Programming Technology

<--->

4^{th}-Task Abaidullah Asif ZLWD2B

<--->

Assignment_3

Task-Description:

Create a game, with we can play the light-motorcycle battle (known from the Tron movie) in a top view. Two players play against each other with two motors, where each motor leaves a light trace behind of itself on the display. The motor goes in each seconds toward the direction, that the player has set recently. The first player can use the WASD keyboard buttons, while the second one can use the cursor buttons for steering. A player loses if its motor goes to the boundary of the game level, or it goes to the light trace of the other player. Ask the name of the players before the game starts, and let them choose the color their light traces. Increase the counter of the winner by one in the database at the end of the game. If the player does not exit in the database yet, then insert a record for him. Create a menu item, which displays a highscore table of the players for the 10 best scores. Also, create a menu item which restarts the game.

Analysis:

TRON Light-Cycle Battle" is an arcade-style two-player game in which players drive motorcycles that leave trails as they move over a square board. The aim is to trap the opponent into forcing him to collide with a trail, boundary, or obstacle while avoiding any collision. Different game boards are used, each presenting special hurdles, such as central hurdles, zigzag obstacles, or visible boundaries. Players can control their motorcycles using keyboard input, while their movements are displayed on a graphical interface. Each motorcycle leaves behind a trail that becomes an obstacle for both players. Collisions with the boundary, hurdles, or trails result in a game over, declaring the winner. It also allows some customization options like player names, colors, and board selection. It implements a high-score system to track player performances over sessions. The core game controller manages the switching between the welcome screen, menu, player setup screen, and game boards. The application design is modular for easy maintainability and scalability. The UI of the application is user-friendly: showing player names, colors, and current game time. It handles game initialization, rendering boards, and result dialogs. The program is designed to provide a fast-paced, strategic experience while ensuring simplicity and aesthetics in both gameplay and interface.

Plan:

The object-oriented implementation for the "TRON Light-Cycle Battle" game is modular and consists of several interconnected classes. Each class encapsulates a specific functionality to ensure code clarity, maintainability, and scalability. The game is designed to provide a competitive and engaging experience with different game boards, each offering unique challenges and mechanics. The main features include player-controlled motorcycles, various board setups with hurdles, and a high-score system.

The implementation is divided into the following classes:

TRON

- **Purpose**: Acts as the entry point of the game.
- **Responsibilities**:
    - Initializes the GameController.
    - Starts the game by transitioning to the welcome screen.

GameController

- **Purpose**: Serves as the central controller of the game.
- **Responsibilities**:
    - Manages screen transitions between the welcome screen, menu, player setup, and game boards.
    - Handles game state control, such as starting and restarting the game.
    - Passes player details and selected game board configurations to the respective classes.
    - Updates high scores using the HighScores class.
- **Interesting Algorithms**:
    - **Game State Management**: Maintains the state of the game and switches between screens based on user actions.
    - **High Score Update**: Implements logic to update the winner's high score dynamically.

WelcomeScreen

- **Purpose**: The initial screen displayed to the user.
- **Responsibilities**:
    - Displays a welcome message and introduces the game.
    - Contains a "Start" button to navigate to the MenuScreen.

MenuScreen

- **Purpose**: Acts as a hub for players to access game features.
- **Responsibilities**:
    - Allows players to start a new game, view high scores, or quit the application.
    - Includes button handlers to perform specific actions like navigating to the PlayerSetupScreen or displaying high scores.

PlayerSetupScreen

- **Purpose**: Collects player details and game configurations.
- **Responsibilities**:
    - Takes player names, colors, and game board selection as input.
    - Passes these details to the GameController to start the game.

GameBoard and Variants

- **Purpose**: Represent the game board where players compete.
- **Responsibilities**:
    - Manage the game grid and handle player movements.
    - Detect collisions (with trails, hurdles, and boundaries) and enforce game rules.
    - Each board variant includes unique hurdle placements and collision logic.

