Tron

Written and Implemented By

Mohammed Efaz (ZTFK53)

3rd Semester | 4th December 2023

Presented To: Sir Hoque A H M Sajedul

Faculty of Informatics

Programming Technology | IP-18fPROGTEG

Eötvös Loránd University

Contents

Task

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 second 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 colour 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 exist in the database yet, then insert a record for him. Create a menu item, which displays a high score table of the players for the 10 best scores. Also, create a menu item which restarts the game.

Analysis of the Task

The task involves two players controlling their respective characters, aiming to avoid collisions while navigating through a grid. The program is structured following the Model-View-Controller (MVC) architecture, ensuring a clear separation of concerns.

Model: The model layer consists of classes like `ModelGame`, `ModelPlayer`, `ModelLevel`, `ModelStorage`, and `ModelTimer`. These handle the game's data and logic. `ModelGame` manages the game state, including level progression and player management. `ModelPlayer` represents each player, tracking position, direction, and score. `ModelLevel` defines the size and other characteristics of each level. `ModelStorage` interfaces with the database for score management. `ModelTimer` is used for tracking time within the game.

View: The view layer includes `ViewGame`, `ViewPlayer`, and `ViewScore`. These classes are responsible for presenting the game's user interface. `ViewGame` displays the game grid, player movements, and light trails. `ViewPlayer` handles player registration, and `ViewScore` presents high score information.

Controller: `ControllerGame` acts as the controller, handling user inputs and updating the model and view accordingly. It processes key events for player movements and orchestrates the game loop for continuous state updates.

Implementation of the Task

Game Loop: The game loop in `ControllerGame`, triggered every 100 milliseconds by a `Timer`, is crucial. It updates player positions, checks for collisions, and updates the game view. This loop is central to the game's real-time dynamics.

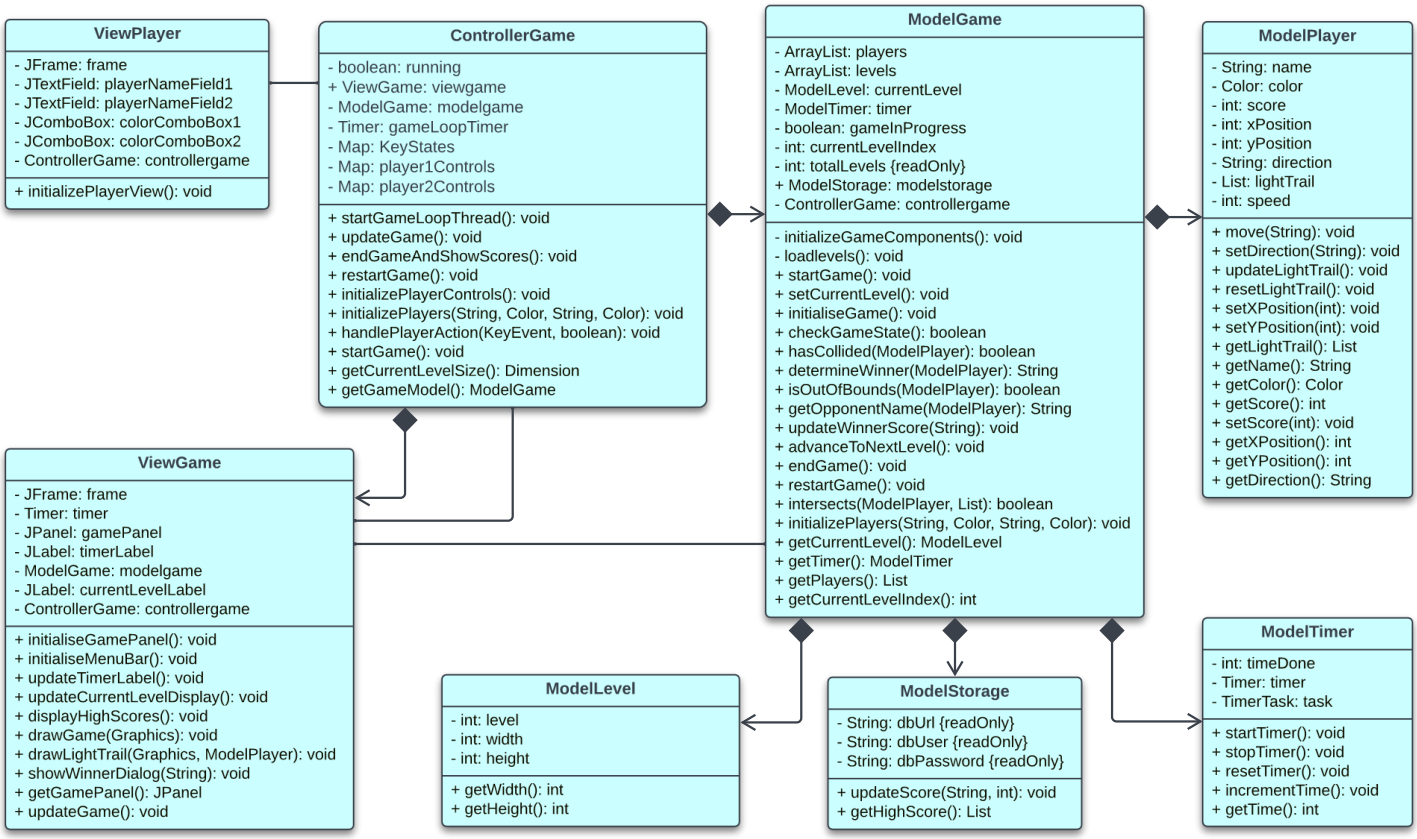
Collision Detection: An important part of the game logic is the collision detection algorithm in `ModelGame`. It checks for collisions of a player with the game boundaries, their own light trail, and the opponent's light trail. This is vital for determining game outcomes.

