**Java Project | Game of life**

Aim: This project is all about investigating cellular automata, and in particular, an algorithm called Game of Life.

GOL or Conway’s game of life is a cellular automaton devised in 1970 by John Conway.

Here is an explanation of the automaton algorithm from Rosetta code.

“A cell   **C**   is represented by a   **1**   when alive,   or   **0**   when dead,   in an   m-by-m   square array of cells.

We calculate   **N**   - the sum of live cells in C's   [eight-location neighbourhood](http://en.wikipedia.org/wiki/Moore_neighborhood),   then cell   C   is alive or dead in the next generation based on the following table.

|  |  |  |
| --- | --- | --- |
| C | N | New C |
| 1 | 0,1 | -> 0 # Lonely |
| 1 | 4,5,6,7,8 | -> 0 # Overcrowded |
| 1 | 2,3 | -> 1 # Lives |
| 0 | 3 | -> 1 # It takes three to give birth! |
| 0 | 0,1,2,4,5,6,7,8 | -> 0 # Barren |

“

Source - <http://rosettacode.org/wiki/Conway%27s_Game_of_Life> , accessed at 3:09 PM 17/08/2016

English rules for the game of life

“

* Any live cell with fewer than two live neighbours dies, as if caused by under-population.
* Any live cell with two or three live neighbours lives on to the next generation.
* Any live cell with more than three live neighbours dies, as if by over-population.
* Any dead cell with exactly three live neighbours becomes a live cell, as if by reproductions

"

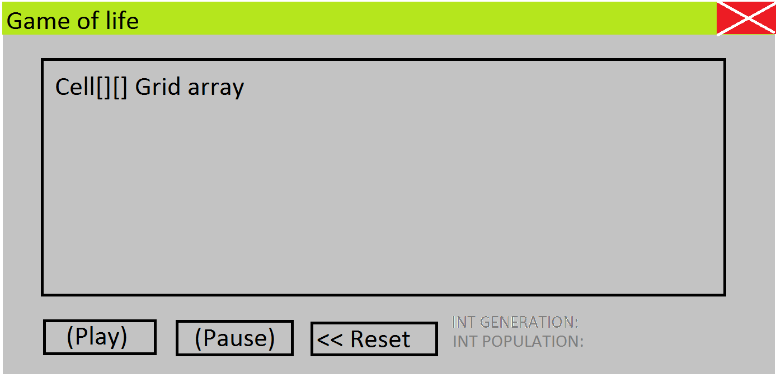
Source - <https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life> accessed 3:20 PM 17/08/2016

***In early development***

After spending some time thinking and playing with ideas, I realised the algorithm could be implemented in java using a 2D Boolean array, which would contain the state for each cell.

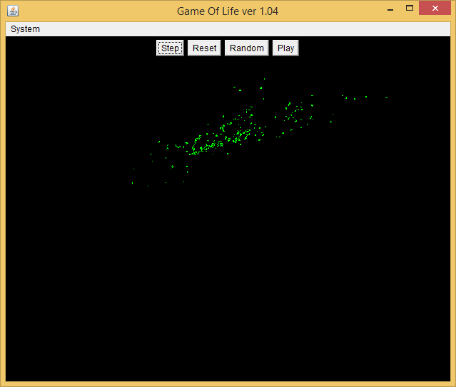
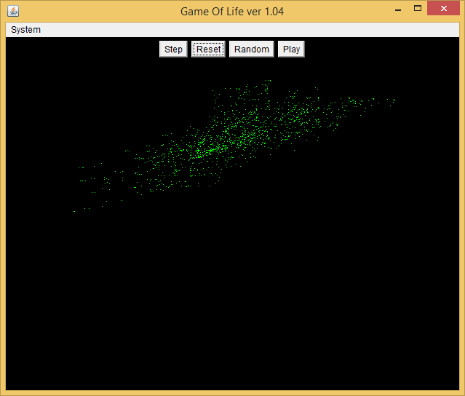
My original design had a more sophisticated window than it does now.