- **Variants**:

1. **GameBoardWithCentralCrossHurdle**: A cross-shaped hurdle in the center.
2. **GameBoardWithHurdle**: A single central hurdle.
3. **GameBoardWithTwoParallelHurdles**: Two parallel hurdles in the center.
4. **GameBoardWithTwoHurdles**: Two horizontal hurdles.
5. **GameBoardWithFourCorners**: Hurdles in all four corners.
6. **GameBoardWithThreeVerticalHurdles**: Three vertical hurdles.
7. **GameBoardWithZigZagHurdles**: Zigzag-shaped hurdles across the board.
8. **GameBoardWithVisibleBoundary**: Features an enhanced boundary collision.
9. **GameBoardWithThreeSmallCircleHurdles**: Includes three circular hurdles.

- **Interesting Algorithms**:
  - **Collision Detection**:
    - Checks for collisions with trails, hurdles, or boundaries.
    - Implements unique logic for detecting hits on circular and zigzag hurdles.
  - **Drawing Mechanics**:
    - Each board variant dynamically draws its unique hurdles and boundaries.

Motorcycle

- **Purpose**: Represents player-controlled motorcycles.
- **Responsibilities**:
  - Tracks the position, direction, and trail of each motorcycle.
  - Manages player movement based on keyboard inputs.
  - Detects collisions with trails, hurdles, and boundaries.
- **Interesting Algorithms**:
  - **Trail Management**: Adds the current position of the motorcycle to its trail after every move.

- **Collision Detection**: Uses efficient checks to determine if the motorcycle hits another trail or an obstacle.

GameDrawArea

- **Purpose**: Handles graphical rendering of the game.
- **Responsibilities**:
  - Draws motorcycles, their trails, and board-specific hurdles.
  - Dynamically updates the visuals based on game state changes.

HighScores

- **Purpose**: Manages the high-score database.
- **Responsibilities**:
  - Fetches, updates, and saves high scores.
  - Ensures that scores are sorted in descending order.
- **Interesting Algorithms**:
  - **Database Interaction**: Efficiently queries the database for fetching and updating player scores.
  - **Sorting**: Uses a comparator to sort scores in real-time for display.

HighScore

- **Purpose**: Represents an individual high-score entry.
- **Responsibilities**:
  - Stores a player's name and score.
  - Provides methods to access these details.

