Contents

Game Class Documentation 3
Overview 3
Methods 4
start 4
current_player 4
next_turn 4
is_valid_move 4
remove_card 5
update_allowed_coords_after_action_card 5
check 6
check_card 6
player_cards_update 6
get_allowed_coords 6
is_connected 6
get_oriented_matrix 7
find_path_a_star 7
add_item 7
update_allowed_coords 7
check_finish_card 8
get_opponents 8
set_gold_card 8
winner_gold 8
Dealer Class Documentation 9
Overview 9
Functions 9
shuffle_array 9
get_random_token 9
Class: Dealer 10
Methods 10
start 10
deal_player_cards 11
make_path_action_cards 11

deal_game_cards 11
show 11
get_card 12
get_gold_cards 12
pop_gold 12
Board, Card, Lobby, and Player Classes Documentation 13
Class: Board 13
Methods 14
to_json 14
to_list 14
get_hidden_card 14
get_board 14
has_card_at 14
get_card_at 15
get_board_copy 15
remove_element 15
show 16
Class: Card 16
Methods 16
show 16
get_json 17
get_data 17
set_matrix 17
Class: Lobby 17
Methods 18
add_player 18
remove_player 18
get_players 18
get_player 18
is_full 19
is_ready 19
Class: Player 19
Properties 20
Methods 20

get_player_json 20
check_status 20
remove_card 20

Game Class Documentation

Overview

The Game class manages the core logic of a multiplayer card-based board game. It handles game initialization, player turns, card placement, action cards, pathfinding, and win conditions. The game involves players placing path cards to connect a start point to finish points or using action cards to affect gameplay.

Class: Game

Initialization

def __init__(self, lobby)

- **Description**: Initializes a new game instance.
- **Parameters**:
  - lobby (Lobby): The lobby object containing player information.
- **Attributes**:
  - players: List of players from the lobby.
  - dealer: Dealer instance for card management.
  - turn_index: Index of the current player's turn.
  - started: Boolean indicating if the game has started.
  - board: Board instance managing the game grid.
  - lobby: Reference to the lobby.
  - DIRECTIONS: Dictionary mapping directions (up, down, left, right) to coordinate offsets.
  - allowed_coords: Set of valid coordinates for card placement.
  - need_check_finish: List of coordinates to check for finish conditions.
  - need_check_finish_card: List of finish card details to verify.

- START_CARD: Tuple (0, 0) representing the starting point.
- FINISH_CARDS: List of finish point coordinates [(4, 2), (4, 0), (4, -2)].
- finish: Boolean indicating if the game has ended.
- winner: The winning player, if any.
- last_action_player: Last player affected by an action card.

Methods

start

def start(self)

- **Description**: Starts the game by dealing cards to players.
- **Actions**:
  - Initializes the dealer with the number of players.
  - Sets started to True.
  - Assigns each player a player card and a set of game cards.
- **Example**:
- game = Game(lobby)
- game.start() # Deals cards and starts the game

current_player

def current_player(self)

- **Description**: Returns the current player whose turn it is.
- **Returns**: The player object at turn_index. Advances to the next player if the current player has no cards.
- **Example**:
- player = game.current_player() # Gets the current player

next_turn

def next_turn(self)

- **Description**: Advances the turn to the next player.
- **Actions**: Increments turn_index modulo the number of players.

is_valid_move

def is_valid_move(self, player_id, move)

- **Description**: Validates a player's move, which can be a path card placement, action card usage, or discarding a card.

