Snake Game Documentation

-SARVESH JHAWAR

Overview

The Snake Game is a classic arcade-style game where the player controls a snake that moves around the screen, eats food, and grows in length. The objective is to eat as much food as possible without colliding with the walls or the snake's own body. This documentation provides a detailed explanation of the code structure, functionality, and logic of the game.

Class: SnakeGame

The **SnakeGame** class is the main class that implements the game logic and graphical interface. It extends **JPanel** to create a custom panel for rendering the game and implements **ActionListener** and **KeyListener** to handle user input and timer events.

Instance Variables

- **Tile Class**: Represents a single segment of the snake or food.
    - **int x**: The x-coordinate of the tile on the game board.
    - **int y**: The y-coordinate of the tile on the game board.
- **Game Board Dimensions**:
    - **int boardWidth**: The width of the game board in pixels.
    - **int boardHeight**: The height of the game board in pixels.
- **Snake Components**:
    - **Tile snakeHead**: Represents the head of the snake, which is the leading part that moves and interacts with food.
    - **ArrayList<Tile> snakeBody**: An ArrayList that holds the body segments of the snake, starting from the segment right behind the head and extending to the tail.
- **Food Component**:
    - **Tile food**: Represents the food that the snake can eat, which appears at random locations on the game board.
- **Random Number Generator**:
    - **Random random**: An instance of the **Random** class used to generate random positions for the food.
- **Game Logic**:
    - **Timer gameLoop**: A timer that controls the game loop, triggering updates at regular intervals.
    - **int velocityX**: The horizontal movement speed of the snake (positive for right, negative for left).

- **int velocityY**: The vertical movement speed of the snake (positive for down, negative for up).
- **boolean gameOver**: A flag that indicates whether the game is over.
- **int tileSize**: The size of each tile in pixels, which determines how the snake and food are rendered on the board.

Constructor: SnakeGame(int boardWidth, int boardHeight)

The constructor initializes the game board and sets up the initial state of the game.

- **Parameters**:
  - **int boardWidth**: The width of the game board in pixels.
  - **int boardHeight**: The height of the game board in pixels.

Key Lines:

- **setPreferredSize(new Dimension(this.boardWidth, this.boardHeight));**: Sets the preferred size of the game panel to the specified width and height.
- **setBackground(Color.black);**: Sets the background color of the game panel to black, providing a contrast for the snake and food.
- **addKeyListener(this);**: Registers the current instance of the class as a key listener, allowing it to respond to key presses.
- **setFocusable(true);**: Makes the panel focusable so that it can receive key events.
- **snakeHead = new Tile(5, 5);**: Initializes the snake's head at position (5, 5) on the game board.
- **snakeBody = new ArrayList<Tile>();**: Initializes the snake's body as an empty list, which will be populated as the game progresses.
- **food = new Tile(10, 10);**: Initializes the food at position (10, 10) on the game board.
- **random = new Random();**: Creates a new instance of the **Random** class for generating random numbers.
- **placeFood();**: Calls the method to place the food at a random location on the board.
- **velocityX = 0;**: Sets the initial horizontal movement speed of the snake to zero (no movement).
- **velocityY = 0;**: Sets the initial vertical movement speed of the snake to zero (no movement).
- **gameLoop = new Timer(100, this);**: Creates a timer that triggers every 100 milliseconds, calling the **actionPerformed** method.
- **gameLoop.start();**: Starts the timer, initiating the game loop.

Method: paintComponent(Graphics g)

This method is called whenever the game panel needs to be repainted. It is responsible for rendering the game elements on the screen.

- **Parameters**:
    - **Graphics g**: The graphics context used for drawing shapes, text, and images on the panel.

Key Lines:

- **super.paintComponent(g);**: Clears the panel before drawing new elements, ensuring that previous frames do not overlap.
- **draw(g);**: Calls the **draw** method to render the current state of the game, including the snake, food, and score.

Method: draw(Graphics g)

This method handles the rendering of the game elements, including the snake, food, and score display.

- **Parameters**:
    - **Graphics g**: The graphics context used for drawing.

Key Lines:

- **g.setColor(Color.red);**: Sets the color to red for drawing the food.
- **g.fill3DRect(food.x * tileSize, food.y * tileSize, tileSize, tileSize, true);**: Draws the food as a 3D rectangle at the food's position.
- **g.setColor(Color.green);**: Sets the color to green for drawing the snake head.
- **g.fill3DRect(snakeHead.x * tileSize, snakeHead.y * tileSize, tileSize, tileSize, true);**: Draws the snake head as a 3D rectangle at the head's position.
- The loop iterates through **snakeBody** to draw each body segment:
    - **for (int i = 0; i < snakeBody.size(); i++) {...}**: Iterates over each segment in the snake's body.
    - **Tile snakePart = snakeBody.get(i);**: Retrieves the current body segment.
    - **g.fill3DRect(snakePart.x * tileSize, snakePart.y * tileSize, tileSize, tileSize, true);**: Draws each body segment as a 3D rectangle.
- **g.setFont(new Font("Arial", Font.PLAIN, 16));**: Sets the font for displaying the score.
- **if (gameOver) {...}**: Checks if the game is over and displays the appropriate message:
    - **g.setColor(Color.red);**: Changes the color to red for the game over message.
    - **g.drawString("Game Over : " + String.valueOf(snakeBody.size()), tileSize - 16, tileSize);**: Displays the game over message along with the score.
- **g.drawString("Score: " + String.valueOf(snakeBody.size()), tileSize - 16, tileSize);**: Displays the current score if the game is still ongoing.

Method: placeFood()

This method places the food at a random location on the game board, ensuring that it appears within the bounds of the board.

