|  |
| --- |
| Brendan Jones |
| SNAKE REMAKE |
| 10/08/2014 |

|  |
| --- |
| Brendan Jones  Zip-Source: https://github.com/PSNB92/SnakeRemake |

SnakeGame

**package** org.psnbtech;

**import** java.awt.BorderLayout;

**import** java.awt.Point;

**import** java.awt.event.KeyAdapter;

**import** java.awt.event.KeyEvent;

**import** java.util.LinkedList;

**import** java.util.Random;

**import** javax.swing.JFrame;

/\*\*

\* The {@code SnakeGame} class is responsible for handling much of the game's logic.

\* **@author** Brendan Jones

\*

\*/

**public** **class** SnakeGame **extends** JFrame {

/\*\*

\* The Serial Version UID.

\*/

**private** **static** **final** **long** ***serialVersionUID*** = 6678292058307426314L;

/\*\*

\* The number of milliseconds that should pass between each frame.

\*/

**private** **static** **final** **long** ***FRAME\_TIME*** = 1000L / 50L;

/\*\*

\* The minimum length of the snake. This allows the snake to grow

\* right when the game starts, so that we're not just a head moving

\* around on the board.

\*/

**private** **static** **final** **int** ***MIN\_SNAKE\_LENGTH*** = 5;

/\*\*

\* The maximum number of directions that we can have polled in the

\* direction list.

\*/

**private** **static** **final** **int** ***MAX\_DIRECTIONS*** = 3;

/\*\*

\* The BoardPanel instance.

\*/

**private** BoardPanel board;

/\*\*

\* The SidePanel instance.

\*/

**private** SidePanel side;

/\*\*

\* The random number generator (used for spawning fruits).

\*/

**private** Random random;

/\*\*

\* The Clock instance for handling the game logic.

\*/

**private** Clock logicTimer;

/\*\*

\* Whether or not we're running a new game.

\*/

**private** **boolean** isNewGame;

/\*\*

\* Whether or not the game is over.

\*/

**private** **boolean** isGameOver;

/\*\*

\* Whether or not the game is paused.

\*/

**private** **boolean** isPaused;

/\*\*

\* The list that contains the points for the snake.

\*/

**private** LinkedList<Point> snake;

/\*\*

\* The list that contains the queued directions.

\*/

**private** LinkedList<Direction> directions;

/\*\*

\* The current score.

\*/

**private** **int** score;

/\*\*

\* The number of fruits that we've eaten.

\*/

**private** **int** fruitsEaten;

/\*\*

\* The number of points that the next fruit will award us.

\*/

**private** **int** nextFruitScore;

/\*\*

\* Creates a new SnakeGame instance. Creates a new window,

\* and sets up the controller input.

\*/

**private** SnakeGame() {

**super**("Snake Remake");

setLayout(**new** BorderLayout());

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

setResizable(**false**);

/\*

\* Initialize the game's panels and add them to the window.

\*/

**this**.board = **new** BoardPanel(**this**);

**this**.side = **new** SidePanel(**this**);

add(board, BorderLayout.***CENTER***);

add(side, BorderLayout.***EAST***);

/\*

\* Adds a new key listener to the frame to process input.

\*/

addKeyListener(**new** KeyAdapter() {

@Override

**public** **void** keyPressed(KeyEvent e) {

**switch**(e.getKeyCode()) {

/\*

\* If the game is not paused, and the game is not over...

\*

\* Ensure that the direction list is not full, and that the most

\* recent direction is adjacent to North before adding the

\* direction to the list.

\*/

**case** KeyEvent.***VK\_W***:

**case** KeyEvent.***VK\_UP***:

**if**(!isPaused && !isGameOver) {

**if**(directions.size() < ***MAX\_DIRECTIONS***) {

Direction last = directions.peekLast();

**if**(last != Direction.***South*** && last != Direction.***North***) {

directions.addLast(Direction.***North***);

}

}

}

**break**;

/\*

\* If the game is not paused, and the game is not over...

\*

\* Ensure that the direction list is not full, and that the most

\* recent direction is adjacent to South before adding the

\* direction to the list.

\*/

**case** KeyEvent.***VK\_S***:

**case** KeyEvent.***VK\_DOWN***:

**if**(!isPaused && !isGameOver) {

**if**(directions.size() < ***MAX\_DIRECTIONS***) {

Direction last = directions.peekLast();

**if**(last != Direction.***North*** && last != Direction.***South***) {

directions.addLast(Direction.***South***);

}

}

}

**break**;

/\*

\* If the game is not paused, and the game is not over...

