`tokensem`) to manage the movement of tokens on the board, ensuring that token movements from different players do not interfere with each other.
6. **Setting Current Player**: The player number (`p_no`) is set to `currentTurnIndex` , indicating whose turn it is.
7. **Move Execution**: The function enters a loop to handle the player's moves based on the dice results. If triple sixes are rolled (`dice == 6 && dice2 == 6 && dice3 == 6`), it sets a counter to 3, indicating a special game rule where the player might have additional privileges or penalties.
8. **Handling Key Presses**: Inside the loop, it checks if a key has been pressed (`keypressed`). If a valid key corresponding to a token number is pressed, it calls `moveToken` to move the appropriate token. The keypress is then reset to 0.
9. **Move Count Management**: Depending on the dice values, the move count (`count`) is incremented or set to 3 to exit the loop, ensuring the player's turn ends after a certain number of moves.
10. **Turn End and Semaphore Release**: After handling all moves, it sets `newTurn` to true, indicating the end of the current player's turn, and releases the token movement semaphore (`tokensem`).

11. **Sleep**: The thread sleeps for another second before the next iteration of the loop, simulating the end of the turn.
12. **Thread Exit**: The function is designed to run indefinitely until the thread is externally terminated (`pthread_exit(NULL)`), which would occur when the game ends.

This function is a critical component of the game's concurrency management, ensuring that dice rolls and token movements are handled in an orderly and conflict-free manner.

FUNCTION MASTER THREAD (957 - 1017):

SUMMARY

The `masterthread` function manages the game's main logic and player threads in a board game simulation. Here's a detailed breakdown of its operations:

1. **Thread Creation**: The function starts by creating four player threads using `pthread_create` . Each thread is passed a unique player number (`p_no`) which is dynamically allocated to ensure each thread has a unique context. These threads execute the `player` function.
2. **Thread Detachment**: After creating the threads, the function detaches them using `pthread_detach` . This allows the threads to run independently and have their resources automatically reclaimed upon termination.
3. **Game Logic Loop**: The function enters an infinite loop (`while(true)`) to continuously check the game state and manage player threads based on the game's progress.
4. **Check Tokens and Manage Threads**:
    - The function iterates over each player's house (`allHouses[i]`) to check if the `tokensRemaining` for any house is zero, indicating that the player has finished the game.
    - If a player finishes, the function updates the player's position in the game ranking (`temp_position`), marks the player as done (`player_done[i] = true`), and resets `tokensRemaining` to 4 for potential future games.
    - It outputs the player's details and cancels the corresponding player thread using `pthread_cancel` .
    - After handling a finished player, it sets `newTurn` to true and calls `rollDice` to potentially update the dice values for the next turn.
5. **Check Game Completion**:
    - After processing individual players, the function counts how many players have finished the game (`counter`).
    - If three players have finished, it prints the final positions of all players and exits the entire program using `exit(0)` , effectively ending the game and terminating the master thread with `pthread_exit(NULL)` .
6. **Reset Counter**: If not enough players have finished to end the game, the counter is reset to zero, and the loop continues.

This function is central to managing the game's state, handling player threads, and determining when the game ends based on the players' progress.

FUNCTION KILLTOKEN (857 - 903):

SUMMARY

The `killToken` function is designed to simulate the action of a token "killing" or capturing another token in a board game, based on specific game rules. Here's a detailed breakdown of its functionality:

1. **Input Parameters**: The function takes two parameters:
   - `color`: A string representing the color of the house whose token is attempting to kill another token.
   - `key`: An integer representing the index of the token within its house that is attempting the kill.
2. **Outer Loop (House Identification)**: The function starts by iterating over the `allHouses` array to find the house that matches the given `color`. This is done through a loop that checks each house's color against the input `color`.
3. **Inner Loops (Token Interaction)**:
   - **House Exclusion**: Once the correct house is identified, another loop iterates over all houses again to check interactions between the selected token and tokens from other houses. It skips the iteration if it's the same house (`j == i`).
   - **Token Safety Check**: For each token in the other houses, the function first checks if either the attacking token or the target token is in a "safe house" state. If so, the interaction is skipped.
   - **Position Check for Special Tiles**: The function checks if the attacking token is on specific board positions that are considered safe (0, 8, 13, 21, 26, 34, 39, 47). If the token is on any of these positions, it skips to the next token.
   - **Kill Logic**: If the positions of the attacking token and a target token match (modulo 52, to wrap around the board), the target token is "killed":
     - The position and iterator of the killed token are reset.
     - The token is marked as not free (`free = false`).
     - The token's coordinates are reset to its starting position.
     - The number of tokens stuck in the house is incremented.
     - The hit rate of the attacking house is incremented, and a message is printed showing the new hit rate.
4. **Termination**: Once a token is killed, the innermost loop breaks, and the function exits the loops for checking other houses and tokens, as the action for that turn is complete.