- **Parameters**:
    - player_id (str): ID of the player making the move.
    - move (str): JSON string containing move details (e.g., card ID, coordinates, or action type).
- **Returns**: True if the move is valid and executed, False otherwise.
- **Actions**:
    - Parses the move JSON.
    - Verifies the card and player exist.
    - Handles:
        - **Trash**: Discards the card.
        - **Path card**: Checks placement validity, updates the board, and verifies paths.
        - **Action card**: Executes actions like see_map, rockfall, or equipment modifications (break_cart, fix_lantern, etc.).
- **Example**:
- move = '{"card": "card1", "type": "trash"}'
- result = game.is_valid_move("player1", move) # Discards card, returns True

remove_card

def remove_card(self, player, card)

- **Description**: Removes a card from a player's hand and updates their cards.
- **Parameters**:
    - player (Player): The player object.
    - card (Card): The card to remove.
- **Actions**: Removes the card and deals a new one if available.

update_allowed_coords_after_action_card

def update_allowed_coords_after_action_card(self)

- **Description**: Updates valid coordinates for card placement after an action card is played.
- **Actions**: Clears allowed_coords, recalculates valid coordinates based on reachable paths.

check

def check(self, x, y, card)

- **Description**: Validates if a path card can be placed at coordinates (x, y).
- **Parameters**:
  - x (int): X-coordinate.
  - y (int): Y-coordinate.
  - card (Card): The card to place.
- **Returns**: True if the placement is valid, False otherwise.
- **Actions**: Checks if the card connects correctly with neighboring cards.

check_card

def check_card(self, id, player_id)

- **Description**: Verifies if a card belongs to a player's hand.
- **Parameters**:
  - id (str): Card ID.
  - player_id (str): Player ID.
- **Returns**: The card object if found, or an error message string.

player_cards_update

def player_cards_update(self, player)

- **Description**: Adds a new card to a player's hand if available.
- **Parameters**:
  - player (Player): The player to update.

get_allowed_coords

def get_allowed_coords(self)

- **Description**: Returns a list of valid coordinates for card placement.
- **Returns**: List of coordinate lists (e.g., [[0, 1], [1, 0]]).

is_connected

def is_connected(self, from_matrix, to_matrix, direction)

- **Description**: Checks if two card matrices are connected in a given direction.
- **Parameters**:
  - from_matrix (list): Matrix of the source card.

    - to_matrix (list): Matrix of the target card.
    - direction (str): Direction (up, down, left, right).
- **Returns**: True if connected, False otherwise.

get_oriented_matrix

def get_oriented_matrix(self, card)

- **Description**: Returns a card's matrix, accounting for rotation.
- **Parameters**:
    - card (Card): The card object.
- **Returns**: The oriented matrix or None if invalid.

find_path_a_star

def find_path_a_star(self, finishes=None)

- **Description**: Finds a path from the start card to a finish card using the A* algorithm.
- **Parameters**:
    - finishes (list, optional): List of finish coordinates. Defaults to FINISH_CARDS.
- **Returns**: List of coordinates representing the path, or None if no path exists.

add_item

def add_item(self, x, y, card)

- **Description**: Adds a card to the board at (x, y) and updates game state.
- **Parameters**:
    - x (int): X-coordinate.
    - y (int): Y-coordinate.
    - card (Card): The card to place.
- **Returns**: The updated board dictionary.
- **Actions**: Places the card, checks for finish conditions, and updates allowed coordinates.

update_allowed_coords

def update_allowed_coords(self, x, y, card)

- **Description**: Updates allowed_coords based on a placed card.
- **Parameters**:
    - x (int): X-coordinate.

    - y (int): Y-coordinate.
    - card (Card): The placed card.

check_finish_card

def check_finish_card(self, x, y, card)

- **Description**: Checks if a card at (x, y) is a winning finish card.
- **Parameters**:
    - x (int): X-coordinate.
    - y (int): Y-coordinate.
    - card (Card): The card to check.
- **Returns**: True if the card is a gold finish card, False otherwise.

get_opponents

def get_opponents(self, player)

- **Description**: Returns a list of players excluding the specified player.
- **Parameters**:
    - player (str/Player): The player or player ID.
- **Returns**: List of opponent player objects.

set_gold_card

def set_gold_card(self, gold_count, player_id)

- **Description**: Assigns gold cards to a player.
- **Parameters**:
    - gold_count (int): Number of gold cards.
    - player_id (str): Player ID.

winner_gold

def winner_gold(self) -> Optional[List[Dict]]

- **Description**: Determines the winner(s) based on gold card counts.
- **Returns**: List of winning players' JSON data, or None if no winners.
- **Actions**: Compares players' gold counts to find the highest.