\*

\* Ensure that the direction list is not full, and that the most

\* recent direction is adjacent to West before adding the

\* direction to the list.

\*/

**case** KeyEvent.***VK\_A***:

**case** KeyEvent.***VK\_LEFT***:

**if**(!isPaused && !isGameOver) {

**if**(directions.size() < ***MAX\_DIRECTIONS***) {

Direction last = directions.peekLast();

**if**(last != Direction.***East*** && last != Direction.***West***) {

directions.addLast(Direction.***West***);

}

}

}

**break**;

/\*

\* If the game is not paused, and the game is not over...

\*

\* Ensure that the direction list is not full, and that the most

\* recent direction is adjacent to East before adding the

\* direction to the list.

\*/

**case** KeyEvent.***VK\_D***:

**case** KeyEvent.***VK\_RIGHT***:

**if**(!isPaused && !isGameOver) {

**if**(directions.size() < ***MAX\_DIRECTIONS***) {

Direction last = directions.peekLast();

**if**(last != Direction.***West*** && last != Direction.***East***) {

directions.addLast(Direction.***East***);

}

}

}

**break**;

/\*

\* If the game is not over, toggle the paused flag and update

\* the logicTimer's pause flag accordingly.

\*/

**case** KeyEvent.***VK\_P***:

**if**(!isGameOver) {

isPaused = !isPaused;

logicTimer.setPaused(isPaused);

}

**break**;

/\*

\* Reset the game if one is not currently in progress.

\*/

**case** KeyEvent.***VK\_ENTER***:

**if**(isNewGame || isGameOver) {

resetGame();

}

**break**;

}

}

});

/\*

\* Resize the window to the appropriate size, center it on the

\* screen and display it.

\*/

pack();

setLocationRelativeTo(**null**);

setVisible(**true**);

}

/\*\*

\* Starts the game running.

\*/

**private** **void** startGame() {

/\*

\* Initialize everything we're going to be using.

\*/

**this**.random = **new** Random();

**this**.snake = **new** LinkedList<>();

**this**.directions = **new** LinkedList<>();

**this**.logicTimer = **new** Clock(9.0f);

**this**.isNewGame = **true**;

//Set the timer to paused initially.

logicTimer.setPaused(**true**);

/\*

\* This is the game loop. It will update and render the game and will

\* continue to run until the game window is closed.

\*/

**while**(**true**) {

//Get the current frame's start time.

**long** start = System.*nanoTime*();

//Update the logic timer.

logicTimer.update();

/\*

\* If a cycle has elapsed on the logic timer, then update the game.

\*/

**if**(logicTimer.hasElapsedCycle()) {

updateGame();

}

//Repaint the board and side panel with the new content.

board.repaint();

side.repaint();

/\*

\* Calculate the delta time between since the start of the frame

\* and sleep for the excess time to cap the frame rate. While not

\* incredibly accurate, it is sufficient for our purposes.

\*/

**long** delta = (System.*nanoTime*() - start) / 1000000L;

**if**(delta < ***FRAME\_TIME***) {

**try** {

Thread.*sleep*(***FRAME\_TIME*** - delta);

} **catch**(Exception e) {

e.printStackTrace();

}

}

}

}

/\*\*

\* Updates the game's logic.

\*/

**private** **void** updateGame() {

/\*

\* Gets the type of tile that the head of the snake collided with. If

\* the snake hit a wall, SnakeBody will be returned, as both conditions

\* are handled identically.

\*/

TileType collision = updateSnake();

/\*

\* Here we handle the different possible collisions.

\*

\* Fruit: If we collided with a fruit, we increment the number of

\* fruits that we've eaten, update the score, and spawn a new fruit.

\*

\* SnakeBody: If we collided with our tail (or a wall), we flag that

\* the game is over and pause the game.

\*

\* If no collision occurred, we simply decrement the number of points

\* that the next fruit will give us if it's high enough. This adds a

\* bit of skill to the game as collecting fruits more quickly will

\* yield a higher score.

\*/

**if**(collision == TileType.***Fruit***) {

fruitsEaten++;

score += nextFruitScore;

spawnFruit();

} **else** **if**(collision == TileType.***SnakeBody***) {

isGameOver = **true**;

logicTimer.setPaused(**true**);

} **else** **if**(nextFruitScore > 10) {

nextFruitScore--;

}

}

/\*\*

\* Updates the snake's position and size.

