Programming Assignment: Game of Life, Animation

Objectives

- Enhancing the UI of Game of Life with simple animation.
- Using the Thread class to control that animation.

Overview

This is the third phase of the Game of Life project. The core task for the first phase was calculating the next generation for a 19x19 Game of Life board. In the second phase, a simple graphical interface was created, minimally a display of the board state, controlled with a Next Generation button. The challenge level for the second phase added controller functionality to the graphical display, click on a cell to toggle its state.

This phase of the project will add animation to the Game of Life.

Background Material

Here are links to the previous write-ups.

- Next Generation
- Additional Notes
- Graphical Interface

This PA assumes a working implementation of both nextGeneration in MyGameOfLife and some JFrame-compatible view of the Game of Life board, probably GameOfLifeBoard.

Work-Around Code

If you are lacking either of these, here is a link to some work-around code.

This page provides:

- Alt2GameOfLife with an alternate version of nextGeneration
- SimpleOutput with an alternate version of a JFrame-compatible view
- a video showing show to use these two components, either separately or jointly
- the code edits that are needed to enable and disable either of these alternate components
- a Java archive with the source code and bytecode, alt2gol.jar

This code is provided so that you can begin work on this PA. Remember, the final phase of the Game of Life project will be evaluated on your own implementations for nextGeneration and a JComponent view. So, it is in your best interest to continue to work to complete those aspects of the Game of Life project.

Minimal Level

There are three task for the Minimal Level:

- Implementing the Observer / Observable design pattern
- Basic Animation, 120 generations per minute
- Thread clean-up

The work for this level is quite straight forward. There are good examples of almost all of the code provided in the examples and notes this week. This is in part to support having the midterm this week.