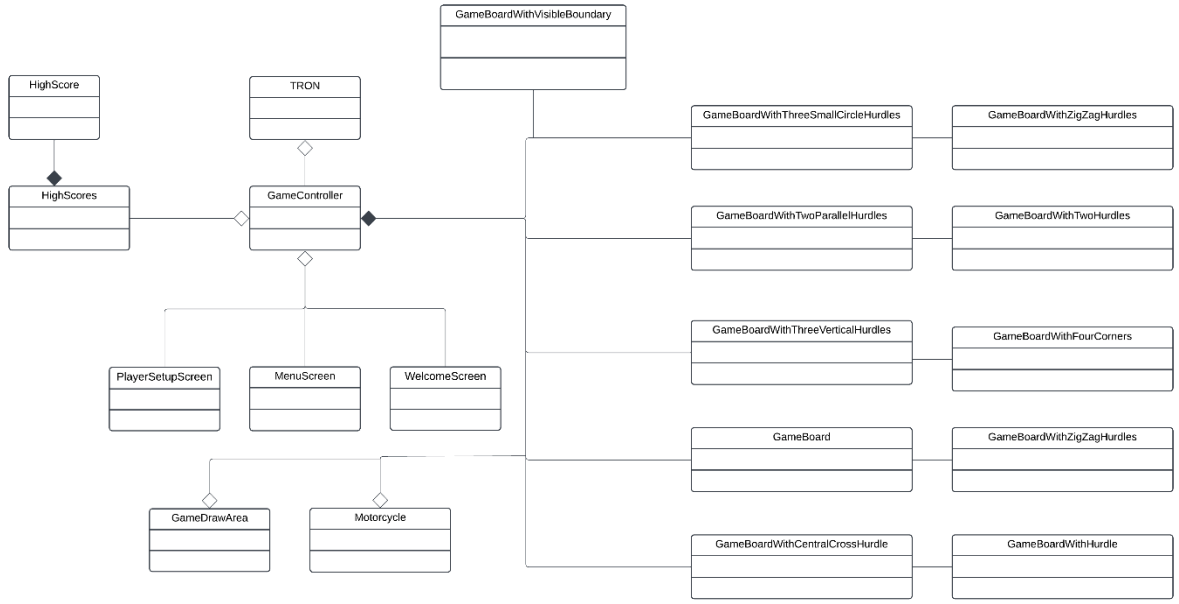
Interesting Algorithms

1. **Collision Detection in GameBoard**:
   - Detects various types of collisions:
     - Motorcycle hitting a trail (self or opponent's).

        - Motorcycle hitting a hurdle (variant-specific logic).
        - Motorcycle moving out of bounds.
    - Each detection type is optimized for the specific board variant.
2. **Trail Drawing and Management**:
    - Maintains an array of positions for each motorcycle's trail.
    - Draws the trail dynamically as the motorcycle moves.
3. **Hurdle Placement and Rendering**:
    - Each board variant defines unique hurdle positions and shapes.
    - Example: Zigzag hurdles involve non-linear placements that require custom collision logic.
4. **High Score Management**:
    - Updates player scores based on game outcomes.
    - Ensures that only the top 10 scores are stored in the database.
    - Handles ties and identical scores efficiently.
5. **Boundary Collision (GameBoardWithVisibleBoundary)**:
    - Implements a highly visible boundary with a thick stroke.
    - Detects when a motorcycle moves beyond the boundary and ends the game.

Note: You can see the documentation for function detail. (java doc file).

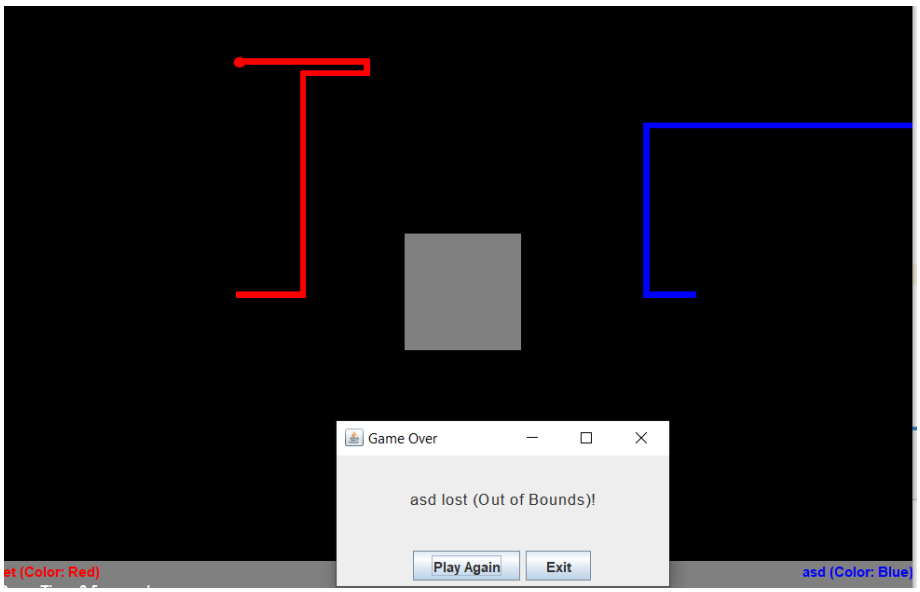
Class Diagram:



Test cases:

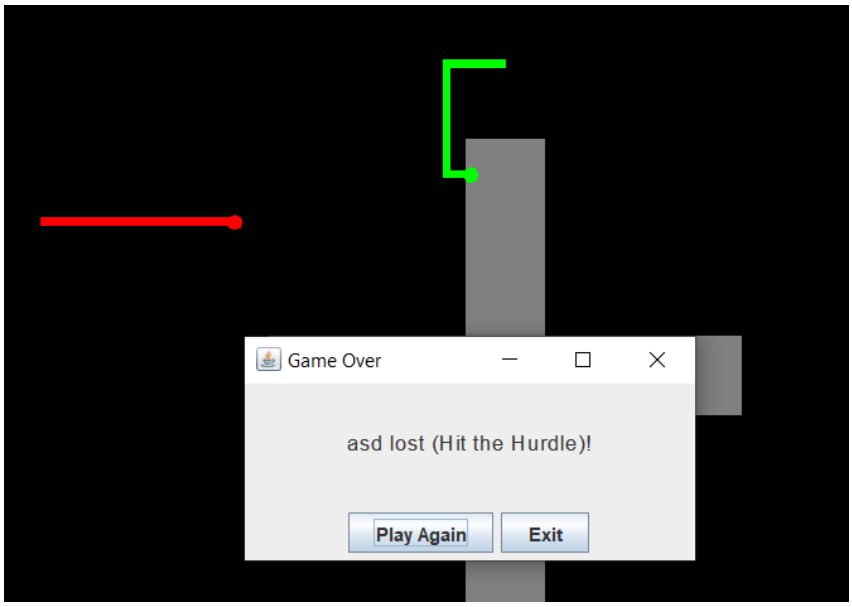
Test Case 1: Boundary Collision

- **AS A** player
- **I WANT TO** lose the game if my motorcycle goes out of bounds
- **GIVEN** I attempt to move outside the game board
- **WHEN** my motorcycle crosses the board boundary
- **THEN** the game should declare my opponent the winner.



Test Case 2: Hurdle Collision

- **AS A** player
- **I WANT TO** lose the game if I hit a hurdle
- **GIVEN** the board has hurdles
- **WHEN** my motorcycle collides with a hurdle
- **THEN** the game should end, and my opponent should be declared the winner.

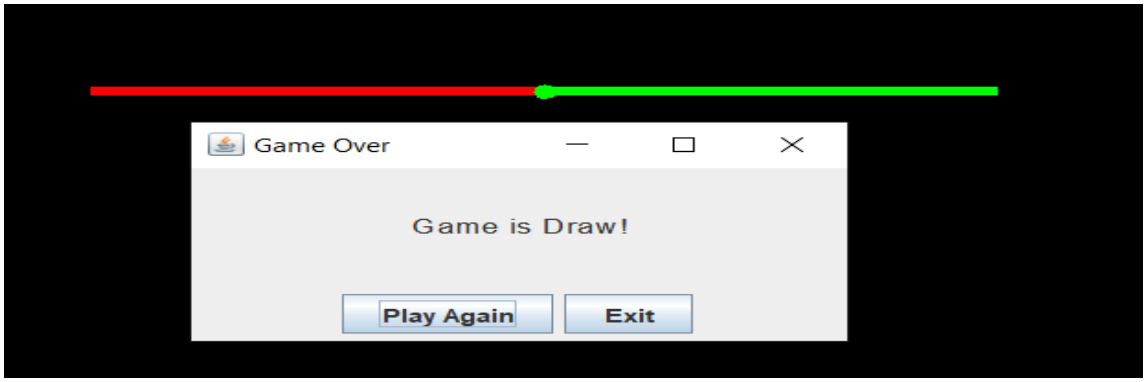


Test Case 3: Trail Collision

- **AS A** player
- **I WANT TO** lose the game if I move into my opponent's trail
- **GIVEN** my opponent has left a trail on the board
- **WHEN** I move into a cell occupied by their trail
- **THEN** the game should declare my opponent the winner

Test Case 4: Head-on Collision

- **AS A** player
- **I WANT TO** see the game end in a draw if both motorcycles collide head-on
- **GIVEN** both players move into the same cell
- **WHEN** the collision happens
- **THEN** the game should declare a draw