\* **@return** Tile tile that the head moved into.

\*/

**private** TileType updateSnake() {

/\*

\* Here we peek at the next direction rather than polling it. While

\* not game breaking, polling the direction here causes a small bug

\* where the snake's direction will change after a game over (though

\* it will not move).

\*/

Direction direction = directions.peekFirst();

/\*

\* Here we calculate the new point that the snake's head will be at

\* after the update.

\*/

Point head = **new** Point(snake.peekFirst());

**switch**(direction) {

**case** ***North***:

head.y--;

**break**;

**case** ***South***:

head.y++;

**break**;

**case** ***West***:

head.x--;

**break**;

**case** ***East***:

head.x++;

**break**;

}

/\*

\* If the snake has moved out of bounds ('hit' a wall), we can just

\* return that it's collided with itself, as both cases are handled

\* identically.

\*/

**if**(head.x < 0 || head.x >= BoardPanel.***COL\_COUNT*** || head.y < 0 || head.y >= BoardPanel.***ROW\_COUNT***) {

**return** TileType.***SnakeBody***; //Pretend we collided with our body.

}

/\*

\* Here we get the tile that was located at the new head position and

\* remove the tail from of the snake and the board if the snake is

\* long enough, and the tile it moved onto is not a fruit.

\*

\* If the tail was removed, we need to retrieve the old tile again

\* incase the tile we hit was the tail piece that was just removed

\* to prevent a false game over.

\*/

TileType old = board.getTile(head.x, head.y);

**if**(old != TileType.***Fruit*** && snake.size() > ***MIN\_SNAKE\_LENGTH***) {

Point tail = snake.removeLast();

board.setTile(tail, **null**);

old = board.getTile(head.x, head.y);

}

/\*

\* Update the snake's position on the board if we didn't collide with

\* our tail:

\*

\* 1. Set the old head position to a body tile.

\* 2. Add the new head to the snake.

\* 3. Set the new head position to a head tile.

\*

\* If more than one direction is in the queue, poll it to read new

\* input.

\*/

**if**(old != TileType.***SnakeBody***) {

board.setTile(snake.peekFirst(), TileType.***SnakeBody***);

snake.push(head);

board.setTile(head, TileType.***SnakeHead***);

**if**(directions.size() > 1) {

directions.poll();

}

}

**return** old;

}

/\*\*

\* Resets the game's variables to their default states and starts a new game.

\*/

**private** **void** resetGame() {

/\*

\* Reset the score statistics. (Note that nextFruitPoints is reset in

\* the spawnFruit function later on).

\*/

**this**.score = 0;

**this**.fruitsEaten = 0;

/\*

\* Reset both the new game and game over flags.

\*/

**this**.isNewGame = **false**;

**this**.isGameOver = **false**;

/\*

\* Create the head at the center of the board.

\*/

Point head = **new** Point(BoardPanel.***COL\_COUNT*** / 2, BoardPanel.***ROW\_COUNT*** / 2);

/\*

\* Clear the snake list and add the head.

\*/

snake.clear();

snake.add(head);

/\*

\* Clear the board and add the head.

\*/

board.clearBoard();

board.setTile(head, TileType.***SnakeHead***);

/\*

\* Clear the directions and add north as the

\* default direction.

\*/

directions.clear();

directions.add(Direction.***North***);

/\*

\* Reset the logic timer.

\*/

logicTimer.reset();

/\*

\* Spawn a new fruit.

\*/

spawnFruit();

}

/\*\*

\* Gets the flag that indicates whether or not we're playing a new game.

\* **@return** The new game flag.

\*/

**public** **boolean** isNewGame() {

**return** isNewGame;

}

/\*\*

\* Gets the flag that indicates whether or not the game is over.

\* **@return** The game over flag.

\*/

**public** **boolean** isGameOver() {

**return** isGameOver;

}

/\*\*

\* Gets the flag that indicates whether or not the game is paused.

\* **@return** The paused flag.

\*/

**public** **boolean** isPaused() {

**return** isPaused;

}

/\*\*

\* Spawns a new fruit onto the board.

\*/

**private** **void** spawnFruit() {

//Reset the score for this fruit to 100.

**this**.nextFruitScore = 100;

/\*

\* Get a random index based on the number of free spaces left on the board.

