Snake Game Project Documentation

1. Introduction

This document outlines the design, mechanics, and structure of the Snake Game project. It's a 2-player (or Player vs. AI, AI vs. AI) version of the classic Snake game, built using Python and the Pygame library. The game features multiple AI bot strategies, a tournament mode, various game elements like traps, and configurable game settings.

2. Core Game Mechanics

The game revolves around controlling a snake to eat food, grow longer, and achieve a higher score than the opponent, all while avoiding hazards.

- **Objective:**
    - Grow your snake by eating food.
    - Score points for each food item consumed.
    - Outlive your opponent or have a higher score when the round ends.
    - Win the majority of rounds in a tournament.
- **Snakes:**
    - **Movement:** Snakes move at a constant speed on a grid. Players/bots can change the snake's direction (Up, Down, Left, Right), but cannot immediately reverse into their own body.
    - **Growth:** Eating food increases the snake's length by a configured number of segments (`growth_per_food`).
    - **Collision:**
        - **Self-Collision:** If a snake's head collides with its own body, it dies (or enters an "advantage time" for the opponent).
        - **Wall Collision:** Colliding with the game boundaries results in death.
        - **Opponent Collision:**
            - **Head-to-Head:** Both snakes may receive penalties (lose segments, score reduction) and a temporary shield. The outcome can depend on relative scores.
            - **Head-to-Body:** The snake whose head hits an opponent's body receives penalties and a shield.
- **Food:**
    - Appears at random unoccupied locations on the grid.
    - When eaten, it disappears, increases the consumer's score and length, and a new food item may spawn elsewhere.
- **Traps:**
    - Appear at random unoccupied locations (avoiding food and snakes initially).
    - If a snake's head hits a trap:
        - The snake loses score points (`trap_penalty`).

        - The snake loses segments (`trap_segment_penalty`).
        - The snake gains a temporary shield.
        - The trap disappears.
- **Walls:**
    - The game area is enclosed by walls. Hitting a wall means the snake dies and is out for the current round.
- **Scoring:**
    - Primarily by eating food (+1 point per food, or as configured).
    - Penalties for hitting traps or colliding with opponents reduce the score.
    - Minimum snake length is enforced.
- **Rounds & Timers:**
    - A game (tournament match) consists of multiple rounds (`max_rounds`).
    - Each round has a time limit (`round_time`).
    - A round ends if:
        - Time runs out.
        - Both snakes die.
        - All food is eaten (if this is a defined win condition, though currently it ends the round).
    - The winner of a round is the snake that is alive at the end, or if both are alive, the one with the higher score. If scores are equal, it's a draw for the round.
- **Tournament Mode:**
    - The game is structured as a tournament between two agents (bots or a player).
    - Wins, losses, and scores are tracked across rounds.
    - The agent winning the most rounds wins the tournament. Tie-breaking rules (like total score or extra rounds) can apply.
    - **Advantage Time:** If one snake dies due to self-collision or wall collision, the other snake gets a period of "advantage time" (`advantage_time`). If the surviving snake also dies during this period, it may affect the round outcome. If it survives, it solidifies its win for that scenario.
    - **Early Victory:** A significant point difference (`early_victory_diff`) after a minimum number of rounds (`min_rounds_for_early_victory`) can result in an early tournament win.
- **Shield Mechanic:**
    - After certain collisions (head-to-head, head-to-body with opponent, or hitting a trap), a snake receives a temporary shield (`shield_duration`).
    - While shielded, the snake is typically immune to further collision penalties with other snakes.
    - Visually indicated by a flashing effect.

3. Project File Structure

The project is organized into several Python files:

- **`game_settings.py`:** Defines all core game entities (Snake, Food, Trap), game constants (screen dimensions, grid size, colors), the `GameConfig` dataclass for game rules, the `GameState` enum, and utility functions like `get_distance` and `is_safe`.
- **`bot.py`:** Contains the base `Bot` class and specific AI implementations (`RandomBot`, `GreedyBot`, `StrategicBot`, `CustomBot`, `UserBot`). This is where agent decision-making logic resides.
- **`main.py` (or equivalent, e.g., `snake_game.py`):** The main script that initializes Pygame, runs the game loop, manages game states, handles user input, renders graphics, and integrates the bots and tournament logic. It contains the `SnakeGame` class.
- **`tournament.py` (inferred):** This file (not provided but used by `SnakeGame`) is responsible for managing the logic of a tournament: tracking round results, scores, wins, determining the overall winner, and saving tournament statistics.

