

# Visualising 2D Lifelike Cellular Automata

Yaseen Azzabi (B719745)

supervised by Dr. Daniel Reidenbach

<b>Acknowledgements</b>	<b>4</b>
<b>1: Introduction</b>	<b>6</b>
1.1: Program Goals	6
<b>2: Technical Chapter: Cellular Automata</b>	<b>7</b>
2.1: An Introduction to 1D Cellular Automata	7
2.1.1: Why This Program Doesn't Simulate 1D Automata	11
2.2: An Introduction to 2D Cellular Automata	12
2.3: History of Cellular Automata	15
2.4: Different Types of Cellular Automata	17
2.4.1: Class 1 Cellular Automata	18
2.4.2: Class 2 Cellular Automata	19
2.4.3: Class 3 Cellular Automata	20
2.4.4: Class 4 Cellular Automata	21
2.4.5: Reversibility	22
2.4.6: Totality	23
2.5: Rule Representation for Cellular Automata	24
2.6: Lifelike Cellular Automata	31
2.6.1: Rule Representation for Lifelike Cellular Automata	35
2.7: Conway's Game of Life: a Lifelike Cellular Automaton	37
2.7.1: Characteristics of the Game of Life	39
2.7.2: Interesting Patterns in the Game of Life	41
2.7.2.1: Still Life	42
2.7.2.2: Oscillators	44
2.7.2.3: Spaceships, Guns and More	46
<b>3: Project Requirements</b>	<b>50</b>
3.1: System Requirements	51
<b>4: Design</b>	<b>52</b>
4.1: Development Approach	53
4.2: Panel Design (Overall Layout Design)	54
4.3: Graphics Panel Design (Panel 1)	58
4.3.1: Animation Design	60

4.4: UI Design and UX Design (Panels 2 and 3)	61
4.4.1: UI / UX Design of Panel 2 (Toolbar)	62
4.4.1.1: Popup Menus (Panel 2, Toolbar)	65
4.4.1.2: Sliders (Panel 2, Toolbar)	66
4.4.2: UI / UX Design of Panel 3 (Stamps)	67
4.4.3: Tooltips	69
4.5: InitialPopup Design	70
<b>5: Implementation and User Manual</b>	<b>77</b>
5.1: Program Structure and Class Diagram	77
5.2: Javadoc	80
5.3: Tools	82
5.3.1: Eclipse (the IDE)	82
5.3.2: WindowBuilder	82
5.4: Libraries and Imports	83
5.4.1: Swing	83
5.4.2: AWT	86
5.4.3: Java.io.* and Java.util.*	86
5.5: Important Features	87
5.5.1: Graphics (Panel 1)	87
5.5.2: Automaton Simulation Logic (check() and nextGeneration())	88
5.5.3: Communication (Flags) and Updating	92
5.5.4: Communication (Non-Flags)	95
5.5.5: mouseListener Methods	96
5.5.6: InitialPopup Implementation	98
5.6: User Manual	101
5.6.1: Starting Up	101
5.6.2: Drawing	104
5.6.3: Speed of Animation	107
5.6.4: Grid Options	107
5.6.5: Save / Load Options	108
5.6.6: Advanced Options	109
5.7: Problems Encountered	111
5.7.1: Drawing Graphics	111

5.7.2: Rule Changes	111
5.7.3: Communication Between Classes	111
<b>6: Testing</b>	<b>112</b>
6.1: Performance Testing	113
6.2: Functionality Tests	114
6.3: User Tests	116
6.3.1: User Comments	122
<b>7: Conclusions</b>	<b>125</b>
7.1: Future Expansion	125
7.1.1: File Browser	125
7.1.2: Rewind Feature	126
7.1.3: Improved UI	126
7.1.4: Increased Neighbourhood Sizes	126
7.1.5: Increased Size (and Resizable Simulation)	127
7.1.6: HashLife	127
7.3: Summary	128
<b>Appendix 1: nVidia G-Sync Incompatibility</b>	<b>129</b>
<b>Appendix 2: AWT Imports</b>	<b>130</b>
<b>References</b>	<b>133</b>

# Acknowledgements

*With special respect to **John Conway**; the main inspiration for this project.*