\*/

**int** index = random.nextInt(BoardPanel.***COL\_COUNT*** \* BoardPanel.***ROW\_COUNT*** - snake.size());

/\*

\* While we could just as easily choose a random index on the board

\* and check it if it's free until we find an empty one, that method

\* tends to hang if the snake becomes very large.

\*

\* This method simply loops through until it finds the nth free index

\* and selects uses that. This means that the game will be able to

\* locate an index at a relatively constant rate regardless of the

\* size of the snake.

\*/

**int** freeFound = -1;

**for**(**int** x = 0; x < BoardPanel.***COL\_COUNT***; x++) {

**for**(**int** y = 0; y < BoardPanel.***ROW\_COUNT***; y++) {

TileType type = board.getTile(x, y);

**if**(type == **null** || type == TileType.***Fruit***) {

**if**(++freeFound == index) {

board.setTile(x, y, TileType.***Fruit***);

**break**;

}

}

}

}

}

/\*\*

\* Gets the current score.

\* **@return** The score.

\*/

**public** **int** getScore() {

**return** score;

}

/\*\*

\* Gets the number of fruits eaten.

\* **@return** The fruits eaten.

\*/

**public** **int** getFruitsEaten() {

**return** fruitsEaten;

}

/\*\*

\* Gets the next fruit score.

\* **@return** The next fruit score.

\*/

**public** **int** getNextFruitScore() {

**return** nextFruitScore;

}

/\*\*

\* Gets the current direction of the snake.

\* **@return** The current direction.

\*/

**public** Direction getDirection() {

**return** directions.peek();

}

/\*\*

\* Entry point of the program.

\* **@param** args Unused.

\*/

**public** **static** **void** main(String[] args) {

SnakeGame snake = **new** SnakeGame();

snake.startGame();

}

}

BoardPanel

**package** org.psnbtech;

**import** java.awt.Color;

**import** java.awt.Dimension;

**import** java.awt.Font;

**import** java.awt.Graphics;

**import** java.awt.Point;

**import** javax.swing.JPanel;

/\*\*

\* The {@code BoardPanel} class is responsible for managing and displaying the

\* contents of the game board.

\* **@author** Brendan Jones

\*

\*/

**public** **class** BoardPanel **extends** JPanel {

/\*\*

\* Serial Version UID.

\*/

**private** **static** **final** **long** ***serialVersionUID*** = -1102632585936750607L;

/\*\*

\* The number of columns on the board. (Should be odd so we can start in

\* the center).

\*/

**public** **static** **final** **int** ***COL\_COUNT*** = 25;

/\*\*

\* The number of rows on the board. (Should be odd so we can start in

\* the center).

\*/

**public** **static** **final** **int** ***ROW\_COUNT*** = 25;

/\*\*

\* The size of each tile in pixels.

\*/

**public** **static** **final** **int** ***TILE\_SIZE*** = 20;

/\*\*

\* The number of pixels to offset the eyes from the sides.

\*/

**private** **static** **final** **int** ***EYE\_LARGE\_INSET*** = ***TILE\_SIZE*** / 3;

/\*\*

\* The number of pixels to offset the eyes from the front.

\*/

**private** **static** **final** **int** ***EYE\_SMALL\_INSET*** = ***TILE\_SIZE*** / 6;

/\*\*

\* The length of the eyes from the base (small inset).

\*/

**private** **static** **final** **int** ***EYE\_LENGTH*** = ***TILE\_SIZE*** / 5;

/\*\*

\* The font to draw the text with.

\*/

**private** **static** **final** Font ***FONT*** = **new** Font("Tahoma", Font.***BOLD***, 25);

/\*\*

\* The SnakeGame instance.

\*/

**private** SnakeGame game;

/\*\*

\* The array of tiles that make up this board.

\*/

**private** TileType[] tiles;

/\*\*

\* Creates a new BoardPanel instance.

\* **@param** game The SnakeGame instance.

\*/

**public** BoardPanel(SnakeGame game) {

**this**.game = game;

**this**.tiles = **new** TileType[***ROW\_COUNT*** \* ***COL\_COUNT***];

setPreferredSize(**new** Dimension(***COL\_COUNT*** \* ***TILE\_SIZE***, ***ROW\_COUNT*** \* ***TILE\_SIZE***));

setBackground(Color.***BLACK***);

}

/\*\*

\* Clears all of the tiles on the board and sets their values to null.

