CS1006 Practical 1 Tutor: Ruth Letham 7A 20 Feb 2020

Group Member: 190020840 190005624

Overview

We have been asked to implement a simulation of Conway's Game Of Life in Java and have successfully implemented all of the functionalities described in the specification sheet:

1. Creating a functional program that updates alive and dead cells accurately
2. Control the speed at which new generations of live and dead cells are calculated and displayed on screen
3. Implementing a "Step" button which calculates a single generation of live and dead cells
4. Load and Save game states
5. Implementing a "Toroidal" grid

In addition to these prerequisites, we have extended our program to support the following additional features:

1. Changing the rules of Conway's game of Life
2. Customize the speed at which the game keeps calculating new generations during play-time
3. Supporting a new file type (.rle) which happens to be the standard way people share Game of Life files online
4. Clearing all the live cells on the board
5. Changing dimensions of the game board (supports rectangular grids as well)
6. Changing live cell color
7. Changing background color of the grid
8. Recording frames displayed on screen and generating a GIF based on the captured frames
9. A console that displays messages to the user about recent changes and game state

Design

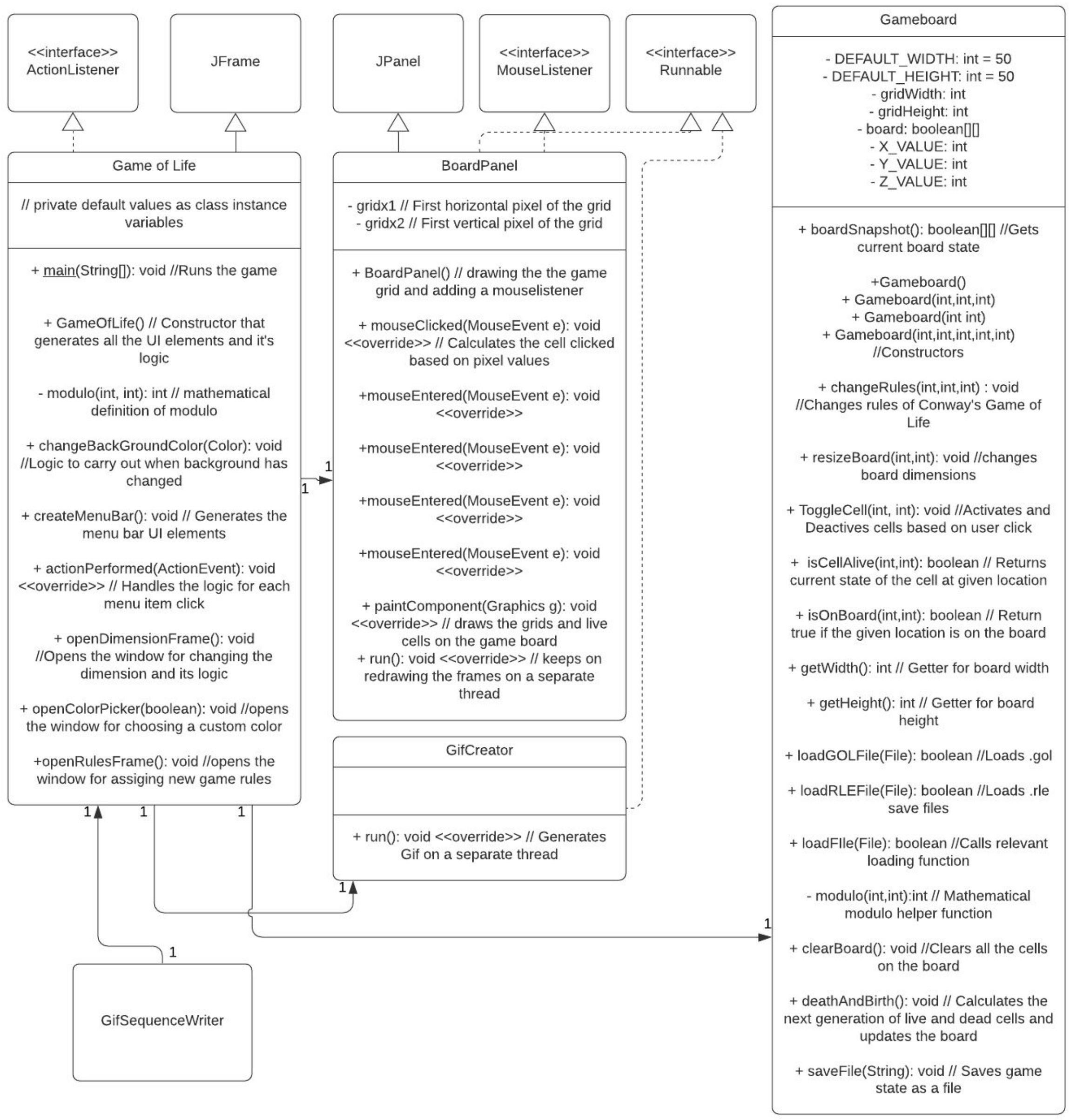
Overall Design

We decided to create the GameOfLife class which handles all the "interactions" with the users and modifies the Gameboard object, which holds the grid and manipulates it. This way, we can separate the game logic from the UI logic. (Check Fig 1 for further insight)

Displaying the Grid

One of the major decisions that had to be made was how to display the grid on the screen. As we were using Java's swing for the GUI, there were two choices. The first of which was to draw buttons to the screen in a grid shape where each cell was represented by a button. The second choice was to use the graphics functionality and draw lines to the screen to represent the grid and then draw rectangles on the grid to represent the alive cells. We went with the second choice as although this required some calculations on which cell is being clicked, it would be less computationally expensive.

Fig 1 (UML Diagram of the Program)

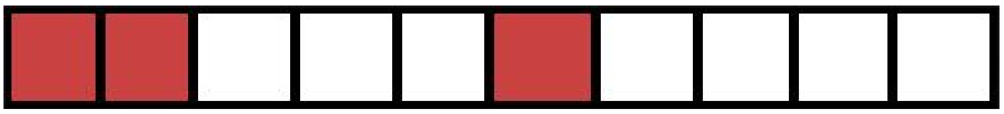

<<interface>> ActionListener
JFrame
JPanel
<<interface>> MouseListener
<<interface>> Runnable
Gameboard
- DEFAULT_WIDTH: int = 50
- DEFAULT_HEIGHT: int = 50
- gridWidth: int
- gridHeight: int
- board: boolean[][]
