By: Izel Escoto

CPSC-39-12112

Mouse Catcher

12/1/2024

What your game or app does and how it is useful or entertaining.

In my game, you take on the role of a snake hunting for mice hidden in the tall grass. If you venture out of the grass, it's game over, and if you accidentally bite your own tail while sneaking up on a mouse, you lose too.

Includes at least 3 algorithms as steps or a flowchart,

and a snapshot of the algorithms code.

```
76      public void newMouse() {  2 usages
77          mouseX = random.nextInt((int) (SCREEN_WIDTH / UNIT_SIZE)) * UNIT_SIZE;
78          mouseY = random.nextInt((int) (SCREEN_HEIGHT / UNIT_SIZE)) * UNIT_SIZE;
79      }
80
```

Steps:

1) Generate a random x coordinate within the grid using random.nextInt(SCREEN_WIDTH / UNIT_SIZE) * UNIT_SIZE.

2) Generate a random y coordinate in the same way.

3) Assign these coordinates to mouseX and mouseY.

Mouse Placement Algorithm (newMouse method)

The algorithm randomly places the "mouse" at a new location within the grid of the game panel

Includes at least 3 algorithms as steps or a flowchart, and a snapshot of the algorithms code.

Snake Movement Algorithm (move method)

This algorithm updates the snake's position as it moves

Steps:

1) For each body part (starting from the tail), set its position to the position of the segment ahead.

2) Update the head's position (x[0] and y[0]) based on the current direction.

3) Add the new head position to “head Position History” for tracking.

```
80
81      public void move() {  1 usage
82          for (int i = bodyParts; i > 0; i--) {
83              x[i] = x[i - 1];
84              y[i] = y[i - 1];
85          }
86
87          switch (direction) {
88              case 'U':
89                  y[0] = y[0] - UNIT_SIZE;
90                  break;
91              case 'D':
92                  y[0] = y[0] + UNIT_SIZE;
93                  break;
94              case 'L':
95                  x[0] = x[0] - UNIT_SIZE;
96                  break;
97              case 'R':
98                  x[0] = x[0] + UNIT_SIZE;
99                  break;
100         }
101
102         // Add the new head position to the history
103         headPositionHistory.add(new Point(x[0], y[0]));
104     }
```

Includes at least 3 algorithms as steps or a flowchart, and a snapshot of the algorithms code.

Collision Detection Algorithm (check Collisions method)
This algorithm ensures the game ends if the snake collides with itself or the boundaries

Steps:

1) Loop through each body part and check if its position matches the head's position. If so, set running = false.

2) Check if the head's position is less than 0 or greater than the screen width/height.

3) Stop the game timer if a collision is detected.

```
114    public void checkCollisions() {  1 usage
115        for (int i = bodyParts; i > 0; i--) {
116            if ((x[0] == x[i]) && (y[0] == y[i])) {
117                running = false;
118            }
119        }
120        if (x[0] < 0 || x[0] > SCREEN_WIDTH || y[0] < 0 || y[0] > SCREEN_HEIGHT) {
121            running = false;
122        }
123
124        if (!running) {
125            timer.stop();
126        }
127    }
```

Explains in a paragraph of the 3 algorithms that you created and how they are used in the game or app. How you created them is important, and if you used ChatGPT here you can explain how you used it

The game runs on three key algorithms to keep the game running smooth and enjoyable. First is the Mouse Placement Algorithm, which randomly drops the food (mouse) on the grid, making sure it stays in the boundaries and spawns a new target every time the snake munches on it. Next, we have the Snake Movement Algorithm, this controls how the snake moves by shifting each segment to follow the one in front, while the head's position updates based on the direction chosen, creating a flow. Lastly, the Collision Detection Algorithm wraps things up by ending the game if the snake's head crashes into its own body or goes off the grid. All these algorithms work together to create a game that's both challenging and addicting to play.

Discusses the Big O time of these algorithms.

Mouse Placement Algorithm

Operation: Generates random coordinates for the food within the grid.

Time Complexity: O(1)O(1)O(1)
This algorithm uses a constant number of operations (random number generation and assignment), making it independent of the size of the game grid or the snake's body length.

Collision Detection Algorithm

Operation: Checks for collisions between the snake's head and its body or game boundaries.

Time Complexity: O(n)O(n)O(n), where nnn is the number of body parts.
The algorithm loops through the body parts to compare each position with the head's position. The boundary checks are O(1)O(1)O(1), but the body collision check dominates the complexity and grows linearly with the snake's length.

Snake Movement Algorithm

Operation: Updates the position of the snake's body parts and the head.

Time Complexity: O(n)O(n)O(n), where nnn is the number of body parts.
The algorithm iterates through the body of the snake to update the position of each segment. As the snake grows longer, the number of operations increases linearly with its size.

An explanation of the data structures that you used, why you chose them, and how they were used.

I used three data structures during this

The Random object helped me create random spots for the mouse so mouseX and mouseY, making sure that every game session feels fresh and full of surprises. This method was very useful.

I used arrays x [] and y [] in my game to keep track of the snake's body parts. Its a basic data structure in Java that arranges elements in a continuous chunk of memory. I chose to use arrays since they can quickly access elements by their indices, it's a big part for updating and monitoring the position of each segment from the snake's body.

I also included a tracking system for the snake's head position using the List<Point> data structure, in other words an ArrayList. The list records the positions of the snake's head, every time it moves.

Explains a step in the design or development process where you encountered an opportunity and how you used this.

At first when I was deciding on what color to change the game over text I decided on red, as for the score that shows up on top of the game over text I felt like having that also be red was too much, So instead I had this creative idea of just having the score change colors each time you played the game since I couldn't decided on one solid color, I did this by adding g.setColor(newColor(random.nextInt(255),random.nextInt(255),random.nextInt(255)));. Into my code.

Explains a step in the design or development process where you encountered an error and how you resolved this.

A problem I faced was After adding “head Position History” to keep tabs on the snake's head positions, the game started to lag during extended play. This was due to the list expanding endlessly, which ended up using way too much memory.

I fixed this problem by setting a limit to my code, (HISTORY_LIMIT) on the list. Whenever I added a new position, I took out the oldest one if the list got too long.

It's important to prevent long-term performance issues, make sure to keep the size of dynamic data structures in check always.

Explains what you would change or add in the next version of your game or app?

In my next update, I'm excited to make my game feel more lifelike by adding a PNG of a realistic snake and mouse. I think it'll really enhance the experience, moving away from just a black dot and green squares.