**GUI Design:**



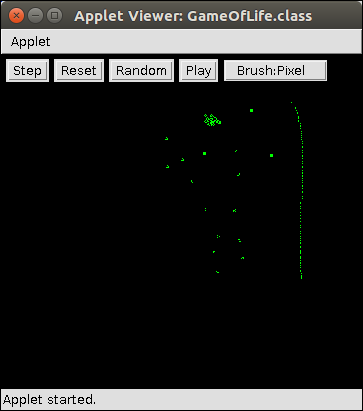
During implementation it got boiled down for simplicity.

**Early development screenshots:**



The current GUI design involves rendering the Game of Life in the Applets Container, along with the buttons over the top, to the left of the window using FlowLayout manager.

***Later in development***



This is not the best design for the GUI, but it is simple enough to be useful and allowed me to focus on the main problems, which was the algorithm implementation and rendering.

**Basic Algorithm of Game of Life in Java**

In my current implementation, there is a 2D array of true or false Boolean states cellAlive [x][y]. This Array is of size ***X \* Y,*** where x and y is the size of a grid of cells drawn to the screen as single pixels.True represents the alive state, and false is the dead state of a cell.

Stepping into the next state is done by:

1. Allocating a 2D integer array ***neighbours*** , size [***X]\*[Y]***
2. Iterating through every cell and incrementing ***neighbours[x][y]*** when a neighbour is alive.
3. Calculating the next state by iterating through every cell and applying the rules of game of life.  
   Which are:  
   * if cellAlive[x][y] is true and neighbours[x][y] is less than 2 or more than 3 then cellAlive[x][y] = false
   * If cellAlive[x][y] is true and (neighbours[x][y] equals 2 or 3) then cellAlive[x][y] = true
   * If cellAlive[x][y] is false and neighbours[x][y] equals 3 then cellAlive[x][y] = true
4. Then the state is rendered onscreen by repainting.

The following page contains initially when notation. This notation is representative of the current implementation, and not the early implementation, which was wrong.

Initially

boolean cellState [] [] = new boolean [screenWidth] [screenHeight];

boolean neighbours [] [] = new boolean [screenWidth] [screenHeight];

When button "step" is clicked

// Count neighbours

for int x = 1; x < screenWidth-1; x++;

for y = 1; y < screenHeight-1; y++;

int n = 0;

if cellState [x - 1] [y - 1]

n += 1;

if cellState [x - 1] [y]

n += 1;

if cellState [x - 1] [y + 1]

n += 1;

if cellState [x][y - 1]

n += 1;

if cellState [x][y + 1]

n += 1;

if cellState [x + 1][y - 1]

n += 1;

if cellState [x + 1][y]

n += 1;

if cellState [x + 1][y + 1])

n += 1;

neighbours[x][y] = n;

// Apply game of life algorithm

for int x = 1; x < screenWidth-1; x++;

for y = 1; y < screenHeight-1; y++;

if (cellState[x][y]) AND (neighbours[x][y] < 2) OR (neighbours[x][y] > 3)

cellState[x][y] = false;

else if cellState[x][y] AND (neighbours[x][y] == 3 OR neighbours[x][y] == 2)

cellState[x][y] = true;

else if NOT cellState[x][y] AND neighbours[x][y] == 3

cellState[x][y] = true;

draw();

***First testing – stepping through code***

Although I now fully understand the algorithm, in practice I had some trouble understanding how to implement it correctly because of a few mistakes and misunderstandings.

The algorithm was not working, and I knew this because I was comparing known cell structures to the ones in my implementation, and they weren't acting the same. I spent some time setting breakpoints in the game of life algorithm code, stepping through the code and realised a few problems.