Level Generation: The levels in the game are dynamically sized. In `ModelGame`, the `loadLevels()` method generates levels with varying dimensions, adding an element of progression and complexity to the game.

Score Management and Database Interaction: `ModelStorage` contains algorithms for database interactions, particularly for updating player scores and retrieving high scores. The SQL queries handle the insertion of new scores and aggregation of top scores.

Light Trail Management: Each player's movement creates a light trail, managed by `ModelPlayer`. The `updateLightTrail()` method updates this trail, a key visual and functional element of the game.

UML Diagram



`ControllerGame` composes **`ViewGame`** and **`ModelGame`**: Is responsible for the instantiation, operation, and lifecycle of these classes. **`ViewGame`** provides the user interface, while **`ModelGame`** holds the game's state and logic.

`ModelGame` composes **`ModelPlayer`**, **`ModelLevel`**, **`ModelTimer`**, and **`ModelStorage`**: These classes are integral parts of the game's core functionality. **`ModelGame`** creates and manages players (**`ModelPlayer`**), levels (**`ModelLevel`**), game timing (**`ModelTimer`**), and persistent storage (**`ModelStorage`**).

`ViewGame` associates **`ControllerGame`**: **`ViewGame`** relies on **`ControllerGame`** to initiate game actions and update the display, but does not manage its lifecycle

`ViewPlayer` associates **`ControllerGame`**: These classes interface with **`ControllerGame`** to perform player registration and display scores, respectively. They do not control or own **`ControllerGame`**.

Description of the methods

`ControllerGame`

- **`ControllerGame()`**: Initialises game controller, sets up key states and player controls, starts game loop.
- **`startGameLoopThread()`**: Starts the game loop timer, triggering periodic game state updates.
- **`updateGame()`**: Updates game state, including player movements, collision detection, level advancement.
- **`endGameAndShowScores()`**: Ends game, shows final scores, high scores, asks if the player wants to restart.
- **`restartGame()`**: Resets the game to its initial state and restarts it.
- **`initializePlayerControls()`**: Initializes the control mappings for two players using keyboard input.
- **`initializePlayers(name1,color1,name2,color2)`**: Initializes 2 players;specified names and colours.
- **`handlePlayerAction(e,isPressed)`**: Handles player actions based on keyboard input.

- `updatePlayerPositions()`: Updates the positions of players based on their current direction.
- `updatePlayerMovement(playerId,keycode,direction)`: Updates the movement of a player based on the provided direction.
- `startGame ()`: Starts the game by initializing the game model, game panel, and menu bar.
- `getCurrentLevelSize()`: Returns the size of the current level as a `Dimension` object.
- `getGameModel()`: Returns the game model associated with this controller.

`ModelGame`

- `ModelGame(controllergame)`: Initialises game components and loads levels.
- `initializeGameComponents()`: Initialises the core components of the game model.
- `loadLevels()`: Loads all game levels.
- `startGame()`: Starts the game by setting the initial level and starting the timer.
- `setCurrentLevel()`: Sets the current game level.
- `initialiseGame()`: Initialises the game state for the current level.
- `checkGameState()`: Checks the game state for collisions.
- `hasCollided(player)`: Checks if the specified player has collided with boundaries or light trails.
- `determineWinner(playerThatMoved)`: Determines winner based on player's movement and collisions.
- `isOutOfBounds(player)`: Checks if the player is out of the game area boundaries.
- `getOpponentName(player)`: Returns the name of the opponent of the specified player.
- `updateWinnerScore(winnerName)`: Updates the score of the winner.
- `advanceToNextLevel()`: Advances the game to the next level.
- `endGame()`: Ends the game and stops the timer.
- `restartGame()`: Restarts the game from the first level.
- `intersects(player,lightTrail)`: Checks if a player intersects with a given light trail.
- `initializePlayers(name1,colour1,name2,colour2)`: Initialises two players in the game.
- `getCurrentLevel()`: Returns the current game level.
- `getTimer()`: Returns the game timer.
- `getPlayers()`: Returns the list of players in the game.
- `getCurrentLevelIndex()`: Returns the index of the current level.

`ModelLevel`

- `ModelLevel(number,width,height)`: Constructor that sets the level number, width, and height.
- `getWidth()`: Returns the width of the level.
- `getHeight()`: Returns the height of the level.
- `getLevel()`: Returns the level number.