Usage Example

```
lobby = Lobby()  # Assume Lobby is defined

game = Game(lobby)
```

```
game.start()

player = game.current_player()

move = '{"card": "card1", "type": "path", "turn": [1, 1]}'

if game.is_valid_move(player.id, move):

  print("Valid move executed")

game.next_turn()
```

Notes

- The game relies on external modules (game.board, game.dealer) and assumes a Lobby class for player management.
- Cards have types (path, action) and specific data (e.g., matrices for paths, actions like break_cart).
- The board uses a coordinate system with matrices to represent card connections.
- A* pathfinding ensures valid paths to finish points.

Dealer Class Documentation

Overview

The Dealer class manages card creation, shuffling, and distribution in a multiplayer card-based board game. It handles player role cards, path/action cards, and gold cards, ensuring randomized and fair distribution to players.

Functions

shuffle_array

```
def shuffle_array(arr)
```

- **Description**: Shuffles an array in place using the Fisher-Yates algorithm.
- **Parameters**:
    - arr (list): The array to shuffle.
- **Returns**: The shuffled array.
- **Example**:
- arr = [1, 2, 3, 4]
- shuffled = shuffle_array(arr) # Returns shuffled array, e.g., [3, 1, 4, 2]

get_random_token

```
def get_random_token(length=5)
```

- **Description**: Generates a random token string for card IDs.
- **Parameters**:
    - length (int, optional): Length of the token (default: 5).
- **Returns**: A random string of letters and digits.
- **Example**:
- token = get_random_token() # Returns a random 5-character string, e.g., "aB3k9"

Class: Dealer

Initialization

def __init__(self)

- **Description**: Initializes a new Dealer instance.
- **Attributes**:
    - players: List of players (initially empty).
    - path_actions_cards: List of path and action cards.
    - player_cards: List of role cards (e.g., saboteur or miner).
    - player_count: Number of players in the game.
    - gold_cards: List of gold card dictionaries with properties:
        - img: Image file name (e.g., gold1.jpg).
        - type: Always "gold".
        - matrix: Empty list.
        - golds: Integer value (1, 2, or 3).
        - action, player, start, finish: Default values.
    - gold_cards_for_players: List of gold cards assigned to players.

Methods

start

def start(self, player_count)

- **Description**: Initializes the dealer for a game with a specified number of players.
- **Parameters**:
    - player_count (int): Number of players.
- **Actions**:
    - Sets player_count.

    - Calls deal_player_cards to create role cards.
    - Calls make_path_action_cards to prepare path/action cards.
- **Example**:
- dealer = Dealer()
- dealer.start(3) # Sets up cards for 3 players

deal_player_cards

def deal_player_cards(self)

- **Description**: Creates and shuffles player role cards based on player count.
- **Actions**:
    - For 3 players: Creates one saboteur card and two miner cards.
    - Shuffles the player_cards list.
- **Example**:
- dealer.deal_player_cards() # Creates and shuffles role cards

make_path_action_cards

def make_path_action_cards(self)

- **Description**: Creates path and action cards from a predefined cards list.
- **Actions**:
    - Iterates through cards (from game.cards_data), creating Card objects for path and action types.
    - Shuffles the path_actions_cards list.
- **Example**:
- dealer.make_path_action_cards() # Populates and shuffles path/action cards

deal_game_cards

def deal_game_cards(self)

- **Description**: Deals three path/action cards to a player.
- **Returns**: A list of up to three Card objects, or fewer if the deck is depleted.
- **Example**:
- cards = dealer.deal_game_cards() # Returns list of 3 cards, e.g., [Card(...), Card(...), Card(...)]