The first mistake was that I tried to count the neighbours of a cell and calculate the next state in the same loop. This was problematic because the state of the grid was being changed every time a cell's neighbour was being counted. I tried creating a copy of the array as a fix, but I just accidentally made a pointer to the array and the problem was still there. The solution was to count the neighbours and calculate the next state in completely separate loops. The algorithm was finally working.

The GUI uses mouse listeners to facilitate user drawing. Left click for draw, right click for erase, and I figured out along the way that I needed bounds checking so that it cannot go past the array bounds. My initial implementation of bounds checking used immediate values and only worked statically, so when I tried to resize the rendering window, it broke the code. This taught me a valuable lessen. Write dynamic code! Later on I added in brushes, which had a size limit of a cell and its neighbours and were written not very dynamically, but its better than just being able to draw one cell at a time.

The algorithm itself ignores every cell at the edges, to avoid out of bounds errors, and there is also, sadly, no wrap around. Edge cells are not displayed on screen and are always dead.

Rendering just draws a black background using a rectangle, and then draws on to a double buffer (a buffer that is not seen by the user until all drawing of the frame is complete) and a green pixel for every living cell. This is then drawn to screen using Graphics.drawImage(buffer). Very early in development I had problems with a flickering screen. Tracking down the culprit took a lot of research online, but I came across a forum talking about how this is the update methods fault. The update() method Is one of the methods called when the java program calls repaint(). Update actually clears the screen before calling paint, so the fix I implemented was to override update and simply make it only call the paint method.

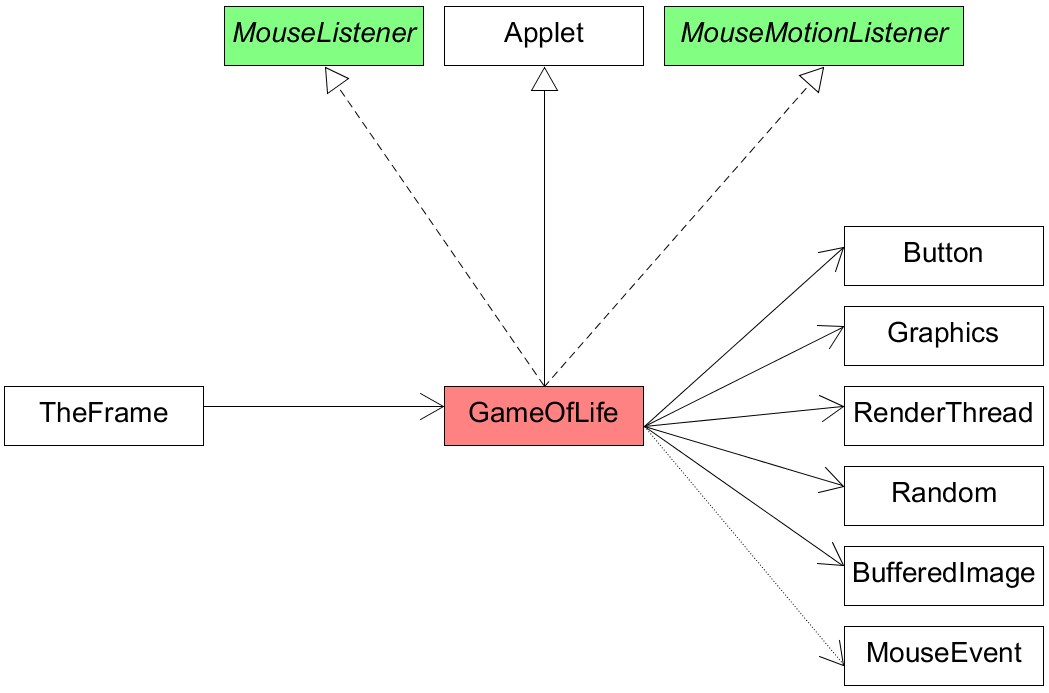
***Secondary testing: Code profiling and code optimization***