This function encapsulates the logic necessary for one token to interact with others on the board in a competitive manner, adhering to the rules of safe zones and direct position conflicts.

FUNCTION DRAWDICE (802 - 835):

SUMMARY:

The function `drawDice()` is responsible for rendering the dice values and the current turn information on the screen in a graphical user interface. Here's a breakdown of its functionality:

1. **Check Current Turn**: The function first checks if `currentTurnIndex` is not equal to `-1`, which indicates that there is an active turn. If there is no active turn, the function does not proceed with drawing anything.
2. **Display Current Turn Information**:
   - It displays the word "Current" at coordinates (800, 40) and "Turn:" at (820, 80) using a font size of 30 and black color.
   - It retrieves the name of the player whose turn it is from the `currentTurn` array using `currentTurnIndex` and displays this name at (820, 120).
3. **Display Player Hit Rates**:
   - It iterates through the `names` array (which holds the names of the players) and for each player, it displays their name and their hit rate (from `allHouses[i].hitrate`) on the screen. This is done in a loop where the y-coordinate is incremented by 40 for each player, starting at 160.
4. **Render Dice Values**:
   - If `dice2` is non-zero, it draws a rectangle at (800, 460) and displays the value of `dice2` inside this rectangle at (830, 475).
   - Similarly, if `dice3` is non-zero, it draws another rectangle at (800, 560) and displays the value of `dice3` at (830, 575).
   - Regardless of the values of `dice2` and `dice3`, it always draws a rectangle at (800, 360) to display the value of `dice` at (830, 375).

The function uses the `DrawText` function to render text and `DrawRectangle` to draw rectangles, which are presumably part of a graphics library (like raylib, as suggested by the use of `DrawRectangle` and `DrawText` functions). This function is crucial for providing visual feedback to the players about the current state of the game, including whose turn it is and the results of the dice rolls.

FUNCTION DRAWPIECE (719 - 800):

SUMMARY:

The function `drawPieces()` is responsible for rendering the game pieces on the board in a graphical user interface using the Raylib library. It defines two circle sizes, `outline` and `inner`, to create a visual effect where each game piece has a colored center and a white border.

The function iterates over the tokens of each house (Red, Green, Yellow, Blue) and draws them on the board. The number of tokens drawn depends on the value of `totalTokens`, which determines how many tokens each player has in the game. For each token, the function draws two circles (one for the outline and one for the inner color) and a number on the token that corresponds to the token's index plus one.

The drawing is done using several Raylib functions:

- `DrawCircle()` is used to draw the outline and the inner colored circle of each token.
- `DrawText()` is used to label each token with a number.

The function checks the number of tokens (`totalTokens`) to decide how many tokens to draw for each house. It supports up to four tokens per house. The coordinates for each token are retrieved from the `tokens` array within each `House` object, which stores the x and y coordinates for each token.

This function is a critical part of the game's visual representation, ensuring that players can see the positions of their tokens on the game board.

FUNCTION DRAWBOARD (603 - 717):

SUMMARY

The `drawBoard` function is responsible for rendering the visual elements of a board game using the Raylib library. It primarily draws the game board, which includes the houses, center area, special tiles within each house, and grid lines.

1. **Houses**: The function draws four colored rectangles representing the houses for the players. These are:
    - Green house at the top-left corner.
    - Red house at the top-right corner.
    - Yellow house at the bottom-left corner.
    - Blue house at the bottom-right corner.
2. **Center Area**: The center of the board is decorated with triangles that point towards each house, colored accordingly (Green, Red, Blue, and Yellow).
3. **House Special Tiles**: Within each house, specific tiles are highlighted with different colors. These special tiles are drawn using smaller rectangles within the larger house rectangles.
4. **Grid Lines**: The function also draws grid lines within each house to visually separate the tiles. This is done using horizontal and vertical lines.

Overall, the function uses a combination of `DrawRectangle`, `DrawTriangle`, and `DrawLine` functions from the Raylib library to create a visually structured board game layout. This setup is crucial for players to navigate and interact with the game effectively.