*John Conway's Game of Life is an inspired piece of art that demonstrably exhibits a great understanding and passion for the field of Computer Science.*

*With special thanks to **Dr. Daniel Reidenbach**; the supervisor of this project, without whom its completion would not have been possible.*

*Dr. Reidenbach has continually and consistently shown exceptional knowledge and wisdom throughout my time at Loughborough University.*

WITH UTMOST RESPECT TO  
**John Horton Conway**  
1937-2020

# 1: Introduction

A program is to be implemented that simulates lifelike cellular automata. These types of automata create complex, and often beautiful, simulated patterns of movement and evolution when visualised.

Such an example is John Conway's prolific Game of Life, the cellular automaton that popularised the idea of such automata.

In fact, the definition of lifelike cellular automata is based on the Game of Life: a cellular automaton must be similar to the Game of Life in order for it to be considered "lifelike".

Note that while lifelike cellular automata are always 2D, this fact is reiterated in the title for clarity's sake.

## 1.1: Program Goals

The main goal of this program is to simulate lifelike cellular automata in a fun and accessible way, even to users who are not particularly well-versed with automata in general.

The program must also be efficient, and run smoothly on any reasonable hardware specifications. These specifications were ultimately decided to be a minimum resolution of 1280x720p.

Furthermore, the program should be well-built; so as to allow for easy expansion of it in the future.

## 2: Technical Chapter: Cellular Automata

### 2.1: An Introduction to 1D Cellular Automata

The program implemented does not simulate 1D cellular automata (it simulates only 2D cellular automata, and only the subset of lifelike cellular automata). This subsection serves only as a brief overview of 1D cellular automata for the reader.

As a result, only the most basic types of 1D cellular automata are considered - these are the ***elementary cellular automata***, which are the most basic of all types of cellular automata.

Cellular automata consist of a grid of cells, where every cell is in exactly one state of a finite set of states. For each cell  $c$ , a set of cells called its *neighborhood* is defined relative to that cell  $c$ . These neighbourhoods can extend in several magnitudes. For 2D automata, these neighbourhoods can only extend in two dimensions. For 1D automata, they can only extend in one direction

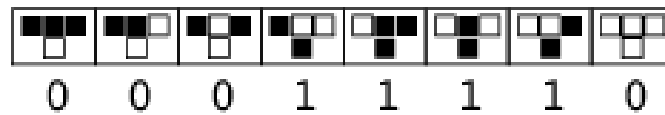
The state of a cellular automaton is simply its grid of cells and those cells' respective states.

Elementary cellular automata have exactly two possible values for each cell (0 or 1), and rules that depend only on the values of that cell's nearest neighbours. That is, the neighbourhood of any cell includes only its nearest neighbours.

This means that the evolution of an elementary cellular automaton can be described entirely with a table that gives the state a given cell will have in the next generation based on the value of the cell to its immediate left, the value of the cell itself, and the value of the cell to its immediate right. In total, there are  $2^8$  possible rules for elementary cellular automata (Weisstein, unknown).



Take the rule 30 as example:



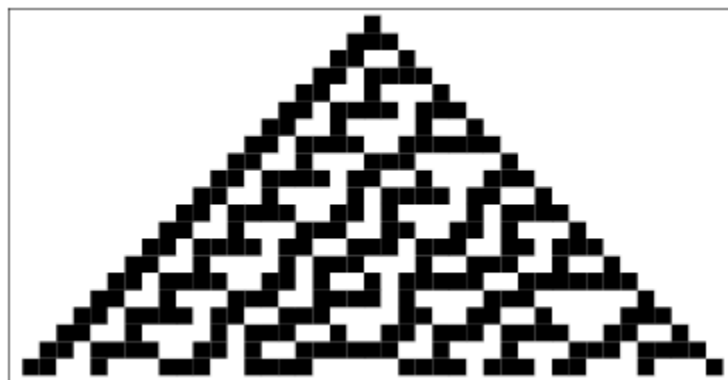
*Figure 1: Rule 30 for Elementary Cellular Automata.*

Each rule consists of two rows:

- The top row (in Figure 1) shows the current state of the automaton. The cell in the middle is the cell that is being considered, and the cells to its left and right are its neighbours.
- The bottom row (in Figure 1) shows the single cell “output” of the three-cell configuration in the top row. The value of the output depends on the binary bit associated with it (1 for black, 0 for white).