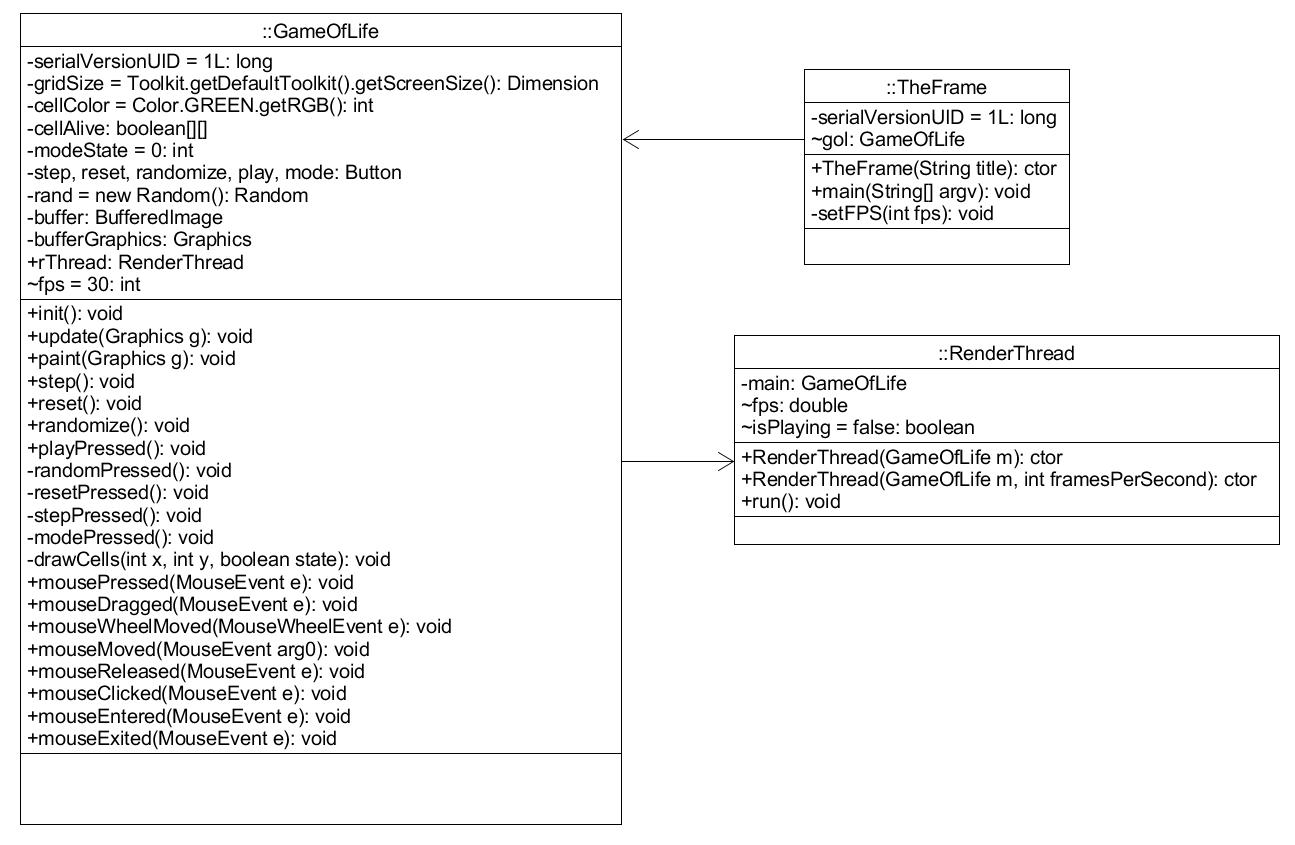
I also used java's inbuilt CPU profiling to look at how much time is spent in different methods and classes. Originally I used Graphics.drawLine to draw each pixel, which after profiling I discovered had a lot of overhead, and was slowing down the program. So I switched to using BufferedImage.setRGB which had much less overhead. Here are the latest profiling results (3:13 PM Wednesday, 19 October 2016). This goes to show the importance of testing for overhead and bottlenecks in the code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| RANK | % TIME | % ACCUMULATED | COUNT | JAVA METHOD |
| 1 | 42.27% | 42.27% | 2970 | sun.awt.windows.WToolkit.eventLoop |
| 2 | 13.09% | 55.37% | 920 | GameOfLife.step |
| 3 | 10.46% | 65.83% | 735 | GameOfLife.step |
| 4 | 8.07% | 73.90% | 567 | sun.java2d.loops.MaskBlit.MaskBlit |
| 5 | 6.43% | 80.33% | 452 | GameOfLife.paint |
| 6 | 5.81% | 86.14% | 408 | java.awt.image.BufferedImage.setRGB |
| 7 | 2.83% | 88.97% | 199 | sun.java2d.loops.FillRect.FillRect |
| 8 | 2.50% | 91.47% | 176 | GameOfLife.step |
| 9 | 2.49% | 93.97% | 175 | sun.java2d.loops.Blit.Blit |
| 10 | 2.35% | 96.31% | 165 | sun.java2d.d3d.D3DRenderQueue.flushBuffer |
| 11 | 1.84% | 98.15% | 129 | RenderThread.run |
| 12 | 0.21% | 98.36% | 15 | sun.java2d.d3d.D3DRenderQueue.flushBuffer |