- X_VALUE: int
- Y_VALUE: int
- Z_VALUE: int
+ boardSnapshot(): boolean[][] //Gets current board state
+Gameboard()
+ Gameboard(int,int,int)
+ Gameboard(int int)
+ Gameboard(int,int,int,int,int)
//Constructors
+ changeRules(int,int,int) : void //Changes rules of Conway's Game of Life
+ resizeBoard(int,int): void //changes board dimensions
+ ToggleCell(int, int): void //Activates and Deactives cells based on user click
+ isCellAlive(int,int): boolean // Returns current state of the cell at given location
+ isOnBoard(int,int): boolean // Return true if the given location is on the board
+ getWidth(): int // Getter for board width
+ getHeight(): int // Getter for board height
+ loadGOLFile(File): boolean //Loads .gol
+ loadRLEFile(File): boolean //Loads .rle save files
+ loadFIle(File): boolean //Calls relevant loading function
- modulo(int,int):int // Mathematical modulo helper function
+ clearBoard(): void //Clears all the cells on the board
+ deathAndBirth(): void // Calculates the next generation of live and dead cells and updates the board
+ saveFile(String): void // Saves game state as a file
Game of Life
// private default values as class instance variables
+ main(String[]): void //Runs the game
+ GameOfLife() // Constructor that generates all the UI elements and it's logic
- modulo(int, int): int // mathematical definition of modulo
+ changeBackGroundColor(Color): void //Logic to carry out when background has changed
+ createMenuBar(): void // Generates the menu bar UI elements
+ actionPerformed(ActionEvent): void <<override>> // Handles the logic for each menu item click
+ openDimensionFrame(): void //Opens the window for changing the dimension and its logic
+ openColorPicker(boolean): void //opens the window for choosing a custom color
+openRulesFrame(): void //opens the window for assiging new game rules
BoardPanel
- gridx1 // First horizontal pixel of the grid
- gridx2 // First vertical pixel of the grid
+ BoardPanel() // drawing the the game grid and adding a mouselistener
+ mouseClicked(MouseEvent e): void <<override>> // Calculates the cell clicked based on pixel values
+mouseEntered(MouseEvent e): void <<override>>
+mouseEntered(MouseEvent e): void <<override>>
+mouseEntered(MouseEvent e): void <<override>>
+mouseEntered(MouseEvent e): void <<override>>
+ paintComponent(Graphics g): void <<override>> // draws the grids and live cells on the game board
+ run(): void <<override>> // keeps on redrawing the frames on a separate thread
GifCreator
+ run(): void <<override>> // Generates Gif on a separate thread
GifSequenceWriter
1
1
1
1
1
1
1
1
1