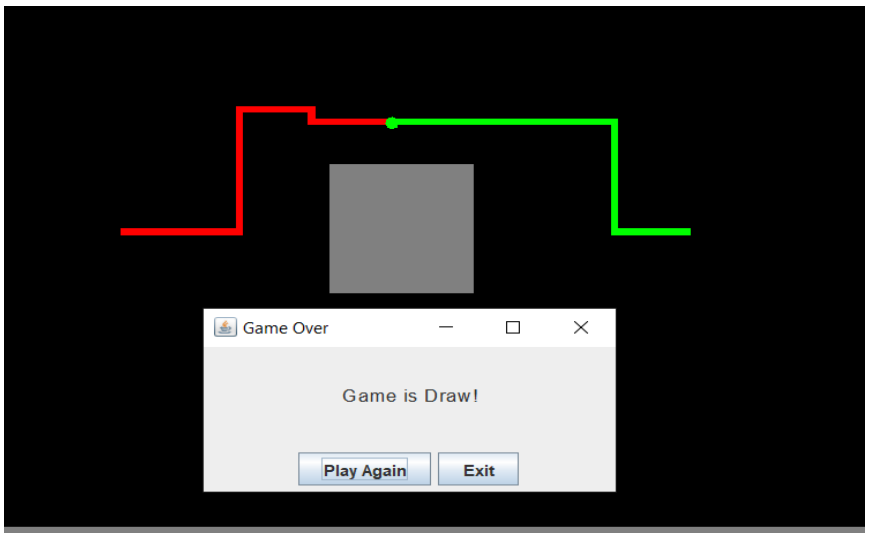


Test Case 5: Turn-based Movement

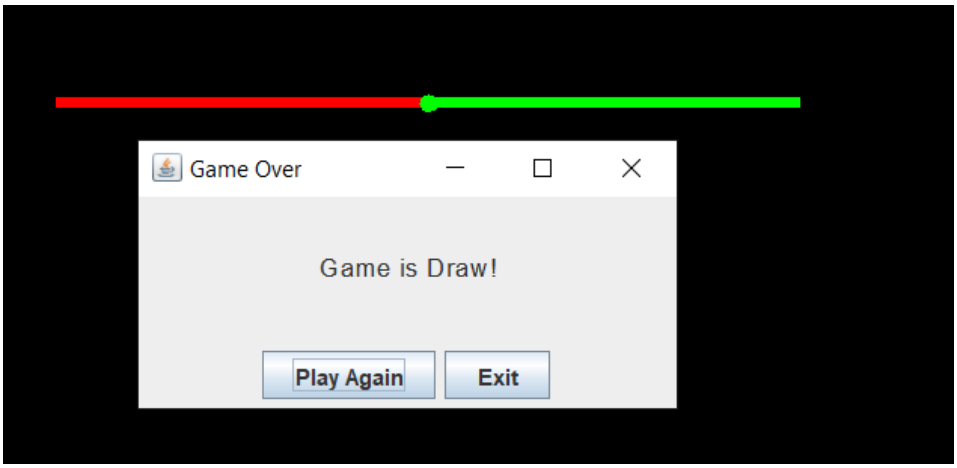
- **AS A** player
- **I WANT TO** move my motorcycle in the specified direction on my turn
- **GIVEN** it is my turn to move
- **WHEN** I press a valid key (e.g., W, A, S, D or arrow keys)
- **THEN** my motorcycle should move in the specified directio

Test Case 6: Game Over Screen

- **AS A** player
- **I WANT TO** see a game over screen when the game ends
- **GIVEN** the game has concluded due to a collision or boundary exit
- **WHEN** the game ends
- **THEN** a dialog box should display the winner or draw status



Test Case 7: Restart Game

- **AS A** player
- **I WANT TO** restart the game after it ends
- **GIVEN** the game has concluded
- **WHEN** I click the "Play Again" button on the game over screen
- **THEN** the game should restart with the same board configuration
- 

