6G4Z1101 Programming

Independent Study Project – The Game of Life

Huw Lloyd, Manchester Metropolitan University.

The 'Game of Life' (GOL) is a *cellular automaton* first described by John Horton Conway. The system evolves a two-dimensional grid of cells, each of which can be in one of two states ('alive' or 'dead') by applying a simple of rules. The new state of a cell after an update depends only on the number of live cells amongst its immediate neighbours at the current time. Despite the simplicity of the rules, rich and complex behaviour emerges in the system. The Wikipedia article on GOL (https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life) carries a good description, along with examples of some of the structures that emerge.

Most programmers code up the Game of Life at some point in their careers. If you haven't done it already, now is a good time to try. In this project, we will use the Processing classes to handle the graphics for an implementation of GOL. The aim is to first produce a GOL evolution of an initially random field, and then implement the 'Gosper Glider Gun' – a configuration that spits out 'gliders' every thirty updates. Gliders are stable structures that move diagonally across the grid.

1. The Game of Life

The Game of Life is based on a two-dimensional grid of cells in which each cell can be in one of two states – alive or dead. The grid is updated in steps. In one step, each cell is updated based on how many of its neighbours (up to eight for cells in the middle of the grid, less on the edges) are alive. The rules are:

1. Live Cells – a live cell stays alive if it has 2 or 3 live neighbours, otherwise it dies.

2. Dead Cells – if a dead cell has exactly 3 live neighbours it is brought to life, otherwise it remains dead.

Note that all the cells have to be updated simultaneously. In practice this can mean keeping two copies of the grid – one to store the results of the update step, and one for the current state. This is so that we use the same value for a cell throughout a timestep as we loop over the grid.

The automaton is usually represented as a grid of square cells, black for live and white for dead (see the Wikipedia page for examples).

2. Preliminaries

First, set up a Java project in Eclipse. Import the processing classes (**core.jar**) and add to the build path. Refer to week 10 on Moodle for details if you have forgotten how to do this.

We will do everything in one class, which extends **PApplet**.

- Add a class to the project, called **Life**
- Extend the class from **PApplet**, remembering to add **import processing.core.*;** at the top of the file.
- Add a constructor, and empty **void setup()** and **void draw()** methods.
- Run as an applet to check everything works so far (you should see an empty applet window appear).

3. Stage 1 - Data

The first thing to do is to set up the data. We will simulate a square area of 64 x 64 cells. The obvious way to represent this in the code is with a 2D array. Given that the cells can only be in one of two states, we will use an array of **boolean**. We use a constant (**final int**) to set the size of the array. Here's the data, and the constructor:

```
final int boardSize = 64;
boolean[][] board;

public Life()
{
	board = new boolean[boardSize][boardSize];
}
```

We will fill this data with random values.

- Add a method **void FillRandom()**
- Call it from **setup()**
- Implement the method so that it randomly assigns **true** or **false** to each cell in **board**.

(**Hint**: use nested for loops to access each element of the array, and use **Math.random(),** which returns a **double** between 0.0 and 1.0. Think about how you could use that random number to decide between setting the value to **true** or **false** with equal probability.

At this point, your class should look like this:

```
import processing.core.*;

public class Life extends PApplet
{
	final int boardSize = 64;
	boolean[][] board;

	public Life()
	{
		board = new boolean[boardSize][boardSize];
	}

	void FillRandom()
	{
		for ( int i = 0; i < boardSize; i++ )
		{
			for ( int j = 0; j < boardSize; j++ )
			{
				// TODO - set board[i][j] here.
			}
		}
	}
	public void setup()
	{
		FillRandom();
	}

	public void draw()
	{
	}
}
```

4. Stage 2 – draw() method

Now we have the data in place, we can implement the draw method. The first thing to think about is how to represent each cell. A single pixel would be too small (a 64 x 64 window would be tiny on the lab Mac 4K displays) so we will need to represent each cell with a square block of pixels. Processing's **rect** method is a good way to do this.

First add some constants – for the number of pixels we will use to represent a cell, and the screen width and height (which are derived from the board size, and the pixel size of a cell).

- Add the following code after the definition of **boardSize:**

```
final int blockSizePixels = 16; // tweak this to suit your display
final int screenWidth = boardSize * blockSizePixels;
final int screenHeight = screenWidth;
```

- Now set the screen size in the **setup** method:

```
size( screenWidth, screenHeight );
```

- Finally, implement the **draw** method. See the hints below.