\*/

**public** **void** clearBoard() {

**for**(**int** i = 0; i < tiles.length; i++) {

tiles[i] = **null**;

}

}

/\*\*

\* Sets the tile at the desired coordinate.

\* **@param** point The coordinate of the tile.

\* **@param** type The type to set the tile to.

\*/

**public** **void** setTile(Point point, TileType type) {

setTile(point.x, point.y, type);

}

/\*\*

\* Sets the tile at the desired coordinate.

\* **@param** x The x coordinate of the tile.

\* **@param** y The y coordinate of the tile.

\* **@param** type The type to set the tile to.

\*/

**public** **void** setTile(**int** x, **int** y, TileType type) {

tiles[y \* ***ROW\_COUNT*** + x] = type;

}

/\*\*

\* Gets the tile at the desired coordinate.

\* **@param** x The x coordinate of the tile.

\* **@param** y The y coordinate of the tile.

\* **@return**

\*/

**public** TileType getTile(**int** x, **int** y) {

**return** tiles[y \* ***ROW\_COUNT*** + x];

}

@Override

**public** **void** paintComponent(Graphics g) {

**super**.paintComponent(g);

/\*

\* Loop through each tile on the board and draw it if it

\* is not null.

\*/

**for**(**int** x = 0; x < ***COL\_COUNT***; x++) {

**for**(**int** y = 0; y < ***ROW\_COUNT***; y++) {

TileType type = getTile(x, y);

**if**(type != **null**) {

drawTile(x \* ***TILE\_SIZE***, y \* ***TILE\_SIZE***, type, g);

}

}

}

/\*

\* Draw the grid on the board. This makes it easier to see exactly

\* where we in relation to the fruit.

\*

\* The panel is one pixel too small to draw the bottom and right

\* outlines, so we outline the board with a rectangle separately.

\*/

g.setColor(Color.***DARK\_GRAY***);

g.drawRect(0, 0, getWidth() - 1, getHeight() - 1);

**for**(**int** x = 0; x < ***COL\_COUNT***; x++) {

**for**(**int** y = 0; y < ***ROW\_COUNT***; y++) {

g.drawLine(x \* ***TILE\_SIZE***, 0, x \* ***TILE\_SIZE***, getHeight());

g.drawLine(0, y \* ***TILE\_SIZE***, getWidth(), y \* ***TILE\_SIZE***);

}

}

/\*

\* Show a message on the screen based on the current game state.

\*/

**if**(game.isGameOver() || game.isNewGame() || game.isPaused()) {

g.setColor(Color.***WHITE***);

/\*

\* Get the center coordinates of the board.

\*/

**int** centerX = getWidth() / 2;

**int** centerY = getHeight() / 2;

/\*

\* Allocate the messages for and set their values based on the game

\* state.

\*/

String largeMessage = **null**;

String smallMessage = **null**;

**if**(game.isNewGame()) {

largeMessage = "Snake Game!";

smallMessage = "Press Enter to Start";

} **else** **if**(game.isGameOver()) {

largeMessage = "Game Over!";

smallMessage = "Press Enter to Restart";

} **else** **if**(game.isPaused()) {

largeMessage = "Paused";

smallMessage = "Press P to Resume";

}

/\*

\* Set the message font and draw the messages in the center of the board.

\*/

g.setFont(***FONT***);

g.drawString(largeMessage, centerX - g.getFontMetrics().stringWidth(largeMessage) / 2, centerY - 50);

g.drawString(smallMessage, centerX - g.getFontMetrics().stringWidth(smallMessage) / 2, centerY + 50);

}

}

/\*\*

\* Draws a tile onto the board.

\* **@param** x The x coordinate of the tile (in pixels).

\* **@param** y The y coordinate of the tile (in pixels).

\* **@param** type The type of tile to draw.

\* **@param** g The graphics object to draw to.

\*/

**private** **void** drawTile(**int** x, **int** y, TileType type, Graphics g) {

/\*

\* Because each type of tile is drawn differently, it's easiest

\* to just run through a switch statement rather than come up with some

\* overly complex code to handle everything.

\*/

**switch**(type) {

/\*

\* A fruit is depicted as a small red circle that with a bit of padding

\* on each side.

\*/

**case** ***Fruit***:

g.setColor(Color.***RED***);

g.fillOval(x + 2, y + 2, ***TILE\_SIZE*** - 4, ***TILE\_SIZE*** - 4);

**break**;

/\*

\* The snake body is depicted as a green square that takes up the

\* entire tile.

