P2 - Game Of Life Report

Overview

The aim of this practical was to create an implementation of Conway's Game of Life. The program should create a graphical interface where the user can create a board configuration and watch how it is processed according to Conway's rules. The program should also implement edge behaviour through a toroidal form to replicate Conway's infinite grid.

Our program meets all requirements specific in the instructions and also includes some other functionalities that make the process better for the user.

Design

Graphical User Interfaces

For our program we created 4 separate GUIs. IntroGUI is used to either create a new board or load an existing board. This means the user immediately has an option to load instead of always just creating a new board first. NewGameGUI and LoadGameGUI let the user input options to either make or load a game board with their preferred settings. MainGUI displays the game board.

NewGameGUI and LoadGameGUI both create Game objects which form the main basis of the coding manipulation. NewGameGUI gets all the game information that the user can customise: board width and height, and x,y and z values for Conway's rules. When setting game rules we put 1 and 8 as the upper and lower limits on the x, y, z rules as these are the only possible options. For the board size we capped it at 4 < size < 50. LoadGameGUI allows the user to load a previously saved game board state. We used a pattern and matcher to find all of the files within the relevant folder without .class.java.git in their path, and add these to an array. This array is then used to fill a JComboBox so that the user can select one of the files and won't accidentally try interacting with the actual java code (or worst case scenario the actual class we are in!). Like in the NewGameGUI, the user can also set the x,y and z values for Conway's rules, regardless of how the board being loaded was saved.

MainGUI is where the game is actually represented. It has a Board attribute that is set to a board object (explained more later). The board object stores true or false values for each button, depending on whether they are "live" or "dead". When a cell is clicked at any point, first swapBoxColor() and swapIndex() are called which look at the current boolean state of the button and switches the background colour and the boolean state. The swapSet() method of Board is then used to add the grid coordinates of the button to a HashSet that keeps track of all the live cells in an array list. This makes generating the next generation faster (explained later). MainGUI also has a Game object and checks whether the game is being played or is paused using the getPauseState() method of Game. Depending on the result, it sets the text value of a JButton called pauseButton to either "Play" or "Pause" and then changes the play/pause state of Game accordingly. The Pause button has its own class within MainGUI, which uses a new thread to call the gameLoop() method of Game, which continues the process of the game running. The GUI then contains various buttons that allow the user to interact with the game. The Load button calls the load() method that creates a new instance of Game and a new instance of LoadGameGUI. The Randomise button calls

the randomise() method of Game to create a random pattern of live cells on the board. The Save button has its own class within MainGUI that utilises a new thread to create an instance of the SavePopup class, passing in the current board and MainGUI to save the current state of the game. The Step button calls step() which calls the requestStep() method of Game, which executes one update of cells on the board. The Reset button calls reset() and then the reset() method of Game, which sets all of the cells to “dead”. We also included a JSlider so that the user could adjust the speed of play. This utilises the stateChanged() method, which calls the setDelay() method of game, passing in the current value of the slider to set the speed. We used an inverse exponential function meaning at higher speeds the rate of change was less than at lower speeds, giving users more control as speeds got silly high (speed is controlled using a sleep statement at the end of every board generation). We also included the option for the user to change the x,y,z and width/height values of the board. If adjusted, an updateButton is made visible, which when clicked adds all the values of the JComboBox’s to a parameters array, passing this into the update() method which changes the parameter values of Game. NB, the values are only updated once the ‘update’ button is pressed, not when the values are changed: this is so users can change multiple values at once. Then the update button is hidden. Users can interact with the board at any time by clicking on a cell (A JButton in a GridLayout). Being able to both constantly interact with cells using the mouse and having the computer constantly work out the next generation necessitates the use of multithreading.

