snake

Contents

Description

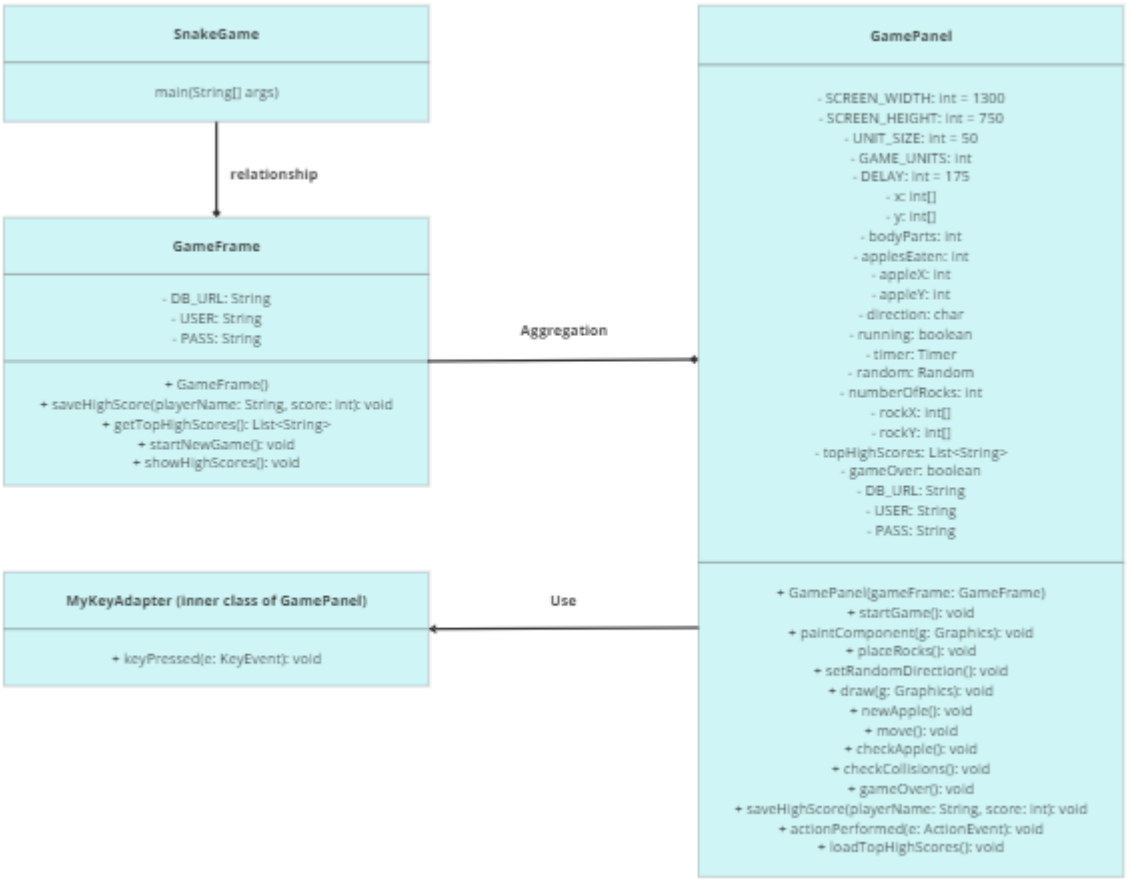
1. Snake

We have a rattlesnake in a desert, and our snake is initially two units long (head and rattler). We have to collect with our snake the foods on the level, that appears randomly. Only one food piece is placed randomly at a time on the level (on a field, where there is no snake). The snake starts off from the center of the level in a random direction. The player can control the movement of the snake's head with keyboard buttons. If the snake eats a food piece, then its length grow by one unit. It makes the game harder that there are rocks in the desert. If the snake collides with a rock, then the game ends. We also lose the game, if the snake goes into itself, or into the boundary of the game level. In these situations show a popup messagebox, where the player can type his name and save it together with the amount of food eaten to the database. 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

This game tests players' reflexes and strategic planning as they navigate through increasing difficulty. The randomness of food placement and initial direction add unpredictability, requiring on-the-fly decision making. The inclusion of a leaderboard for high scores introduces a competitive element, encouraging replayability. A restart feature ensures quick turnaround, keeping players engaged. Storing scores in a database and presenting a highscore table provides persistence and a sense of accomplishment. The game's design is simple yet effective in maintaining player interest.