\*/

**case** ***SnakeBody***:

g.setColor(Color.***GREEN***);

g.fillRect(x, y, ***TILE\_SIZE***, ***TILE\_SIZE***);

**break**;

/\*

\* The snake head is depicted similarly to the body, but with two

\* lines (representing eyes) that indicate it's direction.

\*/

**case** ***SnakeHead***:

//Fill the tile in with green.

g.setColor(Color.***GREEN***);

g.fillRect(x, y, ***TILE\_SIZE***, ***TILE\_SIZE***);

//Set the color to black so that we can start drawing the eyes.

g.setColor(Color.***BLACK***);

/\*

\* The eyes will always 'face' the direction that the snake is

\* moving.

\*

\* Vertical lines indicate that it's facing North or South, and

\* Horizontal lines indicate that it's facing East or West.

\*

\* Additionally, the eyes will be closer to whichever edge it's

\* facing.

\*

\* Drawing the eyes is fairly simple, but is a bit difficult to

\* explain. The basic process is this:

\*

\* First, we add (or subtract) EYE\_SMALL\_INSET to or from the

\* side of the tile representing the direction we're facing. This

\* will be constant for both eyes, and is represented by the

\* variable 'baseX' or 'baseY' (depending on orientation).

\*

\* Next, we add (or subtract) EYE\_LARGE\_INSET to and from the two

\* neighboring directions (Example; East and West if we're facing

\* north).

\*

\* Finally, we draw a line from the base offset that is EYE\_LENGTH

\* pixels in length at whatever the offset is from the neighboring

\* directions.

\*

\*/

**switch**(game.getDirection()) {

**case** ***North***: {

**int** baseY = y + ***EYE\_SMALL\_INSET***;

g.drawLine(x + ***EYE\_LARGE\_INSET***, baseY, x + ***EYE\_LARGE\_INSET***, baseY + ***EYE\_LENGTH***);

g.drawLine(x + ***TILE\_SIZE*** - ***EYE\_LARGE\_INSET***, baseY, x + ***TILE\_SIZE*** - ***EYE\_LARGE\_INSET***, baseY + ***EYE\_LENGTH***);

**break**;

}

**case** ***South***: {

**int** baseY = y + ***TILE\_SIZE*** - ***EYE\_SMALL\_INSET***;

g.drawLine(x + ***EYE\_LARGE\_INSET***, baseY, x + ***EYE\_LARGE\_INSET***, baseY - ***EYE\_LENGTH***);

g.drawLine(x + ***TILE\_SIZE*** - ***EYE\_LARGE\_INSET***, baseY, x + ***TILE\_SIZE*** - ***EYE\_LARGE\_INSET***, baseY - ***EYE\_LENGTH***);

**break**;

}

**case** ***West***: {

**int** baseX = x + ***EYE\_SMALL\_INSET***;

g.drawLine(baseX, y + ***EYE\_LARGE\_INSET***, baseX + ***EYE\_LENGTH***, y + ***EYE\_LARGE\_INSET***);

g.drawLine(baseX, y + ***TILE\_SIZE*** - ***EYE\_LARGE\_INSET***, baseX + ***EYE\_LENGTH***, y + ***TILE\_SIZE*** - ***EYE\_LARGE\_INSET***);

**break**;

}

**case** ***East***: {

**int** baseX = x + ***TILE\_SIZE*** - ***EYE\_SMALL\_INSET***;

g.drawLine(baseX, y + ***EYE\_LARGE\_INSET***, baseX - ***EYE\_LENGTH***, y + ***EYE\_LARGE\_INSET***);

g.drawLine(baseX, y + ***TILE\_SIZE*** - ***EYE\_LARGE\_INSET***, baseX - ***EYE\_LENGTH***, y + ***TILE\_SIZE*** - ***EYE\_LARGE\_INSET***);

**break**;

}

}

**break**;

}

}

}

Clock

**package** org.psnbtech;

/\*\*

\* The {@code Clock} class is responsible for tracking the number of cycles

\* that have elapsed over time.

\* **@author** Brendan Jones

\*

\*/

**public** **class** Clock {

/\*\*

\* The number of milliseconds that make up one cycle.

\*/

**private** **float** millisPerCycle;

/\*\*

\* The last time that the clock was updated (used for calculating the

\* delta time).

\*/

**private** **long** lastUpdate;

/\*\*

\* The number of cycles that have elapsed and have not yet been polled.

