Nelson Zeas
13001108

DEMO CONSENT

Pixel Snake

Yes, this is my best attempt at the classic snake game that originated in 1976.
This game supports all basic features such as manual speed, automatic speed, pausing, toggling the grid ON or OFF, and phasing.
Automatic speed means that manual speed will be disabled and the snake's speed will gradually increase as it eats.
"Phasing" means whether the snake dies when it hits a wall or "phases" to the other side of the grid. Like the flash.

FIRST, KNOWN BUGS:

- In order for this game to run smoothly, please run it locally in Google Chrome on full screen, the game lags really bad on repl.it. Even though the UI is made responsive, it's best to play it on a large screen.
  I tested this game on Macbook 2010 running Google Chrome, Version 70.0.3538.110

- Snake theme wise, it would have being super easy to use colors to represent the snake, either circular or squares, but that would have being boring. Instead I also themed the snake with a custom skin and that required thinking or every possible case such as what direction the snake is going, when its turning and in my case when it eats something too.
  Occasionally, the snake might look weird for reasons that I did not figure out but that happens very rarely, usually when the user manipulates the controls in unexpected/unhandled ways.
  The "bubble" that appears after the snake eats something is not a bug, that is to show that whatever the snake just ate is sliding down its body to the tail.

The game is still loaded by the function **setUp(),** like template provided for this project. In the function **setUp(),** the grid is created and the initial snake drawn.

The project has been structured in different files to make it easier to understand. There are three main files, **Game**.**js**, **Grid**.**js** and **Snake**.**js** and each has code that does specific things as the file name suggests.
Game.js is specifically to maintain the state of the game and do things such as new game, pause game, stop game, enable/disable phasing and mainly the loop function.

Grid.js is to create the grid and and functions to change values in the interface.
Snake.js is mainly to draw the snake.
There are also more files.
I have also included two folders named **Food,** and **snakeTheme** which contain images of food and photos to theme the snake.

Drawing the Snake

Surprisingly, this was very easy to implement. I used a Linked List to represent the snake's location in the grid. The first element in the Linked List represents the snake's head location and the last element, the snake's tail location.
I did not use javascripts' ***Linked List***, instead I implemented my own. The ***Node class*** is implemented specifically for the snake, with a **X** and a **Y** variables to store the snake's coordinates in the grid.
The ***Linked List*** also contains two pointers, **head** and **tail** which means that access to the first element and the last can be done in O(1) time.

Rendering the snake

This is done by two main functions defined in the **Snake** class, **drawHead**(coordinates) and **drawTail**().

In the function **drawHead**(coordinates), coordinates is of type Node, which contains **x** and **y** values.
Before drawing the head in the new location specified, the "current" head location is redrawn like the body or a turn or a "bubble" if the snake just ate, then the new head location is drawn at the specified coordinates . This coordinates are also then inserted in the **Linked List** as the first element.

The function **drawTail()** is just as simple. When the snake "eats" something, it slides down the body to the tail. This is always checked first, if something the snake ate raches the tail, nothing is done.
If no food reached the tail, the tail location in the grid is changed to white in order to delete the tail, then the last element is also deleted from the **Linked List.**

To summarize **drawHead(**coordinates**)** and **drawTail(),** every time this functions are called, an extra pixel is added to the head and one pixel from the tail is either removed or not removed depending on whether the snake ate something or not.

Moving the snake

This is done in **Snake.moveSnake()** and the direction in which the snake has to go next is specified in **Snake.currentDirection** which is updated by a different function **setSnakeDirection()**.

In **moveSnake(),** the new coordinates in which the head of the snake has to be drawn is determined first, based on the value of **currentDirection.**

Once the new coordinates are determined, **drawTail()** is called, but before **drawHead(**coordinates**)** is called, function **couldSnakeDie**(coordinates**)** is called to determine whether the snake would die with new coordinates. If the snake dies, this is where the game is ended too.

Looping the game

Initially, I used setinterval() to maintain the snake moving around the grid and it worked fine with manual speed. However, when auto speed was ON, stopping and restarting the setInterval() function with a shorter delay caused the game to "lag" which made the game unplayable. I still used setInterval() but with a delay of **0.**

setInterval() is never stopped, or it's delay changed. It starts as soon as the game is loaded by the browser. The variable **microSecondsCount** is used to manage the snake's speed.

The function setInterval has a delay of 0, but it doesn't loop as fast as a regular **for** loop. It loops at about 20 ms depending on the browser, so this game might behave different on different browsers. For this reason, when the value of **microSecondsCount** is ~230, it is equivalent to ~1 second.

The slowest **speedInterval** set to move the snake is 60 which is equivalent to about 250 milliseconds and the function **Snake.moveSnake()** is called only when the value of **milliSecondsCount** is greater than **speedInterval.**

This also makes it easy to use automatic speed, because instead of stopping the setInterval() function and restarting it with a shorted delay, I can just decrease **speedInterval,** and the snake moves faster with no lag at all.
*The lowest the **speedInterval** value, the fastest the snake moves.
Also, here is where **Snake.currentDirection** is updated with the value from **Snake.newDirection,** which is updated when the user presses on any of the arrow keys from the keyboard.
Snake.currentDirection is updated with the value from **Snake.newDirection** only right before the snake has to move.
This is necessary because **Snake.newDirection** can contain an "illegal" move which can be rejected when **Snake.currentDirection** is being updated.

didSnakeEatItself(coordinates), generateFood(), drawTail()

These functions do exactly what their names imply.
However, in order to do that, this functions need to traverse the linked list and that can become inefficient as the snake eats and the linked list increases length.
To remedy this, I treated the grid as a HashMap, so for **didSnakeEatItself(coordinates),** instead of traversing the list, simply check if the color of the "cell" at coordinates has the substring "**snakeTheme**." If it does, the snake ate itself, otherwise it didn't.
snakeTheme is the name of the folder that contains the images to theme the snake, and the substring **snakeTheme** can only be in cells where the snake is drawn.
Same for **generateFood(),** the minor difference is to check if the cell color is "white", if it is, food can be spawned there, otherwise food cannot be spawned there.
However, by changing the grid color to any other color that is not white, will break this game. The while loop in **generateFood()** will never halt.
Regardless, this functions can be updated to work with any color.

So by treating the grid as a kind of HashMap, **didSnakeEatItself(coordinates),** will always run at O(1) time, **generateFood()** won't always run at O(1) but will be much better than always having to traverse the list to know if it found an empty cell to spawn food.
The function **drawTail(),** cannot run at O(1), most of the time, the last node has to be removed from the linked list, an since it is a singly linked list, it will always

take O(n) time to remove the last node, unless it is a doubly linked list, but unfortunately I don’t have time to complicate things more.

FOOD

The are a total of six photos in the food folder, everytime food needs to spawn, a random photo is chosen and this is done by the function **generateFood().**