Sean Holdsworth

Conway’s Game of Life

Introduction

This project implements Conway’s game of life, illustrating a simple cellular automaton. The application presents a simple display of a square grid of cells with a selection of controls below it. The number of cells can be controlled as can the number of cell generations per second. The initial layout of cells is random with any individual grid cell having a 20% chance of being “alive”. The game can be restarted at any time by pressing the “Restart” button.

The UI can also be resized so as to make the grid larger. When this is done the grid must preserve a square aspect ratio. Implementing this is really the only part of the design that raises an interesting technical challenge.

Assumptions

I’m assuming that starting the game in a different random state is acceptable and that changing the number of cells in the grid constitutes a restart. It would be a relatively simple change to allow mouse clicks to change the state of individual cells but that goes against the spirit of the original concept of the game.

I’m also assuming that the game should be allowed to continue indefinitely. It would be possible to detect stable or repeating states relatively easily since end states typically have a cycle of only two states and states can be represented efficiently using either a bit array or run length encoding.

The problem description mentions that the display should use two image buffers in addition to two logical buffers. In practice explicit display double-buffering is not necessary for smooth animation because of the way JPanel’s paintComponent method is implemented and integrated with the Swing Timer. Therefore in effect this functionality is provided by default.

Notes on UI

The top level JFrame is a super class of MainFrame. MainFrame then creates instances of two view components: the GridPanel used to display the cells and the ControlPanel holding the control components at the bottom of the UI. All layout is done programmatically using the GridBagLayout class of Swing.

A key part of the implementation of this application is the concept of cell generations. To implement this in such a way as to be able to control that the rate at which new generations are created requires use of a timer. The use of a low level asynchronous timer, asynchronous that is with respect to the main UI event loop, could potentially cause major problems. Fortunately however, Swing provides its own Timer class that integrates with the main event dispatch thread.

The cells of the game are drawn directly onto the JPanel component from which the GridPanel component is derived. This is done using an overridden implementation of JPanel’s paintComponent method. Since Swing graphics deals with integer multiples of pixel sizes, an intermediate layer using Swing’s Graphics2D class is used. This allows for cell sizes to be regarded as fractional pixels which can then be anti-aliased and rendered using the underlying integer pixel model.

The most interesting part of the implementation is the need to preserve the aspect ratio of the JPanel of the GridPanel component. Since this is a constraint that is required only when the application JFrame is resized, it is the responsibility of MainFrame to handle this. To this end the inherited componentResize method of MainFrame is overridden and called when the resize triggers a ComponentEvent.

Table of Classes, Interfaces and Files

So as not to have to deal with more source files than you have to when writing and reviewing code I've placed utility classes and interfaces in the same Java source file as the main class that makes use of them. For ease of navigation though, here is a list of classes and the files in which they can be found.

Class / Interface	Source File
class LifeApp (main)	LifeApp.java
class MainFrame (controller)	MainFrame.java
class GridPanel (view)	GridPanel.java
class ControlPanel (view)	ControlPanel.java
interface RestartListener	ControlPanel.java
interface SpeedChangeListener	ControlPanel.java
interface SizeChangeListener	ControlPanel.java
class Life (logic)	Life.java

Tools & Techniques

The application follows the pattern of similar UI focussed applications with the MainFrame class acting as a controller and allowing loose linkage of the view component classes. The logic of the game is encapsulated in the Life class and the GridPanel provides a graphical representation of the underlying data structures of that class.

The Life class maintains two integer arrays representing the current and previous generation of cells of the cellular automaton. The value of 0 represents a dead cell and 1 a live cell. Before the simulation starts for a given size of grid, another data structure is built which is an ArrayList of ArrayLists of Integers Each cell has an offset and this offset is used as an index into this structure where it finds the offsets of all its neighbor cells.

Given this precalculated array of neighbor cell offsets and the fact that dead and live cells have values 0 and 1 respectively, it is easy to calculate the number of live neighbors and to determine the state of the cell in the next generation with a simple lookup.

```
public void createNextGen() {
    int nextState[][] = {
        {0, 0, 0, 1, 0, 0, 0, 0, 0},
        {0, 0, 1, 1, 0, 0, 0, 0, 0}};

    for (int i = 0; i < nCells; i++) {
        int nNeighbors = 0;

        for (int offset : neighborsByCell.get(i))
            nNeighbors += thisGen[offset];

        nextGen[i] = nextState[thisGen[i]][nNeighbors];
    }

    int temp[] = nextGen;
    nextGen = thisGen;
    thisGen = temp;
}
```

The GridPanel class has the responsibility of taking the logical representation of the grid of cells internal to the Life class and displaying it on screen. This code remains largely unchanged in the subsequent applications so is worth discussing here. Everything is driven by an instance of the Swing Timer class. Modifying the rate at which this generates events allows control over screen refresh rates. This code shows the Timer object being created in the GridPanel constructor and an event handler being attached to it.

```
    tm = new Timer(initialTimerDelay, new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent event) {
            life.createNextGen();
            repaint();
        }
    });

    tm.start();
```

When the timer triggers the next generation of cells is calculated and then repaint is called which will queue a call to the overridden version of JPanel’s paintComponent method.

```
@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);    // Draws background

    Graphics2D g2d = (Graphics2D) g;

    int nCells = cellsPerSide * cellsPerSide;
    int panelWidth = Math.min(getWidth(), getHeight());
    double cellSize = (double) panelWidth / cellsPerSide;
    double cellWidth = cellSize - 2.0;
    double xOffset = 1.0, yOffset = 1.0;
    int col = 0;

    int[] cells = life.getCells();

    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);

    for (int i = 0; i < nCells; i++) {
        g2d.setColor((cells[i] == 0) ? Color.WHITE : Color.BLACK);

        g2d.fill(new Rectangle2D.Double(
            xOffset, yOffset, cellWidth, cellWidth));

        if (++col == cellsPerSide) {
            col = 0;
            xOffset = 1.0;
            yOffset += cellSize;
        }
        else
            xOffset += cellSize;
    }
}
```

Sean Holdsworth

Critique

I was surprised by a peculiar limitation of Java. In creating the neighborsByCell structure. Since the number of cells is known at the time this needs to be initialized, I would have like to use a simple array of type `ArrayList<Integer>[]`, rather than having to use an ArrayList container of type `ArrayList<ArrayList<Integer>>`. Unfortunately if you try to use a simple array this will trigger a “generic array creation” error.

One thing I like about this problem is that the logic of generating the next generation could be reduced to simple table lookups. This allows successive generations of cells to be produced at a fast rate without overloading the processor.

On the same theme of efficiency however this problem lays bare performance problems in the X Windows implementation of the Java graphics environment including Swing and AWT. The applications allows for 50 by 50 grids of cells being refreshed at 40 generations per second. This is easily achieved on a notebook running MacOS and takes about 10% of the processing power of a single core. On Linux this rate is not achievable and causes 90% of processor resource to be used by the X Windows process.