4. Key Classes and Their Logic

4.1. From `game_settings.py`

- **`Direction` Class**
  - **Purpose:** Provides named constants for movement vectors (tuples like `(dx, dy)`) and a utility to find the opposite direction. Enhances code readability.
  - **Key Attributes:** `RIGHT = (1, 0)`, `LEFT = (-1, 0)`, `UP = (0, -1)`, `DOWN = (0, 1)`.
  - **Key Methods:** `opposite(direction)`: Returns the inverse of a given direction tuple.
- **`GameState` Enum**
  - **Purpose:** Defines distinct states the game can be in, controlling the main game loop's behavior.
  - **Values:** `START`, `PLAYING`, `GAME_OVER`, `DRAW`, `ROUND_OVER`.
- **`GameConfig` Dataclass**
  - **Purpose:** A centralized container for all configurable game parameters, making it easy to tweak game rules and behavior.
  - **Key Attributes:** `tournament_mode`, `max_rounds`, `round_time`, `trap_count`, `trap_penalty`, `shield_duration`, `initial_food`, `growth_per_food`, `min_snake_length`, `early_victory_diff`, etc.
- **`GameObject` Class**
  - **Purpose:** An abstract base class for any object in the game that needs to be drawn on the screen.
  - **Key Methods:** `draw(surface)`: Abstract method to be implemented by subclasses for rendering.
- **`Snake` Class (inherits `GameObject`)**
  - **Purpose:** Represents a snake, managing its segments, movement, state, score, and interactions.
  - **Key Attributes:**

        - `segments`: A `deque` of `[x,y]` coordinates representing the snake's body, head at index 0.
        - `direction`: Current actual direction of movement (a tuple like `(1,0)`).
        - `next_direction`: Buffered direction for the next move tick.
        - `score`: The snake's current score.
        - `alive`: Boolean indicating if the snake is currently alive.
        - `shield_timer`: Countdown for how long the shield remains active.
        - `agent_id`: Name of the bot/player controlling this snake.
        - `config`: An instance of `GameConfig`.
    - **Key Methods:**
        - `reset(start_x, start_y)`: Initializes/resets the snake to a starting state.
        - `update(dt)`: Handles movement logic per frame, including moving segments, growing, and checking for self-collision or wall collision. `dt` is delta time.
        - `change_direction(new_dir)`: Sets `next_direction` if `new_dir` isn't opposite to current `direction`.
        - `check_collision_with_other(other_snake)`: Manages logic for head-to-head and head-to-body collisions with another snake, applying penalties and shields.
        - `get_head_position()`: Returns a copy of the head's `[x,y]` coordinates.
        - `draw(surface)`: Renders the snake (head with eyes, body segments, shield effect).
- **`Food` Class (inherits `GameObject`)**
    - **Purpose:** Manages food items in the game.
    - **Key Attributes:** `positions`: A list of `(x,y)` tuples for all active food items.
    - **Key Methods:**
        - `spawn_multiple(num_foods, snake_segments)`: Spawns a specified number of food items in valid locations.
        - `check_collision(head_position)`: Checks if a snake's head has collided with any food; if so, removes the food.
        - `draw(surface)`: Renders all food items.
- **`Trap` Class (inherits `GameObject`)**
    - **Purpose:** Manages traps in the game.
    - **Key Attributes:** `positions`: A list of `(x,y)` tuples for all active traps. `config`: An instance of `GameConfig`.
    - **Key Methods:**
        - `spawn_multiple(num_traps, snake_segments, food_positions)`: Spawns traps in valid locations.
        - `check_collision(snake)`: Checks if a given snake has collided with a trap; if so, applies penalties and shield to the snake and removes the trap.
        - `draw(surface)`: Renders all traps.

4.2. From `bot.py`

- **`Bot` Class (Base)**
  - **Purpose:** An abstract base class defining the interface for all AI agents.
  - **Key Attributes:** `name` (string identifier), `config` (dictionary for bot-specific weights/parameters).
  - **Key Methods:** `decide_move(snake, food, opponent)`: Abstract method. Subclasses must implement this to return a direction tuple `(dx, dy)` representing the chosen move.
- **`RandomBot` Class (inherits `Bot`)**
  - **Logic:** Chooses a random valid direction (not opposite to current movement).
- **`GreedyBot` Class (inherits `Bot`)**
  - **Logic:** Primarily aims for the nearest food. Includes basic safety checks (avoiding walls, self, opponent) and a simple mobility score to avoid getting trapped.
- **`StrategicBot` Class (inherits `Bot`)**
  - **Logic:** A more advanced bot that considers:
    - Safer food choices (e.g., food further from the opponent).
    - Predicting the opponent's next few moves to avoid collisions or contest areas.
    - Mobility and available space.
    - Advanced danger detection.