As seen in Figure 1, the rule is modelled as an 8-bit binary number. Rule 30 means that the 8-bit binary number being used is 00011110, which is 30 in binary. There are 8 possible configurations that a cell and its neighbours can be in, which are totally described in the sub images of figure 1.

The associated binary bit to each sub image gives the resulting state of the cell in the next generation. If the bit is 1, then the cell will be in state 1 in the next generation. If the bit is 0, then the cell will be in state 0 in the next generation.



*Figure 2: The evolution of a 1D elementary cellular automata using Rule 30.<sup>1</sup>*

<sup>1</sup> Figures 1 and 2 taken from <https://mathworld.wolfram.com/ElementaryCellularAutomaton.html>.

Figure 2 illustrates how rules are applied. The top row in Figure 2 is the initial configuration. Each row beneath it is a new generation. Any cells that are “missing” are assumed to be state 0, or white.

This can most clearly be seen in the top row. All three cells in generation 1 are black, and so have the value of 1. However, only one cell in generation 0 even exists. Because of this, we assume that all neighbours that are missing are of state 0.

0 (assumed)	0 (assumed)	1	0 (assumed)	0 (assumed)
	1	1	1	

*Figure 3: A clearer illustration of the top row generating the second row of Figure 2.*

While the four cells in the top row of Figure 3 with state 0 do not actually exist in Figure 2, they are assumed to take these values because they are missing.

It is now clear how the second row of Figure 2 is generated. Counting from the left, we have:

- The leftmost cell in the second row has state 1 because of the seventh part of rule 30; i.e. 00011110 applies.
- The central cell in the second row has state 1 because of the sixth part of rule 30; i.e. 00011110 applies.
- The rightmost cell in the second row has state 1 because of the fourth part of rule 30; i.e. 00011110 applies.