Supported Files

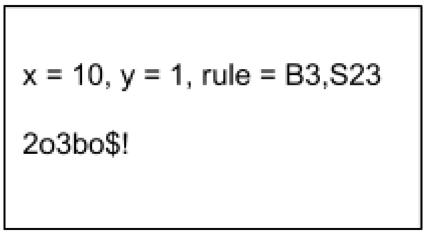
We decided to support a second file type–.rle files–as well as the .gol file type, stated in the specification. This file extension uses the standards defined on https://www.conwaylife.com/wiki/Run_Length_Encoded and has the following format:

1. (Optional) A series of lines beginning with a '#' (which are not necessary to load the board but store information such as who created the file and the name of the file). The '#' character can be followed by different characters to define what information is stored on that line.
2. A line of the form: 'x = <WIDTH>, y = <HEIGHT>, rule = B<Z>,S<X><Y>'
3. A series of lines that are simply filled with the paired values that follow the following format <#OF_CHARACERS><CHARACTER>.
    a. Alive cells are represented by : 'o'

    b. Dead cells are represented by : ‘b’
    c. These pairs are simply placed next to each other with no spaces (Although spaces are dealt with when reading the file).
    d. The ‘$’ character represents the end of line. A run of dead cells at the end of the line do no need to be included
    e. The <#OF_CHARACTERS> can be ignored if there is only a single cell of that state
4. Finally, the end of the file is shown by using ‘!’. Anything after this character is ignored.



For example. the above grid example of width 10 and height 1 would be represented in the following way:



```
x = 10, y = 1, rule = B3,S23

2o3bo$!
```

Game Logic Implementation

Our implementation of the logic of the game changed multiple times. The initial implementation involved storing the value of all neighbouring cells in a different variable. The neighbours were found using try catch blocks to figure out if the cell is outside of the grid and therefore should in fact go to the other side of the grid (toroidal).

We then decided that a much more elegant solution to this would be to simply take the (x_coordinate)%(grid_width) and the (y_coordinate)%(grid_height). This means that if a neighbour is outside of the grid, it will refer to the cell on the other size of the grid. This did not work as we expected, however. After realising that the modulo definition that java uses does not refer to the remainder of Euclidean division, we implemented our own modulo function which will find the value that we needed.

Another change was made so that instead of using variables for all neighbours and calculating the coordinates of each neighbour separately, we instead used two nested for loops which go through the different x and y offsets and simply checks if the cell is alive inside of the loop.

This implementation still meant that the program had to iterate through all cells on the grid which is not necessary. So as well as storing the board state in the 2d boolean array, we also track coordinates of alive cells in an arrayList. These coordinates were stored as Point objects which simply consist of two values: an integer x value and an integer y value. These cells and their neighbours are then added to a Set of cells which need to be checked and then, instead of the method iterating through all of the cells, it only iterates through the relevant cells.

Gif Generation

To generate GIF files, we decided to import a Gif Sequence Writer Class (credits to Elliot Kroo, 2009-04-25, http://elliot.kroo.net/software/java/GifSequenceWriter/GifSequenceWriter.java) that stitches up a sequence of buffered images.