Class Diagram



Description of each method

Directory Structure:

```
project_root/
|
```

```
├── Source Packages/
│   └── com.mycompany.assign3/
│       ├── GameFrame.java
│       ├── GamePanel.java
│       └── SnakeGame.java
│
├── Test Packages/
│   └── <default package>/
│       └── (test files for GameFrame, GamePanel, and SnakeGa
│
├── Dependencies/
│   ├── JDK 20 (Default)
│   └── Libraries/
│       ├── mysql-connector-java-8.0.23.jar
│       └── protobuf-java-3.11.4.jar
│
└── Project Files/
    └── pom.xml
```

GameFrame Class:

- `GameFrame()`: Constructor that sets up the game window with a menu bar containing items like New Game, High Scores, and Exit, and initializes the game panel.
- `getTopHighScores()`: Retrieves the top 10 high scores from the database and returns them as a list.
- `startNewGame()`: Stops the current game, removes the existing game panel, and starts a new game by adding a new game panel.
- `showHighScores()`: Displays a dialog with the top 10 high scores.
- `saveHighScore(String playerName, int score)`: Saves the player's name and score to the database when the game ends.

GamePanel Class:

- `GamePanel(GameFrame gameFrame)`: Constructor that initializes game variables and starts the game.
- `startGame()`: Sets initial conditions and starts the game timer.

- `paintComponent(Graphics g)`: Controls the drawing of the game's components on the panel.
- `placeRocks()`: Randomly places obstacles on the game field.
- `setRandomDirection()`: Randomly sets the initial direction of the player's character.
- `draw(Graphics g)`: Draws the game elements such as the player's character, obstacles, and score.
- `newApple()`: Places a new apple on the game field at a random location.
- `move()`: Updates the player's character position based on the current direction.
- `checkApple()`: Checks if the player's character has eaten an apple and updates the score.
- `checkCollisions()`: Checks for collisions with obstacles or borders, and ends the game if a collision occurs.
- `gameOver()`: Stops the game, displays the game over message, and prompts for player name for high score entry.
- `drawGameOver(Graphics g)`: Draws the game over screen with the final score.
- `actionPerformed(ActionEvent e)`: The main game loop that is called by the timer, handling game logic updates.
- `loadTopHighScores()`: Loads the top high scores from the database to be displayed on the game panel.

SnakeGame Class:

- `main(String[] args)`: The entry point of the program that creates an instance of `GameFrame` to start the game.

The connections between the events and event handlers

Event Handlers and Connections in Snake Game:

- **`newGameItem`**: Responds to clicks on the "New Game" menu item.
- **`highScoresItem`**: Responds to clicks on the "High Scores" menu item.

- `exitItem`: Responds to clicks on the "Exit" menu item.
- `timer`: Triggered at regular intervals to update game logic.
- `keyListener`: Responds to key presses to control the player's character.

Connections:

- `newGameItem` click event is connected to the `startNewGame` method in the `GameFrame` class.
- `highScoresItem` click event is connected to the `showHighScores` method in the `GameFrame` class.
- `exitItem` click event is connected to a lambda that calls `System.exit(0)`.
- `timer` event is connected to the `actionPerformed` method in the `GamePanel` class.
- `keyListener` events are connected to the `keyPressed` method in the `MyKeyAdapter` class inside `GamePanel`, which handles direction changes based on WASD key presses.