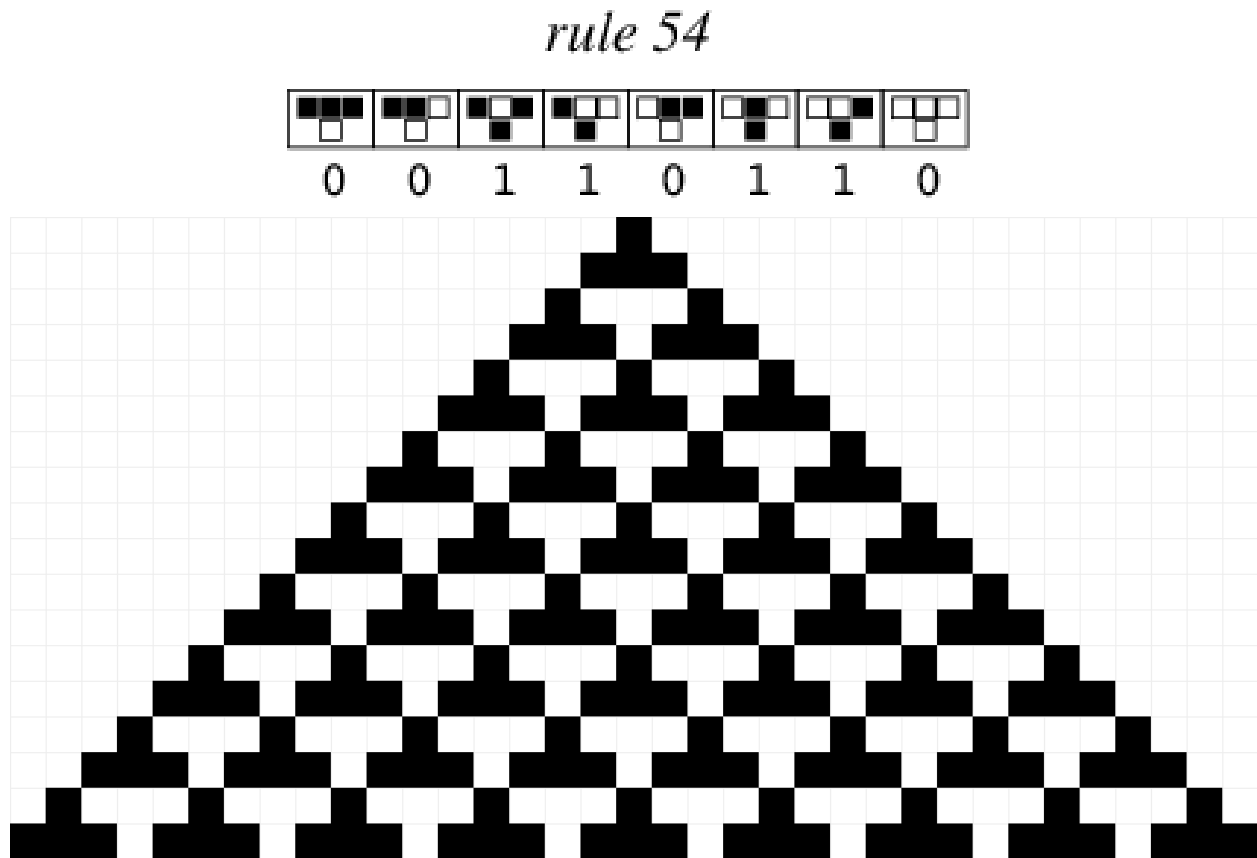


Figure 4: Elementary Cellular Automata rule 54 (top) and the evolution of a cellular automata using rule 54 (bottom).<sup>2</sup>

Instead of considering the top row and the second row, we shall now consider the third row and how it generates the fourth row.

0 (assumed)	0 (assumed)	1	0	0	0	1	0 (assumed)	0 (assumed)
		1	1	1	0	1	1	1

Figure 5: A clearer illustration of the third row generating the fourth row of Figure 4.

Again, counting from the left:

<sup>2</sup> images taken from <https://mathworld.wolfram.com/ElementaryCellularAutomaton.html> and [https://miro.medium.com/max/2400/1\\*\\_vz1un\\_yp-G446BcJZpMgw.png](https://miro.medium.com/max/2400/1*_vz1un_yp-G446BcJZpMgw.png) respectively.

- the leftmost cell (in the bottom row of Figure 5) has state 1 because of the seventh part of rule 54 (shown in Figure 4); i.e. 00110110 applies.
- the second cell has state 1 because of the sixth part of rule 54; i.e. 00110110 applies.
- the third cell has state 1 because of the fourth part of rule 54; i.e. 00110110 applies.
- the fourth cell has state 0 because of the eighth part of rule 54; i.e. 00110110 applies.
- the fifth cell has state 1 because of the seventh part of rule 54; i.e. 00110110 applies.
- the sixth cell has state 1 because of the sixth part of rule 54; i.e. 00110110 applies.
- the rightmost cell has state 1 because of the fourth part of rule 54; i.e. 00110110 applies.

### 2.1.1: Why This Program Doesn't Simulate 1D Automata

1D cellular automata, while being somewhat similar in their concept to their 2D counterparts, are highly dissimilar in their implementation. For more information on 2D cellular automata, see the section: [An Introduction to 2D Cellular Automata](#).

Their rules are defined differently as they only act in one dimension (left and right), and more importantly, they are drawn in subsequent rows whereas their 2D counterparts are drawn in subsequent grids (which usually replace one another).