This shows, as you would expect, that most of the thread time is spent in the calculation, rendering and java event loop, which is what you would expect.

**Class Diagrams (Note: first one a little outdated, but still relevant)**





***Explanation of the frame class:***

This class is intended to be an optional wrapper to run the applet as standalone (rather than converting the project completely to Swing or a Frame. This means I can choose to run it as an applet or standalone, which is useful) I've also added a menu with proper FPS control in the standalone version.

***Render thread class:***

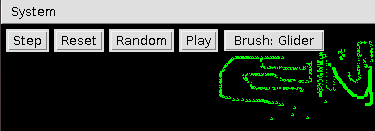
This class has the fps control functionality and calls the step and redraw functions defined in Game Of Life class. It also has the ability to be paused, but must be instantiated again in order to begin playing after being paused, due to the design of the class (pausing kills the render thread).

The FPS algorithm is quite simple and effective, and this is how it works.

A calculation is made at the start of a render thread, which calculates how many milliseconds per frame there should be (which is 1000/frames per second.) This is saved into a variable '*time'*. Then you get the current time in milliseconds and save it in another variable '*lastTick'.* It then enters a loop. If the current time is exactly 'time' milliseconds after the 'lastTick' then we step into the next frame and render, if not it loops until it is. Controlling frames/second is useful because it means we vary the speed of the animation, e.g. slowing it down to look at more detail.

***The brush/stamp drawing feature:***

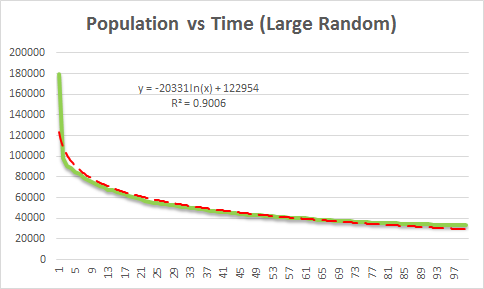
Later in development, A feature that was implemented was brush types. There are 5 types, 3 of which are normal, and two are special types. **Pixel brush**, which is the traditional single pixel drawing mode, **Glider brush**, lets you draw glider ships, **Square brush**, draws squares, and two special brushes, **Random brush** draws random shapes in a 3x3 square and **Inverting brush** inverts any alive cells to dead cells, and dead cells to alive cells in a 3x3 square. My particular implementation does not however have resizable brushes, sadly, due to the way that the backend code is written. The brushes are hard coded, such that it sets the state of cells at x and y coordinates of the users click with no loops, and there cannot be custom brushes made by the user. Another frustrating limitation of my current implementation is that drawing is hard to do accurately, due to it being based on singular pixels and any mouse movement is bound to not go exactly where the user wants it to.