\*/

**private** **int** elapsedCycles;

/\*\*

\* The amount of excess time towards the next elapsed cycle.

\*/

**private** **float** excessCycles;

/\*\*

\* Whether or not the clock is paused.

\*/

**private** **boolean** isPaused;

/\*\*

\* Creates a new clock and sets it's cycles-per-second.

\* **@param** cyclesPerSecond The number of cycles that elapse per second.

\*/

**public** Clock(**float** cyclesPerSecond) {

setCyclesPerSecond(cyclesPerSecond);

reset();

}

/\*\*

\* Sets the number of cycles that elapse per second.

\* **@param** cyclesPerSecond The number of cycles per second.

\*/

**public** **void** setCyclesPerSecond(**float** cyclesPerSecond) {

**this**.millisPerCycle = (1.0f / cyclesPerSecond) \* 1000;

}

/\*\*

\* Resets the clock stats. Elapsed cycles and cycle excess will be reset

\* to 0, the last update time will be reset to the current time, and the

\* paused flag will be set to false.

\*/

**public** **void** reset() {

**this**.elapsedCycles = 0;

**this**.excessCycles = 0.0f;

**this**.lastUpdate = *getCurrentTime*();

**this**.isPaused = **false**;

}

/\*\*

\* Updates the clock stats. The number of elapsed cycles, as well as the

\* cycle excess will be calculated only if the clock is not paused. This

\* method should be called every frame even when paused to prevent any

\* nasty surprises with the delta time.

\*/

**public** **void** update() {

//Get the current time and calculate the delta time.

**long** currUpdate = *getCurrentTime*();

**float** delta = (**float**)(currUpdate - lastUpdate) + excessCycles;

//Update the number of elapsed and excess ticks if we're not paused.

**if**(!isPaused) {

**this**.elapsedCycles += (**int**)Math.*floor*(delta / millisPerCycle);

**this**.excessCycles = delta % millisPerCycle;

}

//Set the last update time for the next update cycle.

**this**.lastUpdate = currUpdate;

}

/\*\*

\* Pauses or unpauses the clock. While paused, a clock will not update

\* elapsed cycles or cycle excess, though the {@code update} method should

\* still be called every frame to prevent issues.

\* **@param** paused Whether or not to pause this clock.

\*/

**public** **void** setPaused(**boolean** paused) {

**this**.isPaused = paused;

}

/\*\*

\* Checks to see if the clock is currently paused.

\* **@return** Whether or not this clock is paused.

\*/

**public** **boolean** isPaused() {

**return** isPaused;

}

/\*\*

\* Checks to see if a cycle has elapsed for this clock yet. If so,

\* the number of elapsed cycles will be decremented by one.

\* **@return** Whether or not a cycle has elapsed.

\* **@see** peekElapsedCycle

\*/

**public** **boolean** hasElapsedCycle() {

**if**(elapsedCycles > 0) {

**this**.elapsedCycles--;

**return** **true**;

}

**return** **false**;

}

/\*\*

\* Checks to see if a cycle has elapsed for this clock yet. Unlike

\* {@code hasElapsedCycle}, the number of cycles will not be decremented

\* if the number of elapsed cycles is greater than 0.

\* **@return** Whether or not a cycle has elapsed.

\* **@see** hasElapsedCycle

\*/

**public** **boolean** peekElapsedCycle() {

**return** (elapsedCycles > 0);

}

/\*\*

\* Calculates the current time in milliseconds using the computer's high

\* resolution clock. This is much more reliable than

\* {@code System.getCurrentTimeMillis()}, and quicker than

\* {@code System.nanoTime()}.

\* **@return** The current time in milliseconds.

\*/

**private** **static** **final** **long** getCurrentTime() {

**return** (System.*nanoTime*() / 1000000L);

}

}

Direction

**package** org.psnbtech;

/\*\*

\* The {@code Direction} enum is used to determine which way the Snake is moving.

\* **@author** Brendan Jones

\*

\*/

**public** **enum** Direction {

/\*\*

\* Moving North (Up).

\*/

***North***,

/\*\*

\* Moving East (Right).

\*/

***East***,

/\*\*

\* Moving South (Down).

\*/

***South***,

/\*\*

\* Moving West (Left).