4.3. From `main.py` (or equivalent)

- **`SnakeGame` Class**
  - **Purpose:** The main class that initializes Pygame, manages the game window, orchestrates the game loop, handles events, updates game states, and renders all visual elements.
  - **Key Attributes:**
    - `screen`: The Pygame display surface.
    - `clock`: Pygame clock for managing frame rate.
    - `game_state`: Current state of the game (instance of `GameState`).
    - `config`: An instance of `GameConfig`.
    - `snake1`, `snake2`: Instances of the `Snake` class.
    - `food`: Instance of the `Food` class.
    - `traps`: Instance of the `Trap` class.
    - `bot1`, `bot2`: Instances of `Bot` subclasses.
    - `tournament`: Instance of the `Tournament` class.
    - `round_start_time`, `snake1_advantage_time`, etc.: Timers for round and advantage logic.
  - **Key Methods:**
    - `run()`: Contains the main game loop.
    - `handle_events()`: Processes Pygame events (keyboard input, quit).

        - `update()`: Updates the game logic each frame based on the current `game_state`. This includes getting moves from bots, updating snakes, checking collisions, and managing round timers/conditions.
        - `draw()`: Renders the current game scene based on `game_state` (e.g., start screen, playing field, round over screen).
        - `reset_round(swap_positions)`: Initializes snakes, food, and traps for a new round.
        - `check_collisions()`: Centralized method to check food, trap, and snake-on-snake collisions.
        - `handle_round_end()`: Processes the end of a round, determines winner, records results with the `Tournament` object, and transitions state.
        - `start_new_tournament()`: Resets tournament and starts a new game.
        - `start_next_round()`: Sets up for the subsequent round.
        - `draw_playing()`, `draw_scores()`, `draw_start_screen()`, `draw_round_over()`, `draw_tournament_end()`: Specific rendering functions.

4.4. From `tournament.py` (Inferred based on usage)

- **`Tournament` Class**
    - **Purpose:** Manages the overarching tournament structure, tracking scores, wins, and determining the final victor over multiple rounds.
    - **Key Attributes (inferred):** `config` (GameConfig instance), `results` (list of round data), `snake1_wins`, `snake2_wins`, `current_round`, `snake1_name`, `snake2_name`, `draw_rounds`, `crashed_rounds`, `total_snake1_apples`, `total_snake2_apples`.
    - **Key Methods (inferred):**
        - `__init__(config)`: Initializes the tournament.
        - `record_round(winner, snake1_score, snake2_score, ...)`: Records the outcome and statistics of a completed round.
        - `is_tournament_over()`: Checks if the conditions for ending the tournament have been met (e.g., max rounds played, one player has enough wins).
        - `get_winner()`: Returns the name of the tournament winner, or None for a draw.
        - `save_to_csv()`: Saves the detailed tournament results to a CSV file.

5. Key Standalone Functions (from `game_settings.py`)

- **`get_distance(pos1: Tuple[int, int], pos2: Tuple[int, int]) -> float`**
    - `Calculates the Euclidean distance between two points (x,y) on the grid.`
    - `Used by bots for heuristics (e.g., distance to food, opponent).`

- **`generate_spawn_positions() -> Tuple[Tuple[int, int], Tuple[int, int], List[Tuple[int, int]]]`**
    - `Generates random starting positions for the two snakes, ensuring they are a minimum distance apart.`
    - `Also generates initial positions for a set number of food items, avoiding snake spawn points.`
    - `Returns: (snake1_spawn, snake2_spawn, food_positions_list).`
- **`is_safe(snake: Snake, new_head_pos: List[int], other_snake: Optional[Snake] = None) -> bool`**
    - `Checks if a proposed new_head_pos for a snake is safe to move into.`
    - `Considers:`
        - `Wall collisions (boundaries of GRID_WIDTH, GRID_HEIGHT).`
        - `Self-collision (running into its own body, excluding the neck).`
        - `Collision with other_snake's body segments.`
    - `Does NOT inherently check for traps; trap collision is handled separately after a move is made.`

6. Game Flow

1. **Game Start:**
    a. The `SnakeGame` is initialized.
    b. The `START` screen is displayed: "Snake Tournament," "Press SPACE to begin."
2. **Tournament Initialization:**
    a. Player presses SPACE.
    b. `start_new_tournament()` is called:
        i. A new `Tournament` object is created.
        ii. `game_state` transitions to `PLAYING`.
        iii. `current_round` is set to 1.
        iv. `reset_round()` is called.
3. **Round Start (`reset_round`):**
    a. Snake spawn positions and initial food layout are generated (positions swapped for subsequent rounds if `swap_positions` is true).
    b. `Snake` objects for `snake1` and `snake2` are created/reset with their respective bot names as `agent_id`.
    c. `Food` object is initialized with the generated layout.
    d. `Trap` objects are spawned.
    e. Round timers are initialized.
4. **Gameplay Loop (`SnakeGame.update()` when `game_state == GameState.PLAYING`):**
    a. **Bot Decisions:** `bot1.decide_move()` and `bot2.decide_move()` are called to get the next intended moves.