Game Loop

The game functions by getting the state of the board object, processing this data and then setting the new state. The state is saved to memory in an attribute of the board object; liveSet, which stores the coordinates of all currently live cells. This method of saving the board’s state allows us to save computation resources by only attempting to compute adjacency on all live cells and their 8 adjacent tiles, as no time is wasted checking if large swathes of dead cells need to be updated (our search algorithm also does not repeat searches on the same cell, for instance if two live cells are adjacent). The program achieves the toroidal grid (in updateBoard()) by saving all cells which need to be checked in a set and if any coordinate happens to be over the edge of the grid, that cell’s position is set to the opposite side of the grid. The scanBoard() method is called on each cell, which returns the number of adjacent live cells and saves it in a hashmap along with the coordinates of the current cell. This search method is far, far faster for very large boards where the majority of cells are dead, but is about the same speed for very large boards where the cells are roughly half and half. However, given that boards tend to decrease the number of live cells anyway if they are half or more than half full, there is no disadvantage to using our search method as most boards will tend to being more empty (when using the 2, 3, 3 rules). We also included a check such that if two generations are equal the board will pause automatically as there will be no different future boards.

Other classes

We have a board class which stores attributes of the board and is used in any Game class methods. The most important element of the Board is a boolean 2d array which represents the state of the board. The board also has a live Set of cells which contains the information in a more concise way and speeds up the updateBoard() method.

P2 - Game Of Life Report

Approach to Multithreading

Multithreading occurs in the mainGUI and Game classes. This is how we accomplish pausing: a new thread is created when the play button is pressed for the first time, and ended automatically when it is pressed again. Similarly for saving, another thread governs the Save popup.

Save/Load methods

Our solution implemented the .gol format. We also added the option for comments at the bottom of the file (NB THIS MEANS THE FILES IN EXAMPLES WILL NOT WORK AS THEY DO NOT INCLUDE COMMENTS). When the user does not input a comment the default is set to ‘No comment’. Not having a default would necessitate trying to write code to work out if the last line was a comment on another row of data, and this could lead to lost or added save data if people put in extremely funny row-like comments.

We also implemented our own file save format which on average produces far smaller file sizes, especially in files in which most of the board is dead. The format is three lines:

(width, height)
(encoded data)
(comment)

We encoded the data along one line by imagining the board as strung out line by line onto one big line.

I O O
O I O
I O O

Would become:

I O O O I O I O O

We then encode it by first indicating whether a given cell is full or empty (f/e), then by how many times in a row this occurs. Along with a specified height and width, the data can be stored completely. Hence the above grid is stored as:

3, 3
f1e3f1e1f1e2
This is an example comment

Granted, for small boards the file size is comparable or very slightly larger, but for large and interesting boards where most of the board is dead, the result is far smaller.

For instance for an oscillating clover board:

-rw-rw-r-- 1 tm282 tm282 145 Mar 20 10:21 clover
-rw-rw-r-- 1 tm282 tm282 2566 Mar 20 10:21 clover.gol

Our original save format is much smaller.
The algorithm for encoding and especially decoding these original files is quite lengthy, but the on average smaller file sizes produced makes it a worthwhile investment.
We also implement some little methods to stop any errant file interactions or errors: for instance, the user is not allowed to save if they have no entered a name for the board, or if the resulting save name already exists in the directory. When loading a board the user can also see the associated comment beside the board when they load it.

NB: Saving is only possible when the board is paused. The board can be played again in the background as we use multithreading, but when the save button is clicked again in the save popup, the 'snapshot' of the board that happened when the save button was clicked when the board was paused is the board that is saved.

NB: in our save Popup there are two buttons: one for save as .gol and one for save in our format. Our method does not append anything to the filename but the .gol method does. When the user loads a file, the load method is dependent on whether or not the file has a .gol suffix. This is not an ideal method (we could break it pretty easily but calling a board saved by our method "broken.gol"), but we cannot exactly examine the file contents to determine the type either as someone could break it by creating a file with a similar method by pasting a custom file into the directory. Unfortunately this is a general problem with loading between two types of data.

Examples

Bugs/defects

The only bugs we have identified through testing are related to loading given the multiple types of file the user can choose to save as. These are outlined in the Saving and Loading section.

Evaluation/Conclusion

We believe our code meets all elements of the basic specification and the going further points, including:

- A separate save and load system resulting in smaller files
- Explanatory comments displayed when loading a file
- A way to customise the x y z rules mid-game or pre-game
- A way to customise board size pre-game and mid-game

We also included some of our own touches beyond the specification:

- A much faster updating algorithm for larger and emptier boards
- A randomise button that is especially useful for testing behaviour on fuller boards
- A way to load saved boards mid-game as well as pre-game
- Auto-pausing when the board stabilises to save on computational power