show

def show(self)

- **Description**: Debugging method to print player cards and their details.
- **Actions**: Prints player_cards, player_count, and calls show on each player card.
- **Example**:
- dealer.show() # Prints card details for debugging

get_card

def get_card(self)

- **Description**: Retrieves one card from the path/action deck.
- **Returns**: A Card object if available, otherwise None.
- **Example**:
- card = dealer.get_card() # Returns a card or None if deck is empty

get_gold_cards

def get_gold_cards(self)

- **Description**: Assigns random gold cards to players.
- **Returns**: A list of gold card dictionaries, one per player (minus one).
- **Actions**: Randomly selects gold cards for player_count - 1 players.
- **Example**:
- gold_cards = dealer.get_gold_cards() # Returns list of gold cards, e.g., [{"golds": 1}, {"golds": 2}]

pop_gold

def pop_gold(self, gold_count)

- **Description**: Retrieves a gold card with a specific gold value from gold_cards_for_players.
- **Parameters**:
    - gold_count (int): The gold value to match (1, 2, or 3).
- **Returns**: A list containing the matching gold card, or an empty list if none found.
- **Example**:
- gold_card = dealer.pop_gold(2) # Returns [{"golds": 2}] or []

Usage Example

dealer = Dealer()

dealer.start(3) # Initialize for 3 players

player_cards = dealer.player_cards # Access role cards

game_cards = dealer.deal_game_cards() # Deal 3 cards to a player

gold_cards = dealer.get_gold_cards() # Assign gold cards

card = dealer.get_card() # Get a single card

Notes

- Relies on game.cards_data for card definitions and game.card for the Card class.
- Player role cards include one saboteur and multiple miners for a 3-player game.
- Gold cards have fixed values (1, 2, or 3 golds) and are assigned randomly.
- The show method is for debugging and may not be used in production.

Board, Card, Lobby, and Player Classes Documentation

Overview

This documentation covers the Board, Card, Lobby, and Player classes, which are core components of a multiplayer card-based board game. The Board manages the game grid, Card represents game cards, Lobby handles player management, and Player tracks individual player states and actions.

Class: Board

Overview

The Board class manages the game grid, including card placement, hidden finish cards, and board state serialization.

Initialization

def __init__(self)

- **Description**: Initializes a new board with predefined start and finish cards.
- **Attributes**:
    - BOARD: Dictionary mapping coordinates (x, y) to Card objects, initialized with:
        - (0, 0): Start card.
        - (4, 2), (4, 0), (4, -2): Finish cards (two stone, one gold).
    - show_finish: List of finish card coordinates that are revealed.
    - HIDDEN_CARDS: List of finish card coordinates [(4, 2), (4, 0), (4, -2)].

Methods

to_json

def to_json(self, board)

- **Description**: Converts a board dictionary to JSON format.
- **Parameters**:
    - board (dict): Board dictionary mapping coordinates to cards.
- **Returns**: Dictionary with string keys (e.g., "x,y") and card JSON data.
- **Example**:
- json_board = board.to_json(board.BOARD) # Returns {"0,0": {...}, "4,2": {...}, ...}

to_list

def to_list(self)

- **Description**: Converts the board to a list of dictionaries.
- **Returns**: List of dictionaries with x, y, and value (card JSON).
- **Example**:
- board_list = board.to_list() # Returns [{"x": 0, "y": 0, "value": {...}}, ...]

get_hidden_card

def get_hidden_card(self)

- **Description**: Creates a placeholder card for hidden finish cards.
- **Returns**: A Card object with type "path" and empty matrix.
- **Example**:
- hidden = board.get_hidden_card() # Returns Card("hidden", ...)

get_board

def get_board(self)

- **Description**: Returns a JSON representation of the board, hiding unrevealed finish cards.
- **Returns**: JSON dictionary with hidden cards replaced by get_hidden_card.
- **Example**:
- board_json = board.get_board() # Returns board with hidden finish cards

has_card_at

def has_card_at(self, x, y)

- **Description**: Checks if a card exists at coordinates (x, y).