    b. **Snake Updates:** `snake.change_direction()` sets the chosen move, then `snake.update(dt)`:
        i. Actual direction is updated.
        ii. Snake head moves one grid cell.
        iii. Body segments follow.
        iv. Growth is handled if food was recently eaten.
        v. Self-collision and wall collision checks occur, potentially setting `snake.alive = False`.
    c. **Advantage Time:** If a snake dies by self-collision/wall, an advantage timer starts for the opponent. If the opponent also dies within this window, it might be considered a mutual destruction or affect scoring.
    d. **Collision Checks (`check_collisions()`):**
        i. Food collision: If head on food, snake grows, score increases, food respawns.
        ii. Trap collision: If head on trap, snake penalized (score, length), gains shield, trap removed.
        iii. Snake-on-snake collision: Penalties, shields applied as per `snake.check_collision_with_other()`.
    e. **Round End Conditions Checked (`check_round_end()`):** Time up, both snakes dead, or no food left.
    f. **Drawing (`SnakeGame.draw()`):** The game board, snakes, food, traps, and HUD (scores, time, round info) are rendered.

5. **Round End (`handle_round_end()`):**
    a. The winner of the round (or draw) is determined based on aliveness and scores.
    b. `tournament.record_round()` is called with all relevant statistics.
    c. If `tournament.is_tournament_over()` is true:
        i. `final_winner` is determined via `tournament.get_winner()`.
        ii. `tournament.save_to_csv()` is called.
        iii. `show_final_results()` prints to console.
        iv. `game_state` becomes `GAME_OVER`.
    d. Else (tournament continues):
        i. `game_state` becomes `ROUND_OVER`.
    e. The `ROUND_OVER` screen is displayed, showing round winner and tournament progress. Press SPACE.

6. **Next Round / Tournament End:**
    a. If `ROUND_OVER`: Pressing SPACE calls `start_next_round()`, which calls `reset_round()` (often with swapped positions) and sets `game_state` back to `PLAYING`.
    b. If `GAME_OVER`: The `TOURNAMENT_END` screen shows the final winner and score. Pressing SPACE calls `quit_game()`.

7. How to Run the Game

1. **Prerequisites:**
    a. Ensure Python 3 is installed.
    b. Install the Pygame library: `pip install pygame`
2. **Execution:**
    a. Download or clone all project files (`game_settings.py`, `bot.py`, `main.py`/`snake_game.py`, and any other dependencies like `tournament.py` if it's separate).
    b. Open a terminal or command prompt.
    c. Navigate to the directory where you saved the files.
    d. Run the main game script: `python main.py` (replace `main.py` with the actual filename of the script containing the `SnakeGame` class and the `if __name__ == "__main__":` block).

8. Customization and Extension

- **Adding New Bots:**
    - Open `bot.py`.
    - Create a new class that inherits from the base `Bot` class.

```python
class MyNewBot(Bot):
    def __init__(self):
        super().__init__()
        self.name = "MyNewBotName"
        # Optionally, define bot-specific config/weights
        # self.config['my_param'] = value


    def decide_move(self, snake: Snake, food: Food, opponent: Optional[Snake] = None) -> Tuple[int, int]:
        # Implement your custom decision-making logic here
        # Access snake.get_head_position(), food.positions, opponent details, etc.
        # Use Direction.UP, Direction.DOWN, etc. or (dx,dy) tuples for moves
        # Example: return Direction.UP
        chosen_move = (0, -1) # Placeholder for UP
        # ... your logic ...
        return chosen_move
```

-
    - Open `main.py` (or your main game script).
    - Import your new bot: `from bot import MyNewBot` (if not already importing all with `*`).

  - In the `SnakeGame.__init__` method, instantiate your bot for `self.bot1` or `self.bot2`:

```
# self.bot1 = StrategicBot()
self.bot1 = MyNewBot() # Use your new bot
# self.bot2 = GreedyBot()
```

- **Adjusting Game Rules:**
  - Most game parameters can be modified by changing the default values in the `GameConfig` dataclass definition within `game_settings.py`. For example, to make rounds longer, change `round_time`. To have more traps, change `trap_count`.
- **Modifying Bot Behavior:**
  - The `config` dictionary within each `Bot` subclass (e.g., `GreedyBot`, `StrategicBot`) can be used to tune their behavior by adjusting weights like `food_weight`, `danger_weight`, etc., without changing their core logic.