Key Lines:

- **food.x = random.nextInt(boardWidth / tileSize);**: Randomly sets the x-coordinate of the food within the width of the board, ensuring it aligns with the tile size.
- **food.y = random.nextInt(boardHeight / tileSize);**: Randomly sets the y-coordinate of the food within the height of the board, ensuring it aligns with the tile size.

Method: collision(Tile tile1, Tile tile2)

This method checks if two tiles occupy the same position on the game board, which indicates a collision.

- **Parameters**:
    - **Tile tile1**: The first tile to check for collision.
    - **Tile tile2**: The second tile to check for collision.

Key Lines:

- **return tile1.x == tile2.x && tile1.y == tile2.y;**: Returns true if both tiles have the same x and y coordinates, indicating a collision.

Method: move()

This method updates the position of the snake and handles game logic, including checking for collisions with food, the snake's body, and the walls of the game board.

Key Lines:

- **if (collision(snakeHead, food)) {...}**: Checks if the snake's head has collided with the food:
    - **snakeBody.add(new Tile(food.x, food.y));**: Adds a new body segment at the food's position, effectively growing the snake.
    - **placeFood();**: Calls the method to place new food at a random location.
- The loop iterates through **snakeBody** to update the position of each body segment:
    - **for (int i = snakeBody.size() - 1; i >= 0; i--) {...}**: Iterates backward through the body segments.
    - **Tile snakePart = snakeBody.get(i);**: Retrieves the current body segment.
    - **if (i == 0) {...}**: If it's the first segment (the one right behind the head), it moves to the head's position:
        - **snakePart.x = snakeHead.x;**: Updates the x-coordinate of the first body segment to match the head.
        - **snakePart.y = snakeHead.y;**: Updates the y-coordinate of the first body segment to match the head.

- **else {...}**: For all other segments, it moves each segment to the position of the segment in front of it:
    - **Tile prevSnakePart = snakeBody.get(i - 1);**: Retrieves the previous segment.
    - **snakePart.x = prevSnakePart.x;**: Updates the current segment's x-coordinate to match the previous segment.
    - **snakePart.y = prevSnakePart.y;**: Updates the current segment's y-coordinate to match the previous segment.
- **snakeHead.x += velocityX;**: Updates the head's x position based on the current horizontal velocity.
- **snakeHead.y += velocityY;**: Updates the head's y position based on the current vertical velocity.
- The second loop checks for collisions with the body segments:
    - **for (int i = 0; i < snakeBody.size(); i++) {...}**: Iterates through each body segment.
    - **Tile snakePart = snakeBody.get(i);**: Retrieves the current body segment.
    - **if (collision(snakeHead, snakePart)) {...}**: Checks if the head collides with any body segment:
        - **gameOver = true;**: Sets the game over flag to true if a collision is detected.
- Checks for collisions with the walls of the game board:
    - **if (snakeHead.x * tileSize < 0 || snakeHead.x * tileSize > boardWidth || snakeHead.y * tileSize < 0 || snakeHead.y * tileSize > boardHeight) {...}**: Determines if the head goes out of bounds:
        - **gameOver = true;**: Sets the game over flag to true if the head goes out of bounds.

Method: keyPressed(KeyEvent e)

This method handles key presses to change the direction of the snake based on user input.

- **Parameters**:
    - **KeyEvent e**: The key event that contains information about the key pressed.

Key Lines:

- **if (e.getKeyCode() == KeyEvent.VK_UP && velocityY != 1) {...}**: Checks if the up arrow key is pressed and ensures the snake is not currently moving down:
    - **velocityX = 0;**: Sets horizontal movement to zero.
    - **velocityY = -1;**: Sets vertical movement to up.
- **else if (e.getKeyCode() == KeyEvent.VK_DOWN && velocityY != -1) {...}**: Checks if the down arrow key is pressed and ensures the snake is not currently moving up:
    - **velocityX = 0;**: Sets horizontal movement to zero.

    - **velocityY = 1;**: Sets vertical movement to down.
- **else if (e.getKeyCode() == KeyEvent.VK_LEFT && velocityX != 1) {...}**: Checks if the left arrow key is pressed and ensures the snake is not currently moving right:
    - **velocityX = -1;**: Sets horizontal movement to left.
    - **velocityY = 0;**: Sets vertical movement to zero.
- **else if (e.getKeyCode() == KeyEvent.VK_RIGHT && velocityX != -1) {...}**: Checks if the right arrow key is pressed and ensures the snake is not currently moving left:
    - **velocityX = 1;**: Sets horizontal movement to right.
    - **velocityY = 0;**: Sets vertical movement to zero.

Method: keyTyped(KeyEvent e)

This method is part of the **KeyListener** interface but is not used in this implementation. It can be left empty.

- **Parameters**:
    - **KeyEvent e**: The key event that contains information about the key typed.

Key Lines:

- **// No implementation needed**: Indicates that this method does not require any action.

Method: keyReleased(KeyEvent e)

This method is also part of the **KeyListener** interface but is not used in this implementation. It can be left empty.

- **Parameters**:
    - **KeyEvent e**: The key event that contains information about the key released.

Key Lines:

- **// No implementation needed**: Indicates that this method does not require any action.

Method: actionPerformed(ActionEvent e)

This method is called by the timer to update the game state and repaint the screen at regular intervals.

- **Parameters**:
    - **ActionEvent e**: The action event triggered by the timer.

Key Lines:

- **move();**: Calls the **move** method to update the game state, including the position of the snake and checking for collisions.
- **repaint();**: Calls the **paintComponent** method to redraw the game elements on the screen.

- **if (gameOver) { gameLoop.stop(); }**: Stops the game loop if the game is over, preventing further updates and rendering.

Conclusion

This detailed documentation provides a comprehensive explanation of the **SnakeGame** project, covering the purpose and functionality of each method, instance variable, and key lines of code.