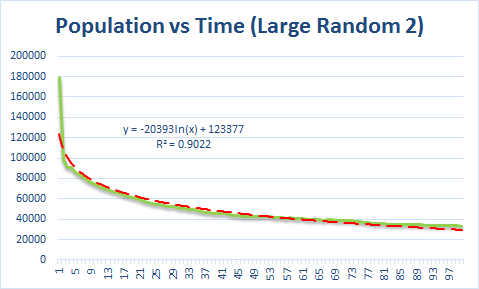


The reason why this feature was added in is that the user can create much more complicated shapes with this without too much need for accuracy.

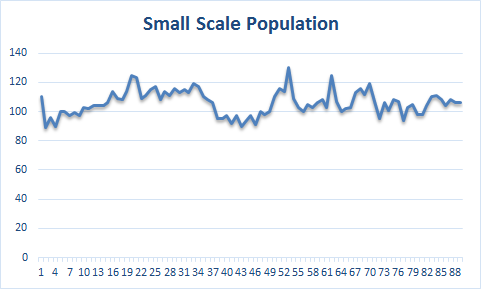
***Population analysis***

It’s interesting to note that population can be predicted with a logarithmic function when using an evenly distributed large population as a starting point. In the next two graphs, Green is the sample and red is the logarithmic function.





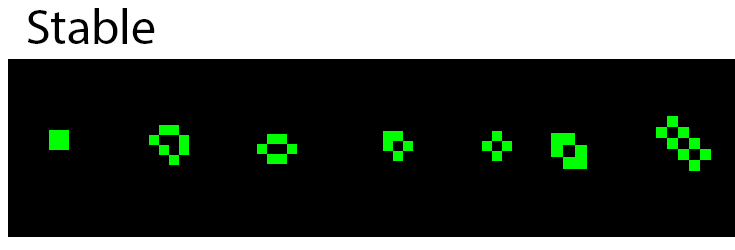
The prediction function however does not work on small scales.



This is simply because with large populations, the number of living cells declines over time, due to overpopulation. In a small scale setting, the number of living cells is mainly determined by intricate interactions between different cell groups, and thus without actually calculating each step directly from the game state, you cannot predict population.

**Interesting game of life structures and cell groups:**

First we have static types, ones that do not change and remain static, examples of these are:



Then we have moving types, these can travel across the screen and are called ‘spaceships’:



*C is the upper bound of which a cell can move, which is one cell per step. Just like how we have the speed of light as our upper bound in the universe. For the game of life, maximum possible speed is actually C/2, however speed of light travel is possible using other algorithms. Ref:* <https://en.wikipedia.org/wiki/Speed_of_light_(cellular_automaton>)