Snake Game Project Documentation

1. Introduction

This document outlines the design, mechanics, and structure of the Snake Game project. It's a 2-player (or Player vs. AI, AI vs. AI) version of the classic Snake game, built using Python and the Pygame library. The game features multiple AI bot strategies, a tournament mode, various game elements like traps, and configurable game settings.

2. Core Game Mechanics

The game revolves around controlling a snake to eat food, grow longer, and achieve a higher score than the opponent, all while avoiding hazards.

- **Objective:**
  - Grow your snake by eating food.
  - Score points for each food item consumed.
  - Outlive your opponent or have a higher score when the round ends.
  - Win the majority of rounds in a tournament.
- **Snakes:**
  - **Movement:** Snakes move at a constant speed on a grid. Players/bots can change the snake's direction (Up, Down, Left, Right), but cannot immediately reverse into their own body.
  - **Growth:** Eating food increases the snake's length by a configured number of segments (`growth_per_food`).

  - **Collision:**
    - **Self-Collision:** If a snake's head collides with its own body, it dies (or enters an "advantage time" for the opponent).
    - **Wall Collision:** Colliding with the game boundaries results in death.
    - **Opponent Collision:**
      - **Head-to-Head:** Both snakes may receive penalties (lose segments, score reduction) and a temporary shield. The outcome can depend on relative scores.
      - **Head-to-Body:** The snake whose head hits an opponent's body receives penalties and a shield.
- **Food:**
  - Appears at random unoccupied locations on the grid.
  - When eaten, it disappears, increases the consumer's score and length, and a new food item may spawn elsewhere.
- **Traps:**
  - Appear at random unoccupied locations (avoiding food and snakes initially).
  - If a snake's head hits a trap:
    - The snake loses score points (`trap_penalty`).
    - The snake loses segments (`trap_segment_penalty`).
    - The snake gains a temporary shield.
    - The trap disappears.
- **Walls:**
  - The game area is enclosed by walls. Hitting a wall means the snake dies and is out for the current round.
- **Scoring:**
  - Primarily by eating food (+1 point per food, or as configured).
  - Penalties for hitting traps or colliding with opponents reduce the score.
  - Minimum snake length is enforced.
- **Rounds & Timers:**
  - A game (tournament match) consists of multiple rounds (`max_rounds`).
  - Each round has a time limit (`round_time`).
  - A round ends if:
    - Time runs out.
    - Both snakes die.
    - All food is eaten (if this is a defined win condition, though currently it ends the round).
  - The winner of a round is the snake that is alive at the end, or if both are alive, the one with the higher score. If scores are equal, it's a draw for the round.
- **Tournament Mode:**
  - The game is structured as a tournament between two agents (bots or a player).
  - Wins, losses, and scores are tracked across rounds.
  - The agent winning the most rounds wins the tournament. Tie-breaking rules (like total score or extra rounds) can apply.

    - **Advantage Time:** If one snake dies due to self-collision or wall collision, the other snake gets a period of "advantage time" (`advantage_time`). If the surviving snake also dies during this period, it may affect the round outcome. If it survives, it solidifies its win for that scenario.
    - **Early Victory:** A significant point difference (`early_victory_diff`) after a minimum number of rounds (`min_rounds_for_early_victory`) can result in an early tournament win.
- **Shield Mechanic:**
    - After certain collisions (head-to-head, head-to-body with opponent, or hitting a trap), a snake receives a temporary shield (`shield_duration`).
    - While shielded, the snake is typically immune to further collision penalties with other snakes.
    - Visually indicated by a flashing effect.

3. Project File Structure

The project is organized into several Python files:

- **`game_settings.py`:** Defines all core game entities (Snake, Food, Trap), game constants (screen dimensions, grid size, colors), the `GameConfig` dataclass for game rules, the `GameState` enum, and utility functions like `get_distance` and `is_safe`.
- **`bot.py`:** Contains the base `Bot` class and specific AI implementations (`RandomBot`, `GreedyBot`, `StrategicBot`, `CustomBot`, `UserBot`). This is where agent decision-making logic resides.
- **`main.py` (or equivalent, e.g., `snake_game.py`):** The main script that initializes Pygame, runs the game loop, manages game states, handles user input, renders graphics, and integrates the bots and tournament logic. It contains the `SnakeGame` class.
- **`tournament.py` (inferred):** This file (not provided but used by `SnakeGame`) is responsible for managing the logic of a tournament: tracking round results, scores, wins, determining the overall winner, and saving tournament statistics.