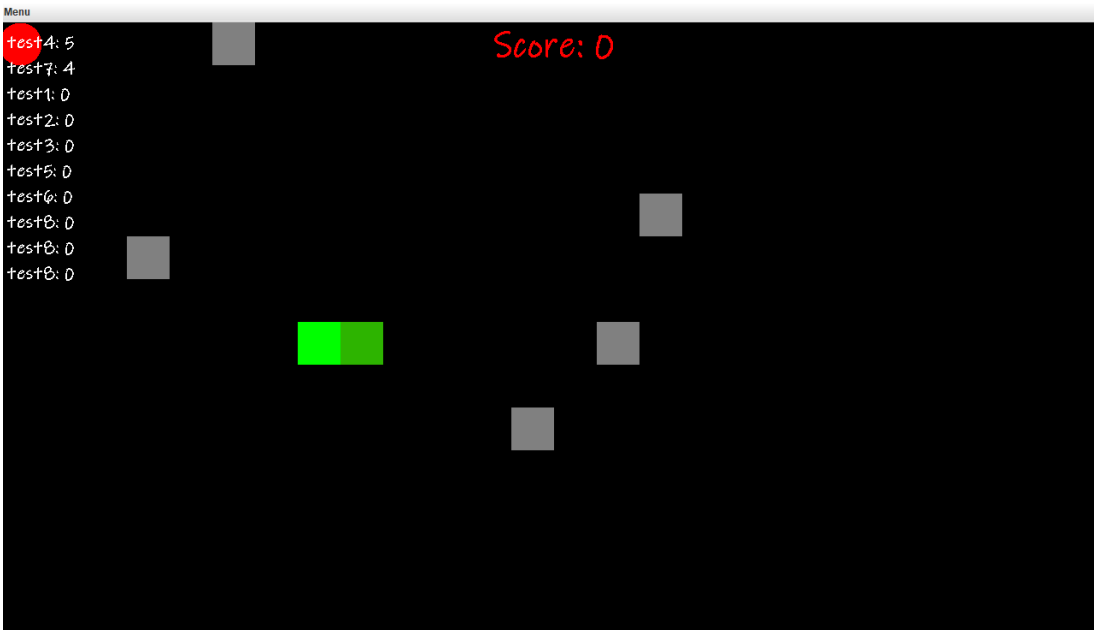
Implementation and Core Algorithms

The implementation of the Snake Game is structured around the `GamePanel` class, which encapsulates the main game loop and rendering logic. The `move()` method updates the snake's position, implementing the core gameplay mechanic. An interesting algorithm is the `checkCollisions()` method, which checks for a game-over condition by detecting collisions with rocks or the snake's own body. Another critical function is `newApple()`, which randomly places a new food item on the board in a location that is not currently occupied by the snake. This method uses a random number generator to select an unoccupied position, ensuring that the food appears in an accessible spot without overlapping the snake or rocks. The game level generation doesn't rely on a complex algorithm since the obstacles are static, but this could be an area for future enhancement by introducing randomly placed rocks or moving obstacles to increase difficulty.