**Experimentation with GameOfLife.step() method:**

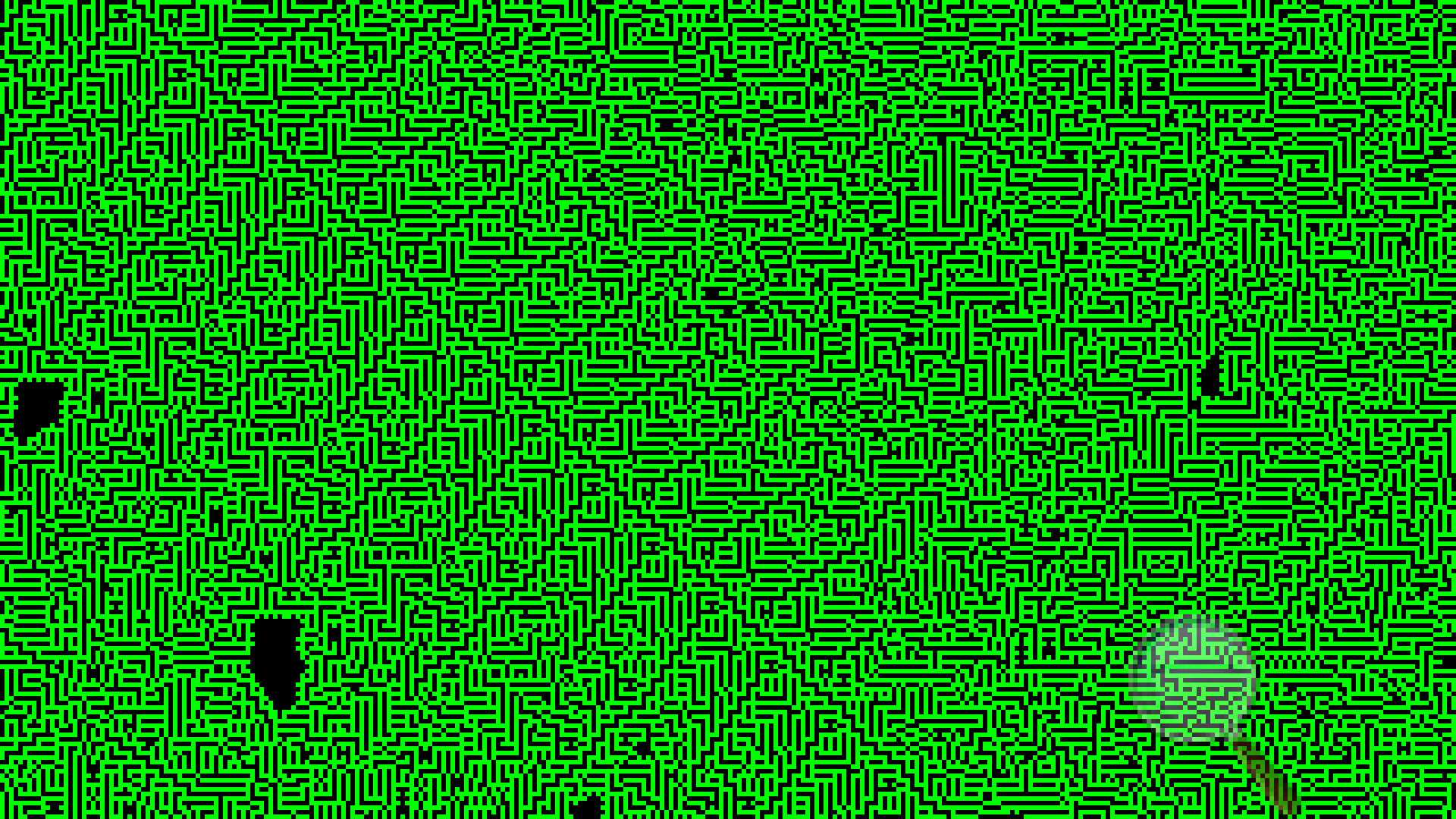
Not only can this applet be used to simulate game of life, but it can be used to simulate other cellular automata rulesets. This can be done by modifying the code in the step method

For example *Highlife* is a ruleset which builds on game of life by adding a single rule: dead cells can also become alive if it is surrounded by 6 living cells. This is easily implemented by adding an if statement ‘if(!cellAlive[x][y] && neighbours[x][y] == 6) cellAlive[x][y] = true’.

Screenshots of a game of life variant:



Maze Example



Maze Example 2

Instead of cells dying when more than 3 neighbours are alive, its more than 5 neighbours. This changes the result completely, and is a good example of how simple rules can create complex structures.

I have left some more examples of these in the Game of life standalone folder on OneDrive.

There is also the latest version of the code, which is a lot more complicated, but allows for simulation of any Survival/Birth ruleset and not just game of life.

Bibliography:

*Variations on the Game of Life*. (2016).*Cs.stanford.edu*. Retrieved 10 October 2016, from <https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/modeling-natural-systems/gameOfLife2.html>