4. Key Classes and Their Logic

4.1. From `game_settings.py`

- **`Direction` Class**
    - **Purpose:** Provides named constants for movement vectors (tuples like `(dx, dy)`) and a utility to find the opposite direction. Enhances code readability.
    - **Key Attributes:** `RIGHT = (1, 0)`, `LEFT = (-1, 0)`, `UP = (0, -1)`, `DOWN = (0, 1)`.
    - **Key Methods:** `opposite(direction)`: Returns the inverse of a given direction tuple.
- **`GameState` Enum**

    - **Purpose:** Defines distinct states the game can be in, controlling the main game loop's behavior.
    - **Values:** `START`, `PLAYING`, `GAME_OVER`, `DRAW`, `ROUND_OVER`.
- `GameConfig` **Dataclass**
    - **Purpose:** A centralized container for all configurable game parameters, making it easy to tweak game rules and behavior.
    - **Key Attributes:** `tournament_mode`, `max_rounds`, `round_time`, `trap_count`, `trap_penalty`, `shield_duration`, `initial_food`, `growth_per_food`, `min_snake_length`, `early_victory_diff`, etc.
- `GameObject` **Class**
    - **Purpose:** An abstract base class for any object in the game that needs to be drawn on the screen.
    - **Key Methods:** `draw(surface)`: Abstract method to be implemented by subclasses for rendering.
- `Snake` **Class (inherits** `GameObject`**)**
    - **Purpose:** Represents a snake, managing its segments, movement, state, score, and interactions.
    - **Key Attributes:**
        - `segments`: A `deque` of `[x,y]` coordinates representing the snake's body, head at index 0.
        - `direction`: Current actual direction of movement (a tuple like `(1,0)`).
        - `next_direction`: Buffered direction for the next move tick.
        - `score`: The snake's current score.
        - `alive`: Boolean indicating if the snake is currently alive.
        - `shield_timer`: Countdown for how long the shield remains active.
        - `agent_id`: Name of the bot/player controlling this snake.
        - `config`: An instance of `GameConfig`.
    - **Key Methods:**
        - `reset(start_x, start_y)`: Initializes/resets the snake to a starting state.
        - `update(dt)`: Handles movement logic per frame, including moving segments, growing, and checking for self-collision or wall collision. `dt` is delta time.
        - `change_direction(new_dir)`: Sets `next_direction` if `new_dir` isn't opposite to current `direction`.
        - `check_collision_with_other(other_snake)`: Manages logic for head-to-head and head-to-body collisions with another snake, applying penalties and shields.
        - `get_head_position()`: Returns a copy of the head's `[x,y]` coordinates.
        - `draw(surface)`: Renders the snake (head with eyes, body segments, shield effect).
- `Food` **Class (inherits** `GameObject`**)**
    - **Purpose:** Manages food items in the game.

    - **Key Attributes:** `positions`: A list of `(x,y)` tuples for all active food items.
    - **Key Methods:**
        - `spawn_multiple(num_foods, snake_segments)`: Spawns a specified number of food items in valid locations.
        - `check_collision(head_position)`: Checks if a snake's head has collided with any food; if so, removes the food.
        - `draw(surface)`: Renders all food items.
- **`Trap` Class (inherits `GameObject`)**
    - **Purpose:** Manages traps in the game.
    - **Key Attributes:** `positions`: A list of `(x,y)` tuples for all active traps. `config`: An instance of `GameConfig`.
    - **Key Methods:**
        - `spawn_multiple(num_traps, snake_segments, food_positions)`: Spawns traps in valid locations.
        - `get_positions()`:It will return all positions of traps in the game board.
        - `check_collision(snake)`: Checks if a given snake has collided with a trap; if so, applies penalties and shield to the snake and removes the trap.
        - `draw(surface)`: Renders all traps.

4.2. From `bot.py`

- **`Bot` Class (Base)**
    - **Purpose:** An abstract base class defining the interface for all AI agents.
    - **Key Attributes:** `name` (string identifier), `config` (dictionary for bot-specific weights/parameters).
    - **Key Methods:** `decide_move(snake, food, opponent)`: Abstract method. Subclasses must implement this to return a direction tuple `(dx, dy)` representing the chosen move.
- **`RandomBot` Class (inherits `Bot`)**
    - **Logic:** Chooses a random valid direction (not opposite to current movement).
- **`GreedyBot` Class (inherits `Bot`)**
    - **Logic:** Primarily aims for the nearest food. Includes basic safety checks (avoiding walls, self, opponent) and a simple mobility score to avoid getting trapped.
- **`StrategicBot` Class (inherits `Bot`)**
    - **Logic:** A more advanced bot that considers:
        - Safer food choices (e.g., food further from the opponent).
        - Predicting the opponent's next few moves to avoid collisions or contest areas.
        - Mobility and available space.
        - Advanced danger detection.