Testing

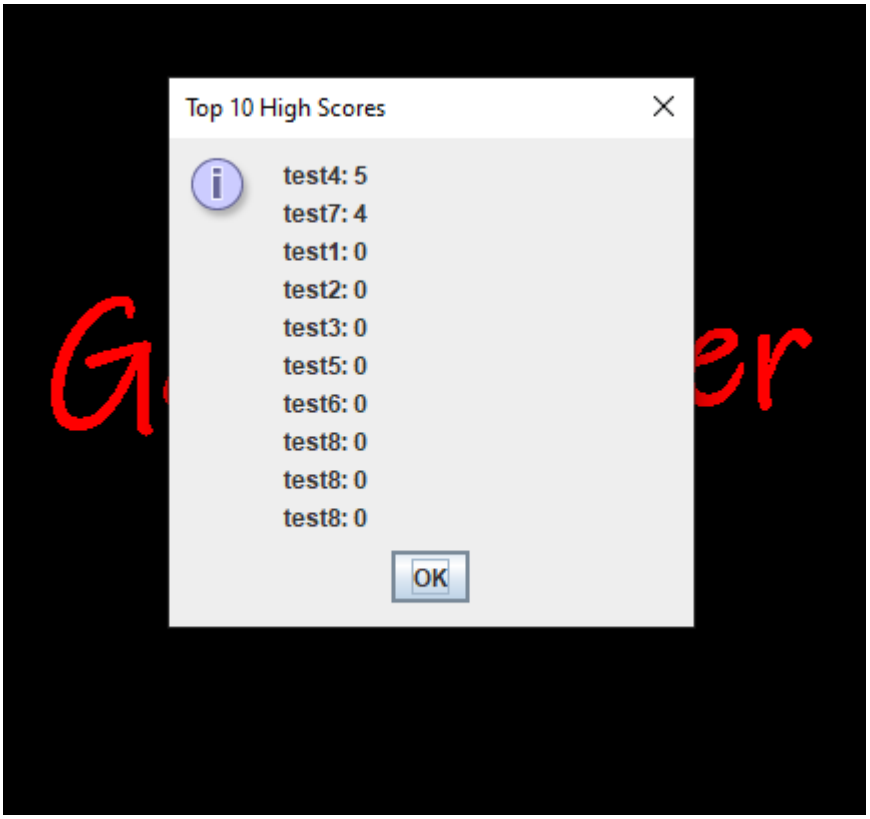
1. **Food Placement Test:**

- **Objective:** Ensure that food spawns in a free space not occupied by the snake or rocks.
- **Procedure:** Start the game and note the positions of the snake and rocks. Spawn food and check its position.
- **Expected Result:** Food appears in an empty space.

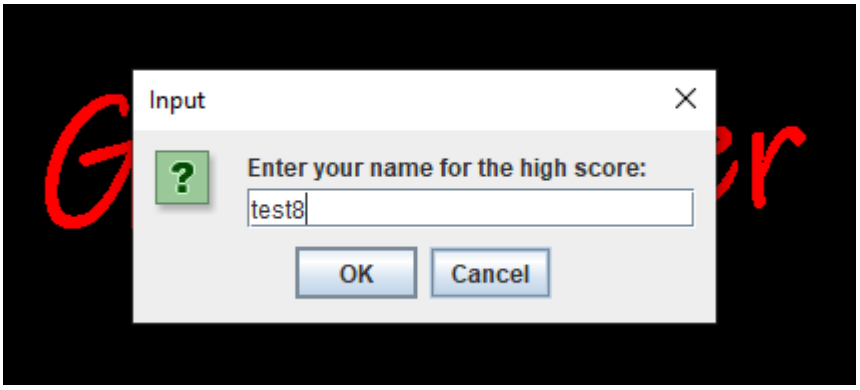


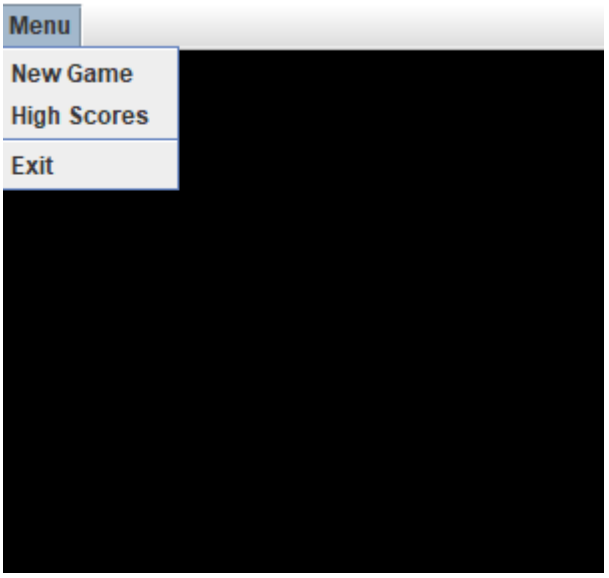
2. **Collision Detection Test:**
   - **Objective:** Verify that the game ends when the snake runs into a rock or itself.
   - **Procedure:** Direct the snake into a rock or its own body.
   - **Expected Result:** Game detects the collision and ends.
3. **Growth Mechanism Test:**
   - **Objective:** Confirm that the snake grows correctly after eating food.
   - **Procedure:** Control the snake to eat a piece of food and observe the snake's length.
   - **Expected Result:** The snake's length increases by one unit.
4. **Score and High Score Test:**
   - **Objective:** Check if the score updates properly and high scores are recorded.

- **Procedure:** Play the game, achieve a score, and then check the high score table after the game ends.
- **Expected Result:** Score during gameplay updates with each food item eaten, and the high score table reflects the new score if it is in the top ten.



5. **Menu Functionality Test:**
    - **Objective:** Ensure that the ‘New Game’ and ‘High Scores’ menu items function as expected.
    - **Procedure:** Click on the ‘New Game’ to start a new session, and ‘High Scores’ to view the high score table.
    - **Expected Result:** ‘New Game’ starts a new session, resetting the game state, and ‘High Scores’ displays the current top ten scores.





Add level speed, rock