- **Parameters**:
    - x (int): X-coordinate.
    - y (int): Y-coordinate.
- **Returns**: True if a card exists, False otherwise.
- **Example**:
- exists = board.has_card_at(0, 0) # Returns True

get_card_at

def get_card_at(self, x, y)

- **Description**: Retrieves the card at coordinates (x, y).
- **Parameters**:
    - x (int): X-coordinate.
    - y (int): Y-coordinate.
- **Returns**: The Card object or None if no card exists.
- **Example**:
- card = board.get_card_at(0, 0) # Returns start card

get_board_copy

def get_board_copy(self)

- **Description**: Returns a deep copy of the board.
- **Returns**: A copy of the BOARD dictionary.
- **Example**:
- board_copy = board.get_board_copy() # Returns deep copy of BOARD

remove_element

def remove_element(self, x, y)

- **Description**: Removes a card at coordinates (x, y).
- **Parameters**:
    - x (int): X-coordinate.
    - y (int): Y-coordinate.
- **Returns**: True if a card was removed, False otherwise.
- **Example**:
- removed = board.remove_element(1, 1) # Returns True if card existed

show

def show(self, size=9)

- **Description**: Prints a text representation of the board for debugging.
- **Parameters**:
  - size (int, optional): Grid radius (default: 9).
- **Actions**: Displays a grid with symbols (S for start, G for gold, # for stone, . for empty).
- **Example**:
- board.show() # Prints board grid

Class: Card

Overview

The Card class represents a game card with properties like type, data, and rotation state.

Initialization

def __init__(self, id, type, card_data)

- **Description**: Initializes a new card.
- **Parameters**:
  - id (str): Unique card identifier.
  - type (str): Card type (e.g., "path", "action", "player").
  - card_data (str): JSON string containing card details.
- **Attributes**:
  - type: Card type.
  - card_data: JSON string of card data.
  - is_rotated: Boolean indicating if the card is rotated.
  - id: Card ID.

Methods

show

def show(self)

- **Description**: Prints card details for debugging.
- **Actions**: Outputs type, card_data, and is_rotated.

- **Example**:
- card.show() # Prints card details

get_json

def get_json(self)

- **Description**: Returns the card's data in JSON format.
- **Returns**: Dictionary with id, type, card_data, and is_rotated.
- **Example**:
- card_json = card.get_json() # Returns {"id": "start", "type": "path", ...}

get_data

def get_data(self)

- **Description**: Parses and returns the card's JSON data.
- **Returns**: Dictionary of card data.
- **Example**:
- data = card.get_data() # Returns {"img": "start.jpg", "type": "path", ...}

set_matrix

def set_matrix(self, new_matrix)

- **Description**: Updates the card's matrix in its data.
- **Parameters**:
    - new_matrix (list): New matrix to set.
- **Actions**: Modifies the matrix field in card_data.
- **Example**:
- card.set_matrix([[0, 1, 0], [1, 1, 1], [0, 1, 0]])

Class: Lobby

Overview

The Lobby class manages a group of players before the game starts.

Initialization

def __init__(self, lobby_id, max_players=3)

- **Description**: Initializes a new lobby.
- **Parameters**:

    - lobby_id (str): Unique lobby identifier.
    - max_players (int, optional): Maximum number of players (default: 3).
- **Attributes**:
    - lobby_id: Lobby ID.
    - max_players: Maximum players allowed.
    - players: List of Player objects.