4.3. From `main.py` (or equivalent)

- **`SnakeGame` Class**
  - **Purpose:** The main class that initializes Pygame, manages the game window, orchestrates the game loop, handles events, updates game states, and renders all visual elements.
  - **Key Attributes:**
    - `screen`: The Pygame display surface.
    - `clock`: Pygame clock for managing frame rate.
    - `game_state`: Current state of the game (instance of `GameState`).
    - `config`: An instance of `GameConfig`.
    - `snake1`, `snake2`: Instances of the `Snake` class.
    - `food`: Instance of the `Food` class.
    - `traps`: Instance of the `Trap` class.
    - `bot1`, `bot2`: Instances of `Bot` subclasses.
    - `tournament`: Instance of the `Tournament` class.
    - `round_start_time`, `snake1_advantage_time`, etc.: Timers for round and advantage logic.
  - **Key Methods:**
    - `run()`: Contains the main game loop.
    - `handle_events()`: Processes Pygame events (keyboard input, quit).
    - `update()`: Updates the game logic each frame based on the current `game_state`. This includes getting moves from bots, updating snakes, checking collisions, and managing round timers/conditions.
    - `draw()`: Renders the current game scene based on `game_state` (e.g., start screen, playing field, round over screen).
    - `reset_round(swap_positions)`: Initializes snakes, food, and traps for a new round.
    - `check_collisions()`: Centralized method to check food, trap, and snake-on-snake collisions.
    - `handle_round_end()`: Processes the end of a round, determines winner, records results with the `Tournament` object, and transitions state.
    - `start_new_tournament()`: Resets tournament and starts a new game.
    - `start_next_round()`: Sets up for the subsequent round.
    - `draw_playing()`, `draw_scores()`, `draw_start_screen()`, `draw_round_over()`, `draw_tournament_end()`: Specific rendering functions.

4.4. From `tournament.py` (Inferred based on usage)

- **`Tournament` Class**
  - **Purpose:** Manages the overarching tournament structure, tracking scores, wins, and determining the final victor over multiple rounds.

    - **Key Attributes (inferred):** `config` (GameConfig instance), `results` (list of round data), `snake1_wins`, `snake2_wins`, `current_round`, `snake1_name`, `snake2_name`, `draw_rounds`, `crashed_rounds`, `total_snake1_apples`, `total_snake2_apples`.
    - **Key Methods (inferred):**
        - `__init__(config)`: Initializes the tournament.
        - `record_round(winner, snake1_score, snake2_score, ...)`: Records the outcome and statistics of a completed round.
        - `is_tournament_over()`: Checks if the conditions for ending the tournament have been met (e.g., max rounds played, one player has enough wins).
        - `get_winner()`: Returns the name of the tournament winner, or None for a draw.
        - `save_to_csv()`: Saves the detailed tournament results to a CSV file.

5. Key Standalone Functions (from `game_settings.py`)

- **`get_distance(pos1: Tuple[int, int], pos2: Tuple[int, int]) -> float`**
    - `Calculates the Euclidean distance between two points (x,y) on the grid.`
    - `Used by bots for heuristics (e.g., distance to food, opponent).`
- **`generate_spawn_positions() -> Tuple[Tuple[int, int], Tuple[int, int], List[Tuple[int, int]]]`**
    - `Generates random starting positions for the two snakes, ensuring they are a minimum distance apart.`
    - `Also generates initial positions for a set number of food items, avoiding snake spawn points.`
    - `Returns: (snake1_spawn, snake2_spawn, food_positions_list).`
- **`is_safe(snake: Snake, new_head_pos: List[int], other_snake: Optional[Snake] = None) -> bool`**
    - `Checks if a proposed new_head_pos for a snake is safe to move into.`
    - `Considers:`
        - `Wall collisions (boundaries of GRID_WIDTH, GRID_HEIGHT).`
        - `Self-collision (running into its own body, excluding the neck).`
        - `Collision with other_snake's body segments.`
    - `Does NOT inherently check for traps; trap collision is handled separately after a move is made.`

6. Game Flow

1. **Game Start:**

a. The `SnakeGame` is initialized.
b. The `START` screen is displayed: "Snake Tournament," "Press SPACE to begin."