Hints:

1. Clear the background to white before drawing.
2. Use nested **for** loops to iterate over the 2D array.
3. If a cell is alive (true), draw a black rectangle.
4. The coordinates of the top corner of the rectangle for cell **i, j** are **i*blockSizePixels, j*blockSizePixels.**
5. The width and height of each rectangle are both equal to **blockSizePixels.**

You should see something like this:

4. Stage 3 – Updating the Automaton.

Now for the update step. We will need a second, temporary, 2D array for the step results. Declare this in the class and initialize in the constructor – call it **tempArray.** The constructor now looks like:

```
boolean[][] board;
boolean[][] tempArray;

public Life()
{
	board = new boolean[boardSize][boardSize];
	tempArray = new boolean[boardSize][boardSize];
}
```

Now for the update method itself. I suggest you make a function

int countNeighbours(int i, int j)

which returns the number of live neighbours to cell i, j (this should look at the **board** array). Then you can update **tempArray** as follows.

```
public void update()
{
	// calculate new solution from old
	for ( int i = 0; i < boardSize; i++ )
	{
		for ( int j = 0; j < boardSize; j++ )
		{
			int neighbours = countNeighbours( i, j );
			// live cells
			if ( board[i][j] )
			{
				if ( neighbours < 2 || neighbours > 3 )
					tempArray[i][j] = false;
				else
					tempArray[i][j] = true;
			}
			else // dead cells
			{
				if ( neighbours == 3 )
					tempArray[i][j] = true;
				else
					tempArray[i][j] = false;
			}
		}
	}
	// swap old and new
	boolean[][] swap;
	swap = board;
	board = tempArray;
	tempArray = swap;
}
```

Notice how we swap the arrays at the end – we could naively copy them element by element, but we can do it much more efficiently - the variables **tempArray** and **board** are *references* to the arrays; we can swap what the references point to without moving any of the data.

- Your task is to write the **countNeighbours** method. Be careful to deal with the boundaries properly – for example, if a cell has x coordinate 0, it has no neighbours to its left (x = -1, which is out of bounds for the array). You might find this easier if you write another function **boolean getCell(int i, int j)** which returns the value of the cell at i, j, taking into account the boundaries – if i or j are out of bounds, this would return false. Try doing it without **getCell**, and you will probably find you end up with a mess of nested conditional statements which deal with the boundaries.
- Add a call to **update** in the **draw** method.

You should now have a fully functioning Life simulation. You should see the field evolve for a while, with a few gliders and spaceships flying around, eventually settling down to a collection of static and oscillating patterns – probably mostly blocks, beehives and blinkers, although you will occasionally see some more exotic things.

5. Stage 4 – Gosper's Glider Gun

Use the following code snippet to create a Gosper's Glider Gun:

```
String gun[] =
{
        "........................0...........",
        "......................0.0...........",
        "............00......00............00",
        "...........0...0....00............00",
        "00........0.....0...00..............",
        "00........0...0.00....0.0...........",
        "..........0.....0.......0...........",
        "...........0...0....................",
        "............00......................"
};
```

Here's how I suggest you do it:

- Write a method **FillGliderGun(int i0, int j0)**. This will fill the field with dead cells, and add a Glider Gun pattern offset from the top corner by i0, j0. Call this instead of **FillRandom.**
- The string array represents the Glider Gun pattern. A dot is a dead cell, a 0 is a live cell. Use two nested **for** loops, and the methods of the **String** class to decode the pattern and fill in the board.
- This pattern was taken from the Life Lexicon (see Further Reading). You can copy and paste other patterns (which are in the same format) and try those out, too.

6. Extending the project

Here are a few suggestions for extending the project

1. Pausing the simulation on a key press (e.g.'P'). This could be a toggle (so that P also sets the simulation running again).
2. In pause mode, another key (e.g. 'S') could step the simulation (a 'single step' mode).

3. Editing – in pause mode, clicking a cell could toggle its state (dead->alive, or alive->dead). This would be a useful feature for experimentation.
4. Different sized boards, possibly set in command line arguments.
5. Toroidal simulation – the board wraps around at the boundaries. You could do this by modifying **getCell** (if you used it) so that cell -1 looks up cell boardSize-1, and cell boardSize looks up cell 0 (in either x or y). A Gosper Glider Gun becomes self-destroying in a toroidal game – its own gliders sneak up behind it and blow it apart.
6. Improve the display. A grid of lines (probably in a light grey) drawn over the cells would be helpful, especially when editing. Showing a generation count as a text string is another possible improvement.

7. Further Reading

http://www.ericweisstein.com/encyclopedias/life/ - Eric Weisstein's 'Treasure Trove of the Life CA'.

http://www.conwaylife.com/ref/lexicon/lex.htm - the Life Lexicon. Huge collection of Life objects, terminology and lore. Initial states are shown as ASCII pictures, in the format we used for the Glider Gun.