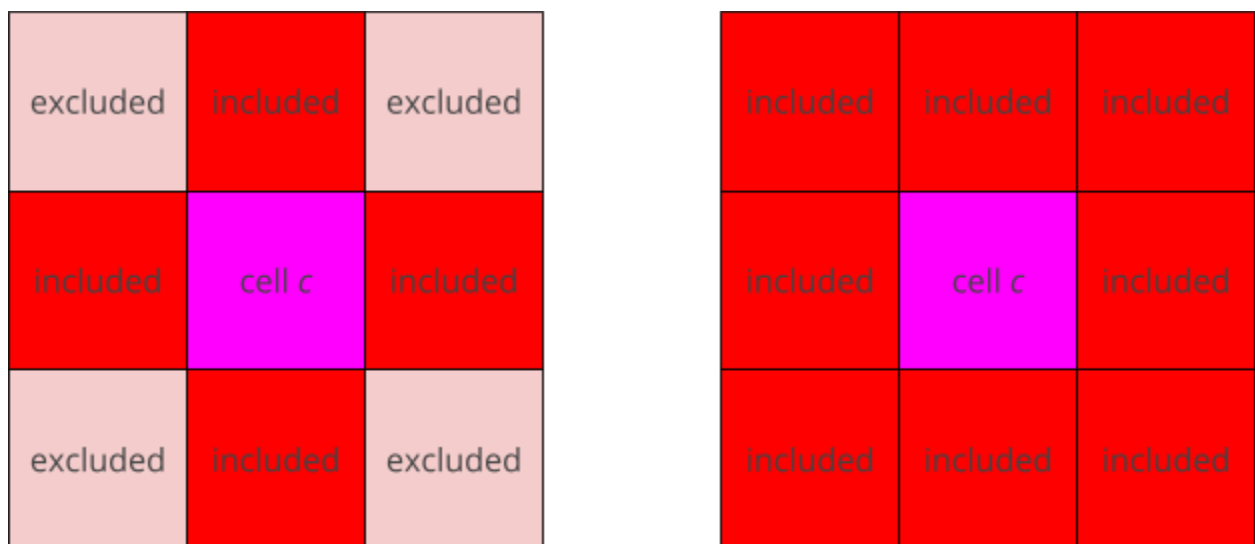
In the most succinct way possible; the code that simulates 1D cellular is completely different from the code that simulates 2D cellular automata.

The implemented program had the overall aim to bring as many features pertaining to 2D lifelike cellular automata as possible. In order to make time for this, given the under-a-year long development window, simulation of 1D cellular automata was ruled out during the planning phase.

## 2.2: An Introduction to 2D Cellular Automata

Similarly to their 1D counterparts, these types of cellular automaton consist of a grid of cells, where every cell is in exactly one state of a finite set of states. For each cell  $c$ , a set of cells called its *neighborhood* is defined relative to that cell  $c$ . These neighbourhoods can extend in several directions and magnitudes.

Take the following example of a 2D cellular automata with its neighbourhood definition highlighted:



*Figure 6.a (left) and Figure 6.b (right).*

Figures 6.a and 6.b illustrate how neighbourhood definitions can be different between two automata. Both automata have the same number of cells and each cell in either automaton has two possible states: *active* and *inactive*. While these automata support more than just two states, we consider only such automata for simplicity's sake (furthermore, lifelike cellular automata only ever have two states).

In each figure, the cell we are considering is the central cell  $c$ , and  $c$ 's neighbourhood cells are highlighted in deep red and labelled "included".

In Figure 6.a, only cells that are above, below, or immediately adjacent to cell  $c$  are part of  $c$ 's neighbourhood. More simply put, each cell has an orthogonal neighbourhood of radius 1.

In Figure 6.b, all cells that surround a cell  $c$  are in  $c$ 's neighbourhood. This type of neighbourhood is known as a *Moore neighbourhood* (with a radius of 1) (Weisstein, unknown).

A new *generation* (or more simply, a new state) of the automaton is brought about by updating the state of every cell in the automaton based on facts about their neighbourhoods.

To do this, *rules* are defined for the cellular automaton. These rules determine how the state of every cell in the automaton should change through generations depending on facts about the cells' neighbourhoods.