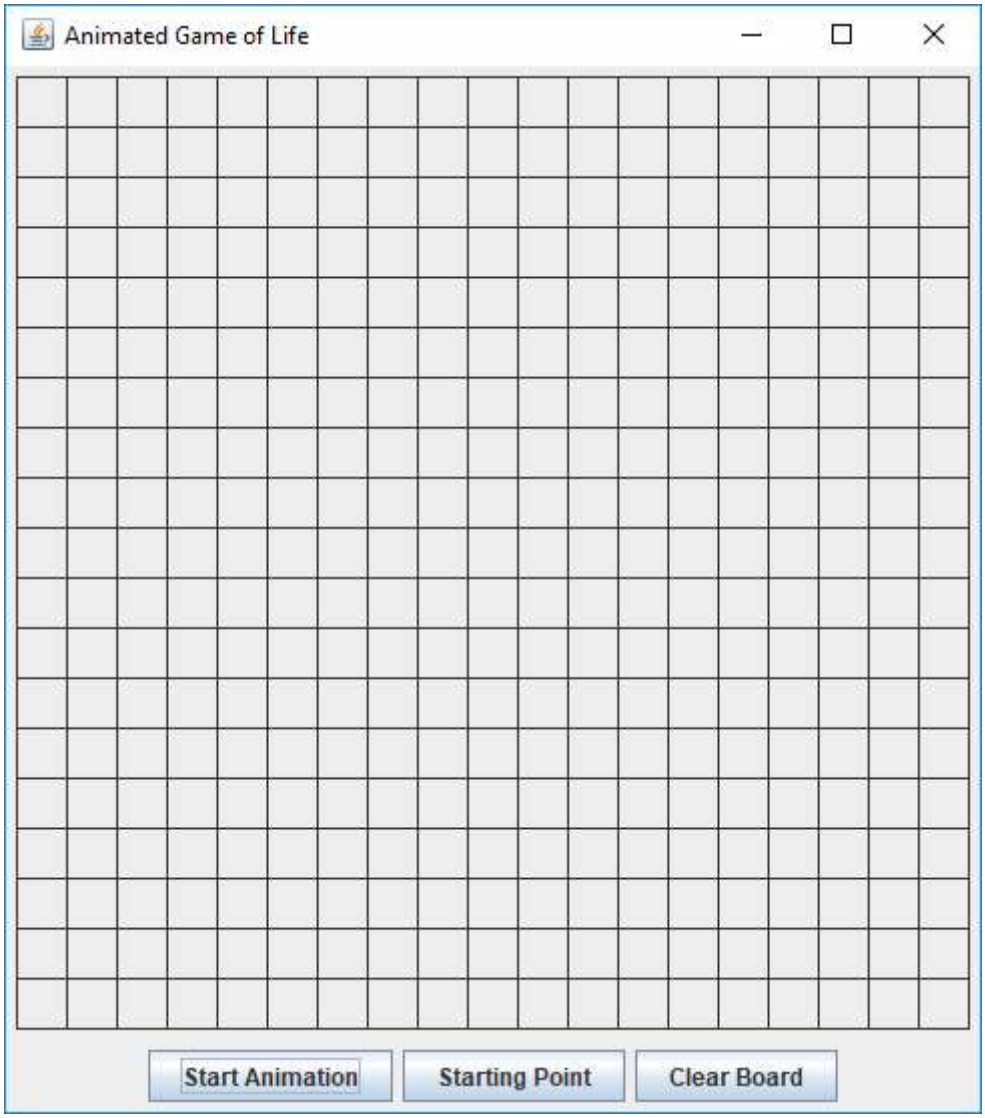
Observer / Observable Design Pattern

As noted in the MVC pdf, the Observer / Observable design pattern is commonly used in MVC applications, particularly when there may be multiple views of a single model. The Observer / Observable design pattern provides a mechanism for the model to notify the views of changes to the state of the system.

In Java, this is implemented by java.util.Observer and java.util.Observable. The model extends the java.util.Observable class. This supports all the machinery needed to notify the observes of change in state. For this project, MyGameOfLife is the model. The view implements the java.util.Observer interface, which contains one method, update. The view registers itself as an observer with the observable calling the addObserver method. This is analogous to event listeners adding themselves to event sources in the Java 1.1 event model, that we have studied.

User Interface Example

This is what the Minimal Level application will look like.



This user interface is essentially the same as the Challenge Level for the Graphical Interface phase. It is not required that the Challenge Level of Graphical interface be implemented, that is, clicking on a cell to toggle its state. The Clear Board functionality is much simpler to implement.

The most important change to the interface is the Next Generation button has been replaced by one with the caption Start Animation. The actions of these buttons is described below.

Basic Animation

There are three user interface controls for the Minimal Level:

Start Animation
Clicking this button starts a thread that will call nextGeneration 120 times a minutes, that is every 500 milliseconds. The text on the button changes to "Stop Animation". Clicking the button again will cause the thread to end and will change the text on the button back to "Start Animation".

Starting Point
Clicking this button will set the configuration of ALIVE and DEAD cells that we have used through out this project.

Clear Board
This button is optional, but useful, as shows in the video example. Clicking this button sets the state of all of the

cells in the board to DEAD.

Video Example

Here is a video example of the Minimal Level for this phase of the project.

Thread Clean-up

As shown in the video, there can be an issue caused by non-terminated threads. For this aspect of the Minimal Level, set the defaultCloseOperation of the JFrame for the application to DISPOSE_ON_CLOSE. Then add a WindowListener to the JFrame object that will end the thread's operation, if it is running. Do not use any deprecated methods of java.lang.Thread.

Use the documentation to determine the appropriate method(s) of WindowListener to override.

Written Report

Write a report in which you describe:

- how did you go about starting this project?
- what works and what doesn't?
- how did you test your code?
- the surprises or problems you encountered while implementing this application
- the most important thing(s) you learned from this assignment
- what you would do differently next time

