Project 1: Conway’s Game Of Life

Design Analysis
Ido Efrati
6.170 Fall 2013
September 22, 2013

1. Overview:

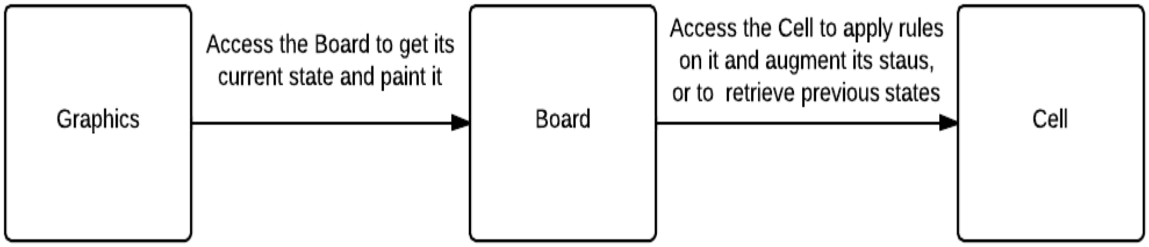
Conway’s Game Of Life phase 1.2 is a browser-based version of John Conway's Game of Life, a cellular automaton that simulates the evolution of an organism using a simple deterministic rule. Phase 1.2 manipulates the document object model (DOM) to create the board, the cells, augment them and paint them.

2. Concepts

In phase1.2 I defined the board as an abstract data type that has functions defined within its scope. These functions have access to all the variables within the board scope. These functions are used to access elements (cells) in the board and perform actions on them. The board is represented by a two dimensional Array, each element within the nested array is another abstract data type (a cell).

A cell like the board also contains closures that give it the ability to keep track of its old, current and next status and change that status as generations’ progress.

Finally, the graphics are also defined as an abstract data type to minimize passing around the array representation of the board.


Graphics
Access the Board to get its current state and paint it
Board
Access the Cell to apply rules on it and augment its staus, or to retrieve previous states
Cell

3. Behavior

3.1 Functionality and features:

In phase 1.2 the game starts with a randomly generated 20x20 board. Dead cells are colored in orange and live cells are colored in blue.

Before a user starts a new game of Conway's game of life he or she can change the size of the board, from a 2x2 to a 25x25, by using a slider. Furthermore the user can click on specific cells and change their state from dead to alive or vice verse. Once the user starts the game, he or she will not be able to change the board anymore.

The game will generate a new generation every 1000ms (1 second) unless the user pauses the game by clicking on the Pause button. Once paused the user can either resume the automatic game by clicking on the Resume button, or manually change the board by clicking on the Step button to move forward or the Reverse button to move backward.