\*/

***West***

}

SidePanel

**package** org.psnbtech;

**import** java.awt.Color;

**import** java.awt.Dimension;

**import** java.awt.Font;

**import** java.awt.Graphics;

**import** javax.swing.JPanel;

/\*\*

\* The {@code SidePanel} class is responsible for displaying statistics and

\* controls to the player.

\* **@author** Brendan Jones

\*

\*/

**public** **class** SidePanel **extends** JPanel {

/\*\*

\* Serial Version UID.

\*/

**private** **static** **final** **long** ***serialVersionUID*** = -40557434900946408L;

/\*\*

\* The large font to draw with.

\*/

**private** **static** **final** Font ***LARGE\_FONT*** = **new** Font("Tahoma", Font.***BOLD***, 20);

/\*\*

\* The medium font to draw with.

\*/

**private** **static** **final** Font ***MEDIUM\_FONT*** = **new** Font("Tahoma", Font.***BOLD***, 16);

/\*\*

\* The small font to draw with.

\*/

**private** **static** **final** Font ***SMALL\_FONT*** = **new** Font("Tahoma", Font.***BOLD***, 12);

/\*\*

\* The SnakeGame instance.

\*/

**private** SnakeGame game;

/\*\*

\* Creates a new SidePanel instance.

\* **@param** game The SnakeGame instance.

\*/

**public** SidePanel(SnakeGame game) {

**this**.game = game;

setPreferredSize(**new** Dimension(300, BoardPanel.***ROW\_COUNT*** \* BoardPanel.***TILE\_SIZE***));

setBackground(Color.***BLACK***);

}

**private** **static** **final** **int** ***STATISTICS\_OFFSET*** = 150;

**private** **static** **final** **int** ***CONTROLS\_OFFSET*** = 320;

**private** **static** **final** **int** ***MESSAGE\_STRIDE*** = 30;

**private** **static** **final** **int** ***SMALL\_OFFSET*** = 30;

**private** **static** **final** **int** ***LARGE\_OFFSET*** = 50;

@Override

**public** **void** paintComponent(Graphics g) {

**super**.paintComponent(g);

/\*

\* Set the color to draw the font in to white.

\*/

g.setColor(Color.***WHITE***);

/\*

\* Draw the game name onto the window.

\*/

g.setFont(***LARGE\_FONT***);

g.drawString("Snake Game", getWidth() / 2 - g.getFontMetrics().stringWidth("Snake Game") / 2, 50);

/\*

\* Draw the categories onto the window.

\*/

g.setFont(***MEDIUM\_FONT***);

g.drawString("Statistics", ***SMALL\_OFFSET***, ***STATISTICS\_OFFSET***);

g.drawString("Controls", ***SMALL\_OFFSET***, ***CONTROLS\_OFFSET***);

/\*

\* Draw the category content onto the window.

\*/

g.setFont(***SMALL\_FONT***);

//Draw the content for the statistics category.

**int** drawY = ***STATISTICS\_OFFSET***;

g.drawString("Total Score: " + game.getScore(), ***LARGE\_OFFSET***, drawY += ***MESSAGE\_STRIDE***);

g.drawString("Fruit Eaten: " + game.getFruitsEaten(), ***LARGE\_OFFSET***, drawY += ***MESSAGE\_STRIDE***);

g.drawString("Fruit Score: " + game.getNextFruitScore(), ***LARGE\_OFFSET***, drawY += ***MESSAGE\_STRIDE***);

//Draw the content for the controls category.

drawY = ***CONTROLS\_OFFSET***;

g.drawString("Move Up: W / Up Arrowkey", ***LARGE\_OFFSET***, drawY += ***MESSAGE\_STRIDE***);

g.drawString("Move Down: S / Down Arrowkey", ***LARGE\_OFFSET***, drawY += ***MESSAGE\_STRIDE***);

g.drawString("Move Left: A / Left Arrowkey", ***LARGE\_OFFSET***, drawY += ***MESSAGE\_STRIDE***);

g.drawString("Move Right: D / Right Arrowkey", ***LARGE\_OFFSET***, drawY += ***MESSAGE\_STRIDE***);

g.drawString("Pause Game: P", ***LARGE\_OFFSET***, drawY += ***MESSAGE\_STRIDE***);

}

}

TyleType

**package** org.psnbtech;

/\*\*

\* The {@code TileType} class represents the different

\* types of tiles that can be displayed on the screen.

\* **@author** Brendan Jones

\*

\*/

**public** **enum** TileType {

***Fruit***,

***SnakeHead***,

***SnakeBody***

}