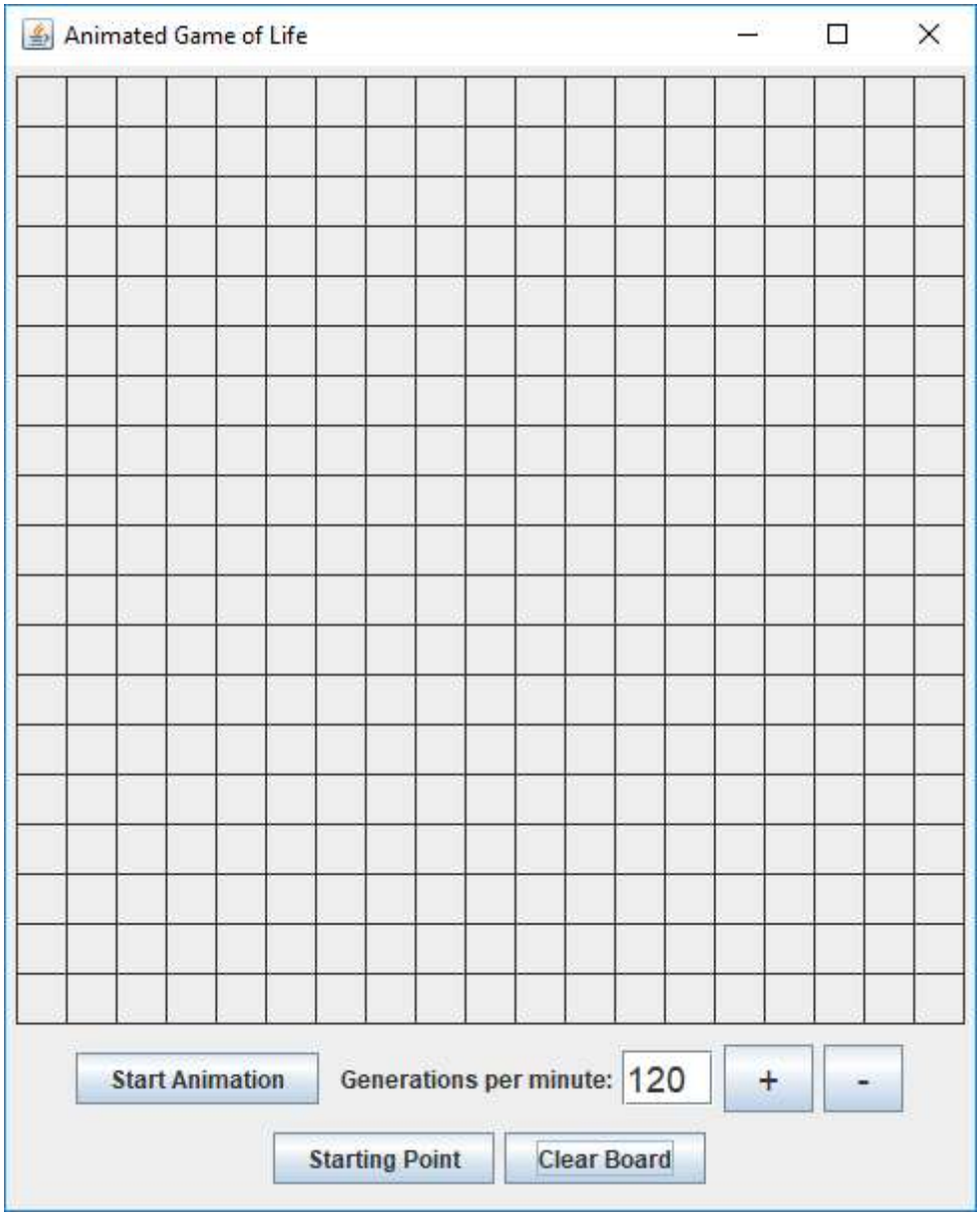
I expect a clear report that has some thought in it. It will be easiest if you take notes about the process as you work on the assignment, rather than waiting till the end to start the written report.

Standard Level

As noted in the video for the Minimal Level, the task for the Standard Level is to give the user the ability to change the speed of the animation.

User Interface Example

This is what the Standard Level application will look like.



This user interface has been expanded to include a text field displaying the number of generations per minute. There are also two additional buttons, to increment and decrement the number of generations per minute. There is also a label for the text field. These additional controls are a required part of this level of the assignment. (The two additional buttons could be combined into a spinner. The implementation is simpler using two separate buttons.)

Because of the larger number of interface elements, they are now arranged in to rows. The implementation shown uses two JPanel instances: one for the controls directly associated with the animation, the top line; and one for the controls that change the state of the model, the second line. You may arrange the controls as you wish.

The font of the JTextField and the two new JButtons instances has been set to 18-point Arial to make those controls a bit more prominent.

```
Font f = new Font("Helvetica", Font.PLAIN, 18);
```

Functionality

The new user interface elements function as follows.

The increment button (+)
Clicking this button increases the number of generations per minute by 20. The maximum value for the number of generations per minute is 500. Attempting to exceed this bound will cause the system beep to sound. Use the beep method of java.awt.Toolkit.

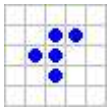
The decrement button (–)
Clicking this button decreases the number of generations per minute by 20. The minimum value for the number of generations per minute is 60. Attempting to exceed this bound will cause the system beep to sound. Use the beep method of java.awt.Toolkit.