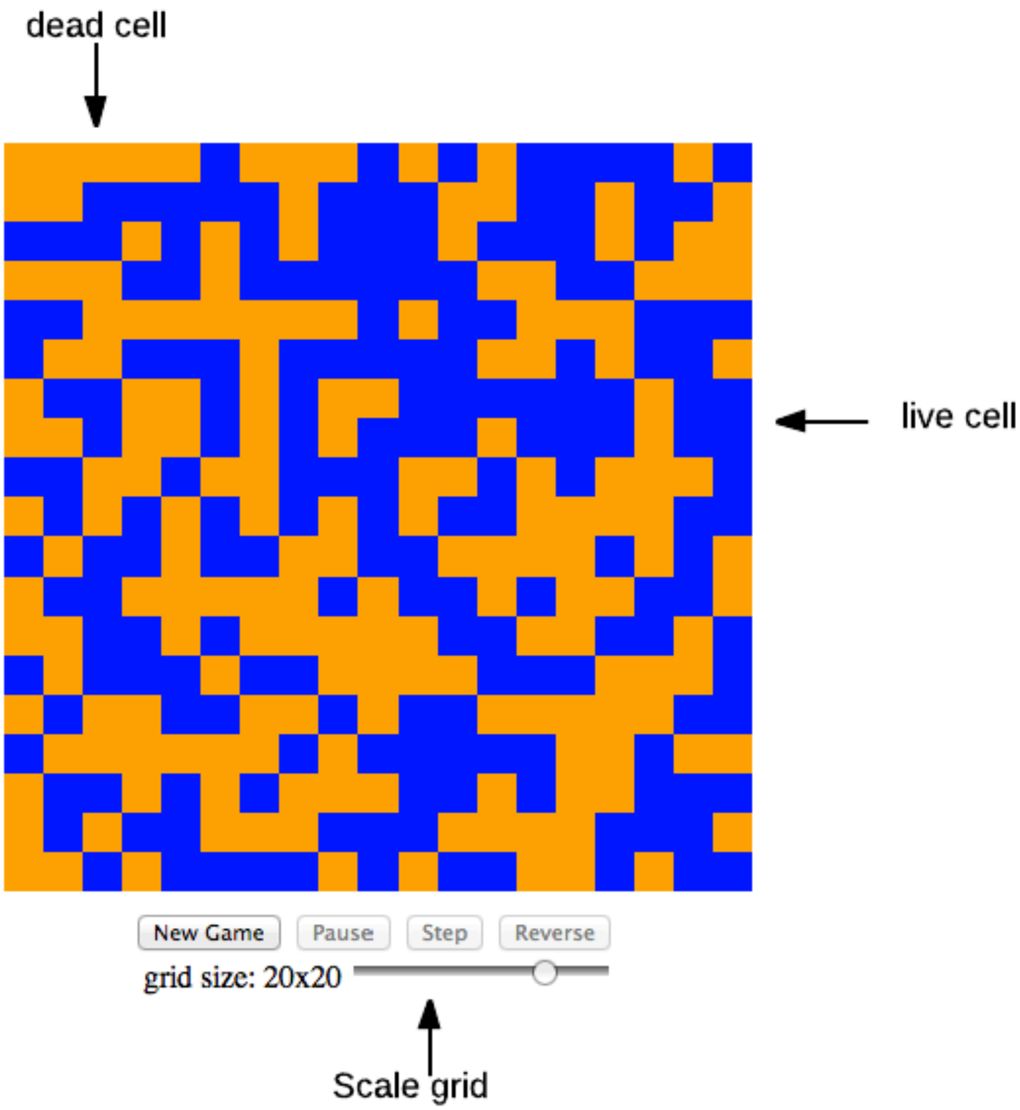
3.2 Security:

A design decision I made to prevent data corruption during the game is to disable the functionality that can augment the board after the game has been started. This design decision grantee that cells data would not change in the middle of a new generation calculation. Thus, the next generation will be accurate.

Another design decision that prevents data corruption is to prevent the Reverse button from performing any action once the board returns to its initial state. If a user clicks on reverse once the board returned to the initial configuration nothing will happen. If I did not implant this decision cells might be defined as "undefined" and the board data would be corrupted by the reverse button functionality

3.3 User interface:

Attached is a picture of the user interface



grid size: 20x20

4. Challenges and design decision

Types: My board, cell and graphics were designed as abstract data types that encapsulate internal states. For each abstract data type I had to develop an API of methods that allowed me to prevent external modification of their values, for example, the cell's state. By using this design I was able to achieve separation of concerns, prevent passing around raw representation.

Alternative design DOM representation: I had two options for DOM representation. The first nested divs, and the second a table. I did not see a an advantage of one over the other, and as a result I decided to go with my first option.

Alternative design relations between cells and divs: each div was assigned a class "dead-cell" or "live-cell" that represented it current status in the DOM. The class colored the Div with the appropriate status color. I had to decide between to options to find the appropriate div that needed to be updated upon creation of a new generation. The two options I had were:

1) To couple a cell object with his div
   a. Advantages:
      i. Quick look up of a div. Each cell has a reference to its div
      ii. No need to write additional functionality
      iii. No need to recreate the DOM after each generation
   b. Disadvantages:
      i. Couple the DOM with an object.
2) To add unique additional attributes (e.g coordinate value) to each div that will be used in JQuery to find the required div.
   a. Advantages:
      i. There is not coupling of DOM elements and cell elements.
   b. Disadvantages:
      i. "Pollution" of the DOM with non useful information
      ii. Requires helper functions to perform calculations to find the right cell and the right div.
      iii. Needs to repaint the board at the DOM level

For the above reasons I decided to go with option number 1. Furthermore, if a user would no longer want to use the DOM, he or she should only do one minimal change to the cell abstract data type (removing the stored reference to the DOM element). Thus, by using option 1 I am still achieving modularity.