FUNCTION ABDOLUTE_POSITION (483 - 601):

SUMMARY

The `drawBoard` function is responsible for rendering the visual elements of a board game using the Raylib library. It primarily draws the game board, which includes the houses, center area, special tiles within each house, and grid lines.

1. **Houses**: The function draws four colored rectangles representing the houses for the players. These are:
   - Green house at the top-left corner.
   - Red house at the top-right corner.
   - Yellow house at the bottom-left corner.
   - Blue house at the bottom-right corner.
2. **Center Area**: The center of the board is decorated with triangles that point towards each house, colored accordingly (Green, Red, Blue, and Yellow).
3. **House Special Tiles**: Within each house, specific tiles are highlighted with different colors. These special tiles are drawn using smaller rectangles within the larger house rectangles.
4. **Grid Lines**: The function also draws grid lines within each house to visually separate the tiles. This is done using horizontal and vertical lines.

Overall, the function uses a combination of `DrawRectangle`, `DrawTriangle`, and `DrawLine` functions from the Raylib library to create a visually structured board game layout. This setup is crucial for players to navigate and interact with the game effectively.

STRUCTURE HOUSE (78 - 479):

SUMMARY

A `House` structure that represents a player's house in a board game. Each `House` has several attributes and methods to manage the game's logic related to the movement and interaction of tokens within the game.

Attributes:

- `color`: A string representing the house's color.
- `tokensRemaining`: An integer tracking the number of tokens that haven't completed their circuit on the board.
- `endHousePosition`: A 2D array storing coordinates for the positions within the house's end zone.
- `tokensStuck`: An integer counting the tokens that are still in the starting area.

- `tokens`: A dynamic array of `Token` objects representing the tokens belonging to the house.
- `startHouseCoordinates`: A 2D array holding the starting coordinates for each token.
- `hitrate`: An integer tracking how many times tokens from this house have sent other tokens back to their start.
- `position`: An integer likely used to track the house's position in some order of play.

Constructor:

- Initializes `hitrate`, `tokensRemaining`, `tokensStuck`, and `position`.
- Allocates memory for the `tokens` array based on the `totalTokens` variable, which seems to be a global variable indicating the number of tokens each player starts with.

Methods:

- `setColor(string c)`: Sets the house's color and initializes token positions and coordinates based on the house color. This method also sets up the starting coordinates for each token depending on the house's color.
- `getColor()`: Returns the house's color.
- `moveToken(int key)`: Manages the movement of a token based on dice rolls (`dice`, `dice2`, `dice3`). This method handles different scenarios such as freeing a token from the start, moving a token along the board, entering a safe house, and completing the game path. It also calls `killToken` to handle interactions with other tokens on the board.
- `killToken(string color, int key)`: A function that checks if a token can "kill" (send back to start) another token based on the board rules. It iterates through all tokens of all houses and adjusts their positions and states if certain conditions are met.

Overall, the code snippet is part of a larger game application where multiple players can interact through tokens on a game board, with rules enforced by the methods within the `House` structure. The logic includes movement based on dice rolls, interactions between tokens from different houses, and tracking of game progress through various counters and state flags.

STRUCTURE TOKEN (27 - 76):

SUMMARY

```
The provided C++ code defines a `struct` named `Token` which represents a
game token with various attributes and methods to manipulate and access its
state. Here's a breakdown of its components:
```

1. **Attributes**:

- `int position`: Stores the current position of the token on the game board.
- `bool free`: Indicates whether the token is free to move on the board.
- `bool stacked`: Indicates whether the token is stacked with another token.
- `int iterator`: Used to track additional movement or specific iterations for the token.
- `bool insideSafeHouse`: Indicates whether the token is inside a safe house, a protected area on the board.
- `int xCoordinates, yCoordinates`: Store the graphical coordinates of the token for rendering purposes.
- `bool ended`: Indicates whether the token has completed its journey on the board.

2. **Constructor (`Token()`)**:
   - Initializes `ended`, `insideSafeHouse`, `iterator`, `position`, `free`, and `stacked` to their default values, setting up the token in an initial, inactive state.
3. **Methods**:
   - `bool getFree()`: Returns the `free` status of the token.
   - `bool getStacked()`: Returns the `stacked` status of the token.
   - `int getPosition()`: Returns the current `position` of the token on the board.
   - `void setPosition(int x)`: Sets the `position` of the token.
   - `void setFree(bool val)`: Sets the `free` status of the token.

This structure is designed to encapsulate all necessary data and behaviors related to a game token within a board game, allowing for easy manipulation and status checks through its methods.