Code Placement Choices

We have decided to place the BoardPanel and GifCreator classes inside the GameOfLife class. This made it extremely easy for us to access the various variables inside the GameOfLife class because we did not have to worry about getters, setters, and static variables, as much.

Grid Board Color

We have decided to automatically calculate the grid color based on the board background to reduce user effort. The colors are calculated using the following equation we derived through trial and error:

$$C_n \equiv (C_0 * 1.5 - 1) mod(255)$$

Where C_0 is the background RGB color components and C_n is the border-line RGB color components

<u>Testing</u>

Whilst testing different board sizes, we found that at a certain point, the board can no longer be displayed correctly. Instead, the the grid is either just black screen or the grid will simply not show as shown below in Figure 2.This is simply a limitation of the resolution of the display in the labs.

Figure 2 - Grid not displaying

Another issue we encountered was during the gif recording where extremely large grids or very long gif clips caused memory heap space issues. For these situations, we figured out that we can run the program with the following command “java -Xmx10g GameOfLife” where the 10g can be replaced with any value which simply represents the size of the memory heap.

Screenshots of the Program

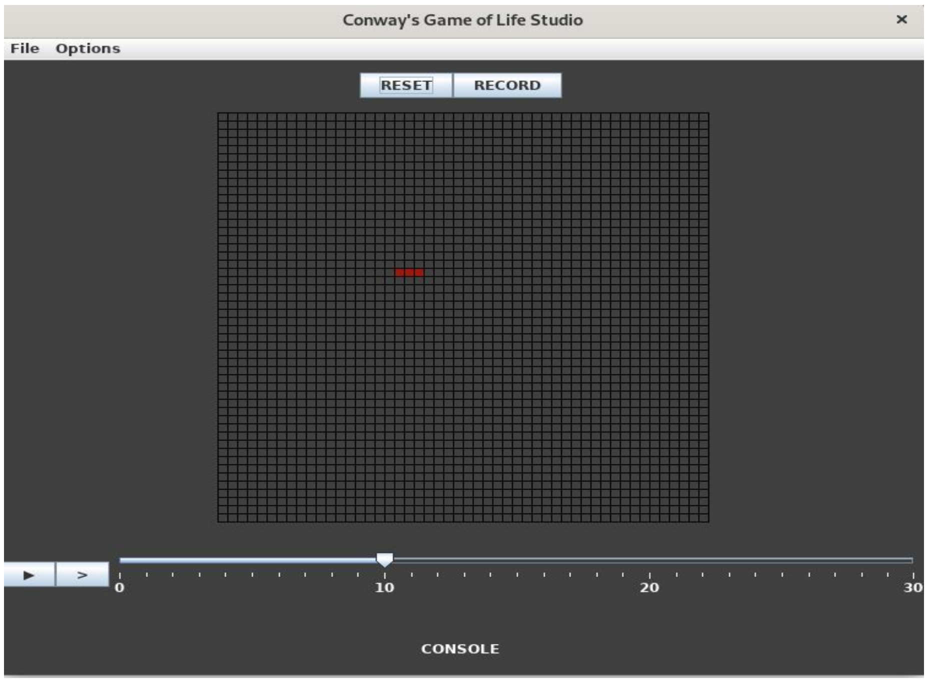
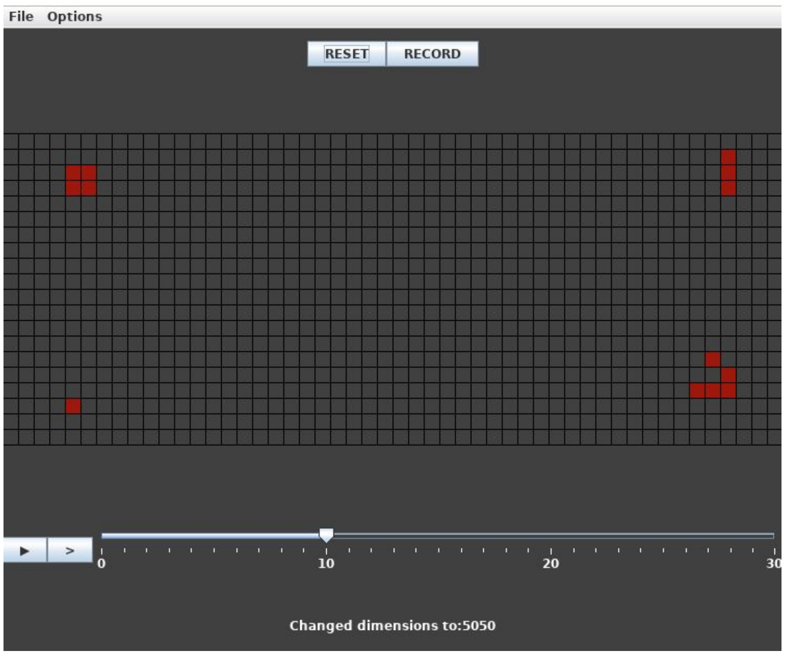
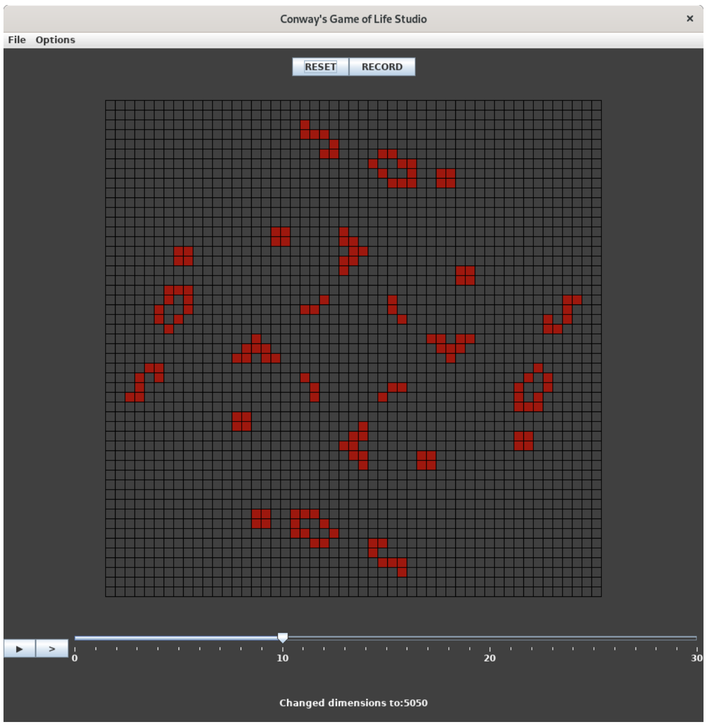