2. **Tournament Initialization:**
   a. Player presses SPACE.
   b. `start_new_tournament()` is called:
      i. A new `Tournament` object is created.
      ii. `game_state` transitions to `PLAYING`.
      iii. `current_round` is set to 1.
      iv. `reset_round()` is called.
3. **Round Start (`reset_round`):**
   a. Snake spawn positions and initial food layout are generated (positions swapped for subsequent rounds if `swap_positions` is true).
   b. `Snake` objects for `snake1` and `snake2` are created/reset with their respective bot names as `agent_id`.
   c. `Food` object is initialized with the generated layout.
   d. `Trap` objects are spawned.
   e. Round timers are initialized.
4. **Gameplay Loop (`SnakeGame.update()` when `game_state == GameState.PLAYING`):**
   a. **Bot Decisions:** `bot1.decide_move()` and `bot2.decide_move()` are called to get the next intended moves.
   b. **Snake Updates:** `snake.change_direction()` sets the chosen move, then `snake.update(dt)`:
      i. Actual direction is updated.
      ii. Snake head moves one grid cell.
      iii. Body segments follow.
      iv. Growth is handled if food was recently eaten.
      v. Self-collision and wall collision checks occur, potentially setting `snake.alive = False`.
   c. **Advantage Time:** If a snake dies by self-collision/wall, an advantage timer starts for the opponent. If the opponent also dies within this window, it might be considered a mutual destruction or affect scoring.
   d. **Collision Checks (`check_collisions()`):**
      i. Food collision: If head on food, snake grows, score increases, food respawns.
      ii. Trap collision: If head on trap, snake penalized (score, length), gains shield, trap removed.
      iii. Snake-on-snake collision: Penalties, shields applied as per `snake.check_collision_with_other()`.
   e. **Round End Conditions Checked (`check_round_end()`):** Time up, both snakes dead, or no food left.
   f. **Drawing (`SnakeGame.draw()`):** The game board, snakes, food, traps, and HUD (scores, time, round info) are rendered.
5. **Round End (`handle_round_end()`):**

   a. The winner of the round (or draw) is determined based on aliveness and scores.
   b. `tournament.record_round()` is called with all relevant statistics.
   c. If `tournament.is_tournament_over()` is true:
      i. `final_winner` is determined via `tournament.get_winner()`.
      ii. `tournament.save_to_csv()` is called.
      iii. `show_final_results()` prints to console.
      iv. `game_state` becomes `GAME_OVER`.
   d. Else (tournament continues):
      i. `game_state` becomes `ROUND_OVER`.
   e. The `ROUND_OVER` screen is displayed, showing round winner and tournament progress. Press SPACE.

6. **Next Round / Tournament End:**
   a. If `ROUND_OVER`: Pressing SPACE calls `start_next_round()`, which calls `reset_round()` (often with swapped positions) and sets `game_state` back to `PLAYING`.
   b. If `GAME_OVER`: The `TOURNAMENT_END` screen shows the final winner and score. Pressing SPACE calls `quit_game()`.

7. How to Run the Game

1. **Prerequisites:**
   a. Ensure Python 3 is installed.
   b. Install the Pygame library: `pip install pygame`
2. **Execution:**
   a. Download or clone all project files (`game_settings.py`, `bot.py`, `main.py`/`snake_game.py`, and any other dependencies like `tournament.py` if it's separate).
   b. Open a terminal or command prompt.
   c. Navigate to the directory where you saved the files.
   d. Run the main game script: `python main.py` (replace `main.py` with the actual filename of the script containing the `SnakeGame` class and the `if __name__ == "__main__":` block).

8. Customization and Extension

- **Adding New Bots:**
  - Open `bot.py`.
  - Create a new class that inherits from the base `Bot` class.

```
● class MyNewBot(Bot):
●     def __init__(self):
●         super().__init__()
```

```
        self.name = "MyNewBotName"
        # Optionally, define bot-specific config/weights
        # self.config['my_param'] = value


    def decide_move(self, snake: Snake, food: Food, opponent: Optional[Snake] = None) -> Tuple[int, int]:
        # Implement your custom decision-making logic here
        # Access snake.get_head_position(), food.positions, opponent details, etc.
        # Use Direction.UP, Direction.DOWN, etc. or (dx,dy) tuples for moves
        # Example: return Direction.UP
        chosen_move = (0, -1) # Placeholder for UP
        # ... your logic ...
        return chosen_move
```

-
    - Open `main.py` (or your main game script).
    - Import your new bot: `from bot import MyNewBot` (if not already importing all with `*`).
    - In the `SnakeGame.__init__` method, instantiate your bot for `self.bot1` or `self.bot2`:

```
# self.bot1 = StrategicBot()
self.bot1 = MyNewBot() # Use your new bot
# self.bot2 = GreedyBot()
```

- **Adjusting Game Rules:**
    - Most game parameters can be modified by changing the default values in the `GameConfig` dataclass definition within `game_settings.py`. For example, to make rounds longer, change `round_time`. To have more traps, change `trap_count`.
- **Modifying Bot Behavior:**
    - The `config` dictionary within each `Bot` subclass (e.g., `GreedyBot`, `StrategicBot`) can be used to tune their behavior by adjusting weights like `food_weight`, `danger_weight`, etc., without changing their core logic.