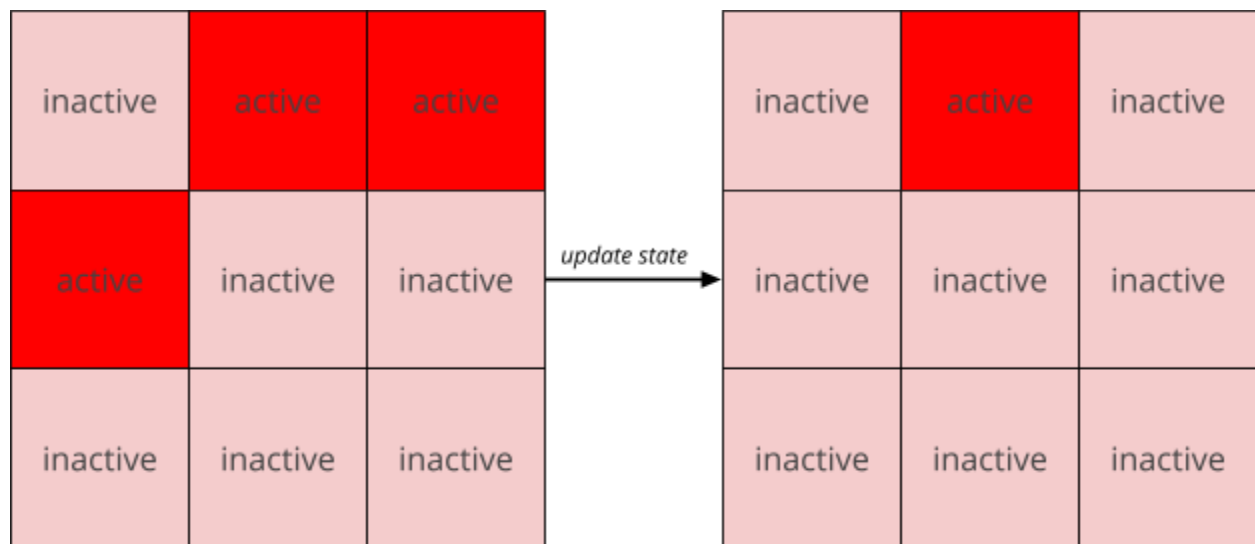


Figure 7: A cellular automaton based on Figure 6.b updating its state.

To illustrate this, Figure 7 shows a cellular automaton with the following rules and neighbourhood definition:

- FIRST RULE: if an active cell  $c$  has at least 2 active neighbours, then it remains active.
- SECOND RULE: if an active cell  $c$  has fewer than 2 active neighbours, then it becomes inactive.
- NEIGHBOURHOOD DEFINITION: a cell  $d$  is in the neighbourhood of  $c$  if it immediately surrounds  $c$  in any direction, including diagonally.

We can clearly see that only one of three active cells in the automaton above have a neighbourhood size of 2. This is the cell that remains active, due to the first rule. The other two active cells have a neighbourhood size of only 1, so they both become inactive due to the second rule.

## 2.3: History of Cellular Automata

Cellular automata were originally proposed by John von Neumann as formal models of self-reproducing organisms (Palash Sarkar, 2000). John von Neumann worked on the problem of self-replicating systems, whose initial design was based on the idea of one robot building another robot. As he developed this design, von Neumann realised that such automata would be very expensive to implement, given the cost in providing the robot with a "sea of parts" from which it would need to draw from in order to build its copy (Buckley, 2000).

Circa 1947, von Neumann began by thinking about robotics and imagined creating an example using a toy construction set. By using an analogy to electric circuit layouts, he realized that a 2D design would be enough to form a proof that a robot could replicate. Following a 1951 suggestion from Stanislaw Ulam, von Neumann simplified his model and ended up with what we now refer to as a 2D cellular automaton (Wolfram, 2002).

Von Neumann constructed this cellular automaton (known as a "von Neumann universal constructor") in 1952 and 1953. It had 29 possible colors for each cell - that is 29 possible states for each cell - and was given rules that were specifically set up to emulate the operations of components of a computer and various mechanical devices (i.e. to simulate a real machine) (Wolfram, 2002).

To give proof of the possibility of real-world self-reproduction, von Neumann then outlined the construction of a 200,000 cell configuration which would reproduce itself (Wolfram, 2002).

Because the automaton worked with rules that were rooted in reality, von Neumann had successfully proved that a machine could technically be created that would reproduce itself - all without having to actually build any machine.

Shortly after von Neumann's work in cellular automata, attempts were made to capture the essence of self-replication by mathematical studies of the properties of cellular automata. Over the course of the 1960s, constructions were found for



progressively simpler (in terms of rules) cellular automata capable of self-replication (Wolfram, 2002).