`ModelPlayer`

- `ModelPlayer(name,colour)`: Initialises the player with a name and colour.
- `move(newDirection)`: Moves the player in the specified direction and updates the light trail.
- `resetPlayer(x,y)`: Resets the player's position, direction, and light trail.
- `setDirection(direction)`: Sets the player's movement direction.
- `updateLightTrail()`: Updates the player's light trail based on their current position.
- `resetLightTrail()`: Clears the player's light trail.
- `setXPosition(xPosition)`: Sets the player's x-coordinate position.
- `setYPosition(yPosition)`: Sets the player's y-coordinate position.

- `getLightTrail(playerThatMoved)`: Returns the player's light trail.
- `getName()`: Returns the player's name.
- `getColor()`: Returns the player's colour.
- `getScore()`: Returns the player's score.
- `setScore(score)`: Sets the player's score.
- `getXPosition()`: Returns the player's x-coordinate position.
- `getYPosition()`: Returns the player's y-coordinate position.
- `getDirection()`: Returns the player's current direction.

ModelStorage

- `updateScore(name,newPoints)`: Updates the score of a player in the database.
- `initializeGameComponents()`: Retrieves the top 10 high scores from the database.

ModelTimer

- `ModelTimer()`: Initialises game components and loads levels.
- `startTimer()`: Starts the timer task that increments the time.
- `stopTimer()`: Stops the timer and cancels any scheduled tasks.
- `resetTimer()`: Resets the timer to zero.
- `incrementTime()`: Increments the timer by one.
- `getTime()`: Returns the current time value of the timer.

ViewGame

- `ViewGame(controllergame)`: Initialises the game view with a controller.
- `initialiseGamePanel()`: Initialises the game panel.
- `initialiseMenuBar()`: Initialises the menu bar with game options.
- `updateTimerLabel()`: Updates the timer label with the current time.
- `updateCurrentLevelDisplay()`: Updates the display of the current level.
- `displayHighScores()`: Displays the high scores in a dialog.
- `drawGame(g)`: Draws the game state on the panel.
- `drawLightTrail(g,player)`: Draws the light trail of a player.
- `showWinnerDialog(playerThatMoved)`: Shows a dialog announcing the winner.
- `getGamePanel(player)`: Returns the game panel.
- `updateGame()`: Requests a repaint of the game panel.

ViewPlayer

- `ViewPlayer(controllergame)`: Initialises the player view with a controller.
- `initializeGameComponents()`: Initialises the player registration view.

ViewScore

- `ViewScore()`: Constructor that initializes the score view.
- `initializeScoreView()`: Initialises the score view window.
- `display()`: Displays the high scores in the score view window.

Event-Handler Connections

`ControllerGame`

Key Event Handlers: In the `ViewGame` class, the game panel (`gamePanel`) has key listeners added to it. These listeners call `controllergame.handlePlayerAction(Key Event e, boolean isPressed)` when key events occur.

- `keyPressed(KeyEvent e)`: Invokes `controllergame.handlePlayerAction(e, true)` when a key is pressed.
- `keyReleased(KeyEvent e)`: Invokes `controllergame.handlePlayerAction(e, false)` when a key is released.

`ViewGame`

Time Event Handler: The `timer` in `ViewGame` is set up to call `updateTimerLabel()` every 1000 ms.

Menu Item Action Handlers:

- **High Scores Menu Item**: The `highScoresMenuItem` in `initialiseMenuBar()` uses an action listener to display high scores. It calls `displayHighScores()` when clicked.
- **Exit Menu Item**: The `exitMenuItem` in `initialiseMenuBar()` uses an action listener to exit the application. It calls `System.exit(0)` when clicked.

`ViewPlayer`

Action Event Handler for Register Button: The `registerButton` in `initializePlayerView()` uses an action listener to handle player registration. It retrieves player names and colours from the input fields and combo boxes, then calls `controllergame.initializePlayers(name1, color1, name2, color2)` and `controllergame.startGame()`, and finally disposes of the registration frame.

`ModelGame`

No direct event handler connections are present in `ModelGame`. However, it is indirectly involved in event handling as it is manipulated by `ControllerGame` in response to game events.

`ControllerGame`

The `ControllerGame` class orchestrates much of the game's functionality in response to events. It acts as a mediator between the view (`ViewGame`, `ViewPlayer`, etc.) and the model (`ModelGame`, `ModelPlayer`, etc.).
The `Timer` object in `ControllerGame` (named `gameLoopTimer`) is crucial for continuously updating the game state. It triggers the `updateGame()` method at regular intervals (every 100 ms), which, in turn, handles game logic like movement updates and collision checks. The game employs `KeyListeners` to detect player inputs, and ActionListener for menu items, making the user interface interactive and responsive to user actions.

Tests

- **Initialization Test:** Validate `ControllerGame` creates `ViewGame` and `ModelGame`.
- **Player Move Test:** Ensure `ModelPlayer` position changes with direction input.
- **Collision Test:** Confirm collision detection works in `ModelGame`.
- **Score Test:** Check score recording and retrieval in `ModelStorage`.
- **Level Up Test:** Verify level change and timer reset in `ModelGame`.