Methods

add_player

def add_player(self, player)

- **Description**: Adds a player to the lobby.
- **Parameters**:
    - player (Player): Player to add.
- **Returns**: True if added, False if lobby is full.
- **Example**:
- lobby.add_player(Player("p1")) # Returns True if not full

remove_player

def remove_player(self, player_id)

- **Description**: Removes a player by ID.
- **Parameters**:
    - player_id (str): ID of the player to remove.
- **Example**:
- lobby.remove_player("p1") # Removes player with ID "p1"

get_players

def get_players(self)

- **Description**: Returns the list of players.
- **Returns**: List of Player objects.
- **Example**:
- players = lobby.get_players() # Returns [Player(...), ...]

get_player

def get_player(self, id)

- **Description**: Retrieves a player by ID.
- **Parameters**:
    - id (str): Player ID.
- **Returns**: Player object or False if not found.
- **Example**:
- player = lobby.get_player("p1") # Returns Player or False

is_full

def is_full(self)

- **Description**: Checks if the lobby is full.
- **Returns**: True if player count equals max_players, False otherwise.
- **Example**:
- full = lobby.is_full() # Returns True if 3 players

is_ready

def is_ready(self)

- **Description**: Checks if all players are ready.
- **Returns**: True if all players have is_ready set to True, False otherwise.
- **Example**:
- ready = lobby.is_ready() # Returns True if all players are ready

---

Class: Player

Overview

The Player class represents a game participant, managing their state, cards, and equipment.

Initialization

def __init__(self, id)

- **Description**: Initializes a new player.
- **Parameters**:
    - id (str): Unique player identifier.
- **Attributes**:
    - _id: Player ID.

- _is_ready: Boolean indicating readiness.
- _gold_cars: List of gold cards.
- _cards: List of game cards.
- _status: Player status ("Normal" or "Blocked").
- _player_card: Role card (e.g., saboteur or miner).
- _see_card: Card the player has revealed.
- pickaxe, lantern, cart: Booleans indicating equipment state.

Properties

- id: Gets the player ID.
- is_ready: Gets/sets the readiness state (must be boolean).
- gold_cars: Gets/sets the list of gold cards (must be list).
- cards: Gets/sets the list of game cards (must be list).
- status: Gets/sets the player status (must be "Normal" or "Blocked").
- player_card: Gets/sets the role card.
- see_card: Gets/sets the revealed card.

Methods

get_player_json

def get_player_json(self)

- **Description**: Returns player data in JSON format.
- **Returns**: Dictionary with id, pickaxe, lantern, and cart.
- **Example**:
- json_data = player.get_player_json() # Returns {"id": "p1", "pickaxe": True, ...}

check_status

def check_status(self)

- **Description**: Updates player status based on equipment.
- **Actions**: Sets status to "Normal" if all equipment (cart, pickaxe, lantern) is True, otherwise "Blocked".
- **Example**:
- player.check_status() # Updates status

remove_card

def remove_card(self, rm_card)

- **Description**: Removes a card from the player's hand.
- **Parameters**:
    - rm_card (Card): Card to remove.
- **Returns**: True if removed, False if not found.
- **Example**:
- removed = player.remove_card(card) # Returns True if card was in hand

---

Usage Example

```
# Create lobby and players
lobby = Lobby("lobby1")
player = Player("p1")
lobby.add_player(player)

# Initialize board
board = Board()
board_json = board.get_board()  # Get board with hidden finish cards

# Create and manipulate a card
card = Card("c1", "path", json.dumps({"img": "path.jpg", "type": "path", "matrix": [[0, 1, 0], [1, 1, 1], [0, 1, 0]]}))
card.set_matrix([[0, 0, 0], [1, 1, 1], [0, 0, 0]])
card.is_rotated = True

# Player actions
player.cards = [card]
player.remove_card(card)  # Remove card
player.check_status()  # Update status
```

Notes

- The Board class uses a coordinate system with predefined start and finish cards.

- Finish cards are hidden until revealed, tracked by show_finish.
- The Card class supports path cards with matrices and rotation states.
- The Lobby class ensures a maximum of 3 players by default.
- The Player class enforces strict type checking for properties and manages equipment-based status.