Research progressed, and in 1970, a now incredibly well-known cellular automaton was borne by John Conway, which he dubbed the “Game of Life”. It was allegedly motivated in part by questions in mathematical logic, and in part by work on so-called *simulation games* by Stanislaw Ulam (among other researchers) (Wolfram, 2002).

In 1968, Conway began experimenting with 2D cellular automaton rules, and by 1970 he had come up with the strikingly simple, yet beautiful, set of rules for his Game of Life (Wolfram, 2002). These simple rules were able to exhibit a range of complex behavior, as seen in the program written here that visualises the Game of Life by default.

The Game of Life became widely known, and many people (hobbyists, not scientists) spent their free time finding specific initial conditions to the Game of Life that give birth to particular forms of aesthetically pleasing or otherwise interesting behaviour. It has become so popular that entire websites and forums have been created devoted to it, such as <https://www.conwaylife.com/>.

Almost no systematic scientific work was done (Conway himself treated the Game of Life as a recreation, as its name suggests), and almost without exception only the very specific rules of the Game of Life were ever investigated scientifically (Wolfram, 2002).

## 2.4: Different Types of Cellular Automata

In his paper "Universality and Complexity in Cellular Automata", Stephen Wolfram proposed a classification of cellular automaton rules into four types, according to the results of *evolving* (evolving is the passing of generations in a cellular automaton) the system from a "disordered initial state".

A "disordered initial state" can be modelled as an initial state where the state of every cell in the automaton is randomly selected.

In order of increasing complexity, these classes are named in a straightforward manner:

- Class 1,
- Class 2,
- Class 3,
- Class 4.

For instance, the rules of class 4 automata are more complex than the rules of class 3, class 2 and class 1 automata.

These classes are generalisations of cellular automata. Lifelike cellular automata can be considered a specialisation of cellular automata, and are discussed [in their own chapter](#).

### 2.4.1: Class 1 Cellular Automata

In class 1 cellular automata, *evolution* (the passing of generations) leads to a single homogeneous state.

Wolfram (1984) states that “class 1 cellular automata evolve after a finite number of time steps from almost all initial states to a unique homogeneous state, in which all sites have the same value”.

When an automaton reaches a homogeneous state, it does not change from that state through any number of generations.

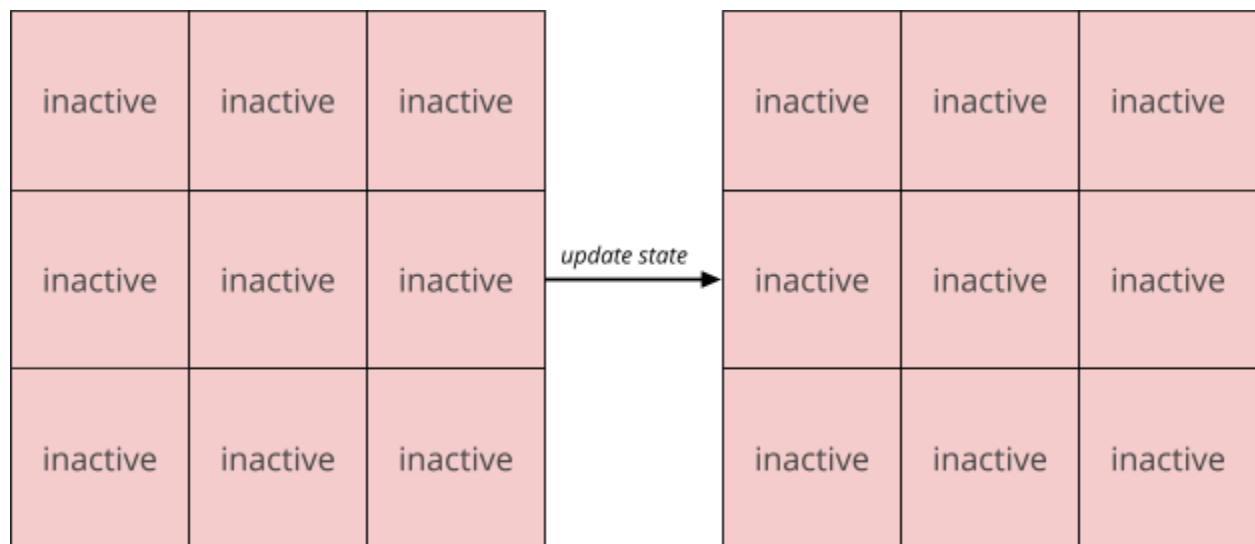


Figure 8: A 2D cellular automaton that is in a homogeneous state.