Figure 2 - Oscillator.gol loading correctly



Changed dimensions to:5050

Figure 3 - Saved game board (.gol) loading correctly



Changed dimensions to:5050

Figure 4 - rle loading correctly

Further Testing

Functionality	**Works?**
Updates live and dead cells accurately	✓
Controls the speed of playback	✓
Step Button	✓
Load .rle	✓
Save .rle	✓
Load .gol	✓
Save .gol	✓
Toroidal Grid	✓
Change game rules	✓
Clearing board	✓
Change Dimensions	✓
Supports rectangular grids	✓
Change live colours	✓
Change background colour	✓
Automatic grid line colour	✓
Record frames as gif	✓
Generates gif that matches game colours, framerate, and dimensions (most up to date)	✓
Gif resolution changes dynamically based on grid size	✓
Console to display messages accurately	✓
Console colour updates to match	✓

Conclusion

With more time, we could fix the issue with grid sizes that are too big to be displayed from being able to be entered. Another feature we would have implemented was a cut/copy/paste feature in honour of Larry Tesler.

We have successfully implemented all of the necessary features stated in the specscheet. Furthermore, we have also added extra features, stated above, to make our program more versatile. The hardest part of this practical was figuring out JLayouts that displays our elements correctly. Another thing we struggled with was handling a stack overflow error caused by the recursive nature of a past revision of our program. We successfully overcame this wall by modifying our code to use loops instead.

Mercurial repository: https://maat1.hg.cs.st-andrews.ac.uk/GameOfLife

WORD COUNT: 1424