The text field
This is an editable control. The user can type a value into it. Pressing the enter key, when the text field has input focus, will generate an ActionEvent. In response to the ActionEvent, the value in the text field shall be read. If it can be converted into an integer value (java.lang.Integer.parseInt) and if the int value is between 60 and 500, inclusive. update the number of generations per minute to the given value. If the user input is not acceptable, either not an integer out of bounds, display an error message. This message can use the intrusive javax.swing.JOptionPane.showMessageDialog. This output mechanism is generally consider too "heavy-handed" for most uses. However, its use may be reasonable here, since the user is almost certainly already interacting with the application using both mouse and keyboard. So, the abruptness of the message dialog may be reasonable.

Video Example

Here is a video example of the Standard Level for this phase of the project.

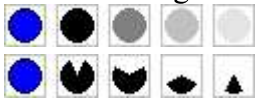
Here is the simple 5-live cell pattern named “R-pentomino” that was used in the video. In the standard implementation, this starting point does not reach a stable point in 1,000 generations. With our limited board size, it does stabilize before that.



Challenge Level

This involves creating small animations for “birth”, “loneliness”, and “overcrowding”.

Here are some samples.

- Birth
  
- Loneliness
  
- Overcrowding
   or

These four-frame animations look reasonable if they take place within a half a second. So, if the time between generations is 1 second (60 generations a minute), the view should remain unchanged for the first half second, then the animation should occur within the second half second. Each of the images will display for 125 ms. If the time between generations is shorter than 500 ms, you can choose to compress the animation (either, shorting the time between steps or omitting one or more steps) or to skip the animation all together.

Implementation Notes

There is the known issue that the javax.swing package is not thread-safe. We can address this issue by only using one thread to do the rendering, using repaint method to trigger the rendering.

Note, the animation of these transitions is essential an enhancement to the nextGeneration method of the model. Most of the processing will occur within the model. To control the speed of the transition animation and whether it should occur or not, use package private resources in the model that can be accessed by GameOfLifeAnimation, among these, which step in the transition animation is current.

Use javax.swing.Timer to prompt the repaint for the animation. This gives more flexibility than Thread.sleep. Still use Threads for the primary timing, the calls to nextGeneration.

This animation will need to distinguish between the different types of transition in cell state. This was mentioned in the Next Generation PA. as "Approach 2". MyGameOfLife can declare additional constants for the transition states of BIRTH, LONELINESS, and OVERCROWDING. These constants could be package private, since there are solely for internal use.

The actions of "Approach 2" will need to be split into to separate steps. First the transition states will need to be marked. Since this is internal to the application, the setCellState input validation can be side-stepped by manipulating the the array values directly. Now, the transitional states and the current "step" are available to the view for rendering the animation.

At the end of the animation, that is, at the fourth step, the cell states shall transition from BIRTH to ALIVE, and LONELINESS and OVERCROWDING to DEAD. This is the second part of the original nextGeneration code.

Submission

Two files:

- One (1) Java archive file, with source code and bytecode for the Game of Life project. Set the entry point of the archive to GameOfLifeAnimation. (.jar)
- One (1) ASCII, plain-text report file (.txt)

Grading Rubric

Functionality: 8 points
Minimal Level - 5 points
- MyGameOfLife Is-A java.lang.Observable, GameOfLifeBoard Is-A java.lang.Observer, the appropriate connection between observer and observable is established
- GameOfLifeAnimation supports Thread-based animation of nextGeneration at 120 generations per minute
- JFrame is set with DISPOSE_ON_CLOSE and closing the window while the thread is still running will terminate the application in a controlled mannter

Standard Level - 8 points
- Minimal Level functionality
- user control of the speed of the animation, between 60 and 500 generations per minute, using either the increment and decrement buttons (delta of 20 generations per minute) or typing directly into the text field and pressing Enter
- report of unacceptable values.

Challenge Level - 10 points
- Standard Level functionality
- mini-animations of the transition states: BIRTH, LONELINESS, and OVERCROWDING

Style: 3 points
- conforms to Homework Guidelines regarding style

Documentation: 4 points
- JavaDoc comments on all public and protected elements
- a separate comment on each class, field, method (can be JavaDoc comment)
- JavaDoc comments generate no warning or error messages
- each JavaDoc comment include a brief description in addition to required tags
- method / constructor internal comments to explain functionality
- report addresses the questions posed.

Back