Test Case 8: Timer Display

- **AS A** player
- **I WANT TO** see the elapsed time during the game
- **GIVEN** the game is in progress
- **WHEN** time passes
- **THEN** the timer should display the correct elapsed time in seconds
- 

Test Case 9: High Scores Update

- **AS A** player
- **I WANT TO** have my high score updated when I win
- **GIVEN** I win a game
- **WHEN** the game ends
- **THEN** my score should increment in the high scores table

1. Boundary Collision Detection

AS A developer
I WANT TO ensure that motorcycles cannot move out of the game board boundaries
GIVEN the Motorcycle.isOutOfBounds method is called with the motorcycle's position
WHEN the position is outside the board dimensions (e.g., x < 0 or y >= board height)
THEN the method should return true to indicate a boundary collision

2. Hurdle Collision Detection

AS A developer
I WANT TO accurately detect when a motorcycle collides with a hurdle
GIVEN the GameBoard.checkCollisions method is called with the motorcycle's position
WHEN the motorcycle's position overlaps with a hurdle's rectangle
THEN the method should trigger game over logic and end the game

3. Trail Collision Detection

AS A developer
I WANT TO verify the detection of trail collisions during gameplay
GIVEN the GameBoard.checkCollisions method is called with the trail positions of motorcycles
WHEN a motorcycle moves into a cell occupied by another motorcycle's trail
THEN the method should trigger game over logic and declare the other player the winner

4. Head-on Collision Detection

AS A developer
I WANT TO detect when both motorcycles collide head-on
GIVEN the GameBoard.checkCollisions method is called
WHEN both motorcycles' positions are the same
THEN the method should declare a draw and end the game

5. Timer Functionality

AS A developer
I WANT TO ensure the game timer updates correctly during gameplay
GIVEN the Timer object is active and updating every tick
WHEN the ActionListener increments the game time
THEN the elapsed time displayed in the JLabel should increase accurately in seconds

6. High Scores Update

AS A developer
I WANT TO verify that the high scores are correctly updated after a game ends
GIVEN the GameController.updateHighScores method is called with the winning player's name
WHEN the method executes the SQL update query
THEN the database should increment the player's score by 1

Events:

Welcome Screen Events

- **Event**: Clicking the "Start" button
- **Handler**: StartButtonActionListener.actionPerformed()
- **Action**: Transitions to the Menu Screen by calling GameController.showMenuScreen().

Menu Screen Events

- **Event**: Clicking the "Start Game" button
- **Handler**: StartGameButtonActionListener.actionPerformed()
- **Action**: Transitions to the Player Setup Screen by calling GameController.showPlayerSetupScreen().
- **Event**: Clicking the "High Scores" button
- **Handler**: HighScoresButtonActionListener.actionPerformed()
- **Action**: Displays the high scores by calling GameController.showHighScores().
- **Event**: Clicking the "Exit" button
- **Handler**: ExitButtonActionListener.actionPerformed()
- **Action**: Exits the application.

Player Setup Screen Events

- **Event**: Clicking the "Start Game" button
- **Handler**: StartGameButtonActionListener.actionPerformed()
- **Action**: Initializes the game board with player configurations by calling GameController.startGame().

Game Board Events

- **Event**: Timer tick
- **Handler**: ActionListener.actionPerformed() (in Timer)
- **Action**: Updates motorcycle positions, increments game time, and checks for collisions by calling GameBoard.checkCollisions().
- **Event**: Key press for controlling motorcycles
- **Handler**: KeyHandler.keyPressed()

- **Action**: Updates the direction of the respective motorcycle.

Game Over Dialog Events

- **Event**: Clicking the "Play Again" button
- **Handler**: ActionListener.actionPerformed()
- **Action**: Restarts the game by calling GameController.restartGame().
- **Event**: Clicking the "Exit" button
- **Handler**: ActionListener.actionPerformed()
- **Action**: Exits the application.

High Scores Events

- **Event**: Game ends with a winner
- **Handler**: GameController.updateHighScores()
- **Action**: Updates the winner's score in the database.