System Programing Assignment #2

2022320039 컴퓨터학과 정진욱

Problem 1. Blokus

Feature Definition

The implementation of the Blokus game consists of the following features:

1. Board Creation and Display

- **Purpose**: To represent the game state visually for players.
- **Implementation**:
  - A 9x9 grid (`board`) of `Player` enum values is used to track the board state.
  - Each cell can be `Player::P1`, `Player::P2`, or `Player::Empty`.
  - The `print_board` method displays the grid with appropriate markers (`O`, `@`, `_`).

2. Block Representation and Random Generation

- **Purpose**: To simulate block placement using predefined block types with random selection.
- **Implementation**:
  - **Block List**: `BLOCK_LIST` is a constant array of 14 blocks, each with 4 rotations, represented by `Point` structures. Every blocks and each rotation is predefined in this list for easily utilize blocks in other functions.
  - **Random Selection**: The `generate_block` function selects a random block (`b_id`) and rotation (`rotate`) using the `rand` crate.

3. Block Placement

- **Purpose**: To allow players to place blocks on the board following game rules.
- **Implementation**:
  - Players input coordinates (`r, c`) to place their blocks.
  - The `place_block` method validates whether the block can fit:
    - Ensures all block cells are within bounds.
    - Checks that all cells are unoccupied (`Player::Empty`).
    - Using closure `is_valid`
  - Utilize `is_valid` with iterator.
  - If valid, the block is placed; otherwise, an error is shown.

4. Block Rotation

- **Purpose**: To allow players to adjust the orientation of blocks.
- **Implementation**:
  - The `rotate_block` method updates the `rotate` value of the current block (`current_block.rotate = (rotate + 1) % 4`).
  - Rotated shapes are retrieved dynamically from `BLOCK_LIST`.

5. Turn Management

- **Purpose**: To alternate turns between Player 1 and Player 2.
- **Implementation**:
  - The `swap_turn` method switches the `current_player` between `Player::P1` and `Player::P2` and assigns a new random block for the next player.

6. Win/Loss Determination

- **Purpose**: To identify when a player loses the game.

- **Implementation**:
    - The `is_loss` method checks if the current block can be placed anywhere on the board.
    - It iterates over all board positions and all block rotations using the helper `can_place_block`.

```
fn can_place_block(&self, block_id: usize, rotation: usize, start_row: i32, start_col: i32) -> bool { let
block = Self::get_block(block_id, rotation); let is_valid = |row: i32, col: i32| row >= 0 && row < 9 &&
col >= 0 && col < 9 && self.board[row as usize][col as usize] == Player::Empty; if
block.iter().all(|point| is_valid(point.r + start_row - 1, point.c + start_col - 1)) { return true }
false // All points are within bounds and unoccupied }
```

    - `can_place_block` is implemented by closure `is_valid` to check the coordinates are valid in board.
    - Utilize `is_valid` with iterator.
    - If no valid placement exists, the current player loses.

Implementation Strategy

1. Data Structures

- **`Point` Struct**:
    - Represents individual cells of a block as `(row, column)` coordinates.
- **`Player` Enum**:
    - Tracks ownership of each board cell and is also used to identify the current player.
- **`Block` Struct**:
    - Represents a block using its type (`b_id`) and rotation (`rotate`).

2. Core Functionalities

a. Board Initialization

- A 9x9 `board` is initialized with `Player::Empty` in the `Game::new` method.

b. Block Placement Validation

- **Algorithm**:
    - Calculate the absolute position of each `Point` in the block based on the input `(r, c)`.
    - Use the `can_place_block` method to:
        - Verify that all cells of the block fit within the 9x9 grid.
        - Ensure no overlap with existing blocks.
- If placement is invalid, the turn is not advanced.

c. Block Rotation

- Rotations are handled using the `BLOCK_LIST` constant:
    - Each block has precomputed positions for four rotations.
    - Rotation is applied dynamically by adjusting the `rotate` index.

d. Random Block Generation

- Random blocks are assigned at the start of each turn using `rand::thread_rng`.

e. Loss Condition

- The `is_loss` method checks all positions on the board for each rotation of the current block:
    - Iterates over all grid cells.
    - For each cell, checks all four rotations of the block.
    - Exits early if a valid placement is found.

3. User Input Handling

- Input is read as `(r, c)` coordinates or a command to rotate (`0`).
- Errors from invalid input such as out of bound coordinates or point has been already placed, are handled gracefully with retry prompts.

4. Display Logic

- **Board Display**:
  - The `print_board` method visualizes the 9x9 grid with players' blocks.
- **Block Preview**:
  - The `print_block` method shows the current block in a 3x3 preview grid.

Description of `main.rs`

The `main.rs` file serves as the entry point for the Blokus game and implements the game's runtime loop, user interaction, and turn management. It leverages the methods and data structures defined in `lib.rs` to provide a seamless gameplay experience.

1. Game Initialization

The program begins by initializing a new game using `Game::new()`. This sets up:

- An empty 9x9 game board.
- A random block assigned to Player 1 (`Player::P1`) as the first player.

2. Game Loop

The game operates within a `while` loop that runs until a loss condition is met (`game.is_loss()` returns `true`)

3. User Input Handling

Inside the loop, the program prompts the current player for input to:

1. **Rotate the Block**:
   - If the player inputs `0`, the block's rotation is updated using `rotate_block`.
2. **Place the Block**:
   - Players input coordinates `(r, c)` to place their block.
   - The coordinates are parsed and validated. If invalid (e.g., out of bounds or overlapping), an error message is displayed, and the player must try again.

Input handling logic ensures:

- Graceful handling of invalid inputs (e.g., non-integer or improperly formatted strings).
- Players cannot proceed until a valid action is taken.

4. Block Placement

Once valid coordinates are provided, `game.place_block(r, c)` is called:

- **Successful Placement**:
  - The block is placed on the board (`return None`), and the turn ends.
- **Failed Placement**:
  - If the block cannot be placed at the given coordinates, the game prints an error message and prompts the player to try again.

5. Turn Management

After a successful block placement:

- The `swap_turn` method alternates the current player (`Player::P1` ⇔ `Player::P2`).
- A new random block is assigned to the next player.

6. Game Over Condition

When the game loop exits:

- The board and the last player's block are displayed one final time.

- The `print_ending` method announces the winner, specifying which player failed to place their block.

Testing

Unit tests are included to verify core functionalities:

- **Board Initialization:**
    - Ensures the board starts empty.
- **Block Placement:**
    - Tests valid and invalid placements for blocks.
- **Turn Switching:**
    - Confirms proper player alternation.
- **Game Over Condition:**
    - Tests valid and invalid game over condition.