In Figure 8, the rule: “if a cell is active, it becomes inactive in the next generation.” is depicted. This exhibits class 1 behaviour, as the rule forces the state in which all cells are inactive forever.

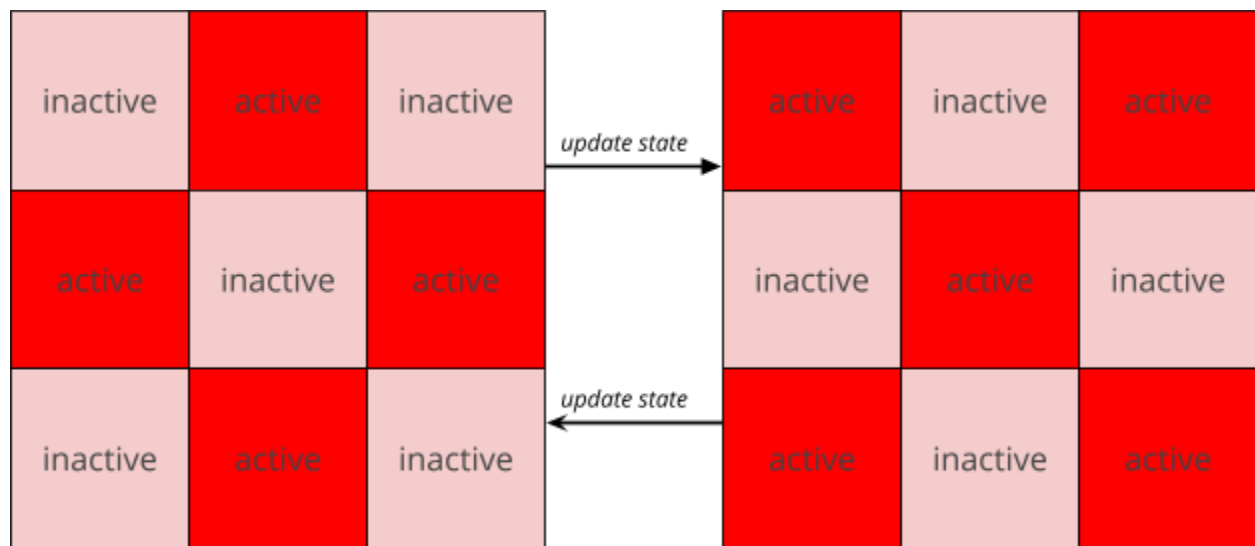
### 2.4.2: Class 2 Cellular Automata

In class 2 cellular automata, evolution leads to a set of stable or periodic structures.

Wolfram (1984) said that “class 2 cellular automata evolve after a finite number of time steps from almost all initial states into stable or oscillating structures”.

That is, Class 2 cellular automata reach a point where structures in that automata cycle back and forth between certain “shapes”.

Some of the randomness in the initial pattern of the automaton may filter out, but typically some remains. Local changes to the initial pattern tend to remain local - that is, they usually do not propagate through the entire automaton (Illachinsky, 2001).



*Figure 9: A 2D cellular automaton that is oscillating between two states.*

Figure 9 depicts the rule: “if a cell has an active orthogonal neighbour, it becomes active in the next generation. Otherwise, it becomes inactive in the next generation.”

This rule is exhibiting class 2 behaviour as the automaton is oscillating between the two states shown in Figure 9.

### 2.4.3: Class 3 Cellular Automata

In class 3 cellular automata, evolution leads to a chaotic pattern.

Wolfram (1984) said that the “evolution of infinite class 3 cellular automata from almost all possible initial states leads to aperiodic (or *chaotic*) patterns. After sufficiently many time steps, the statistical properties of these patterns are typically the same for almost all initial states”.

That is, a class 3 cellular automaton will follow no discernable pattern after a given amount of time - they lack oscillation (*aperiodic*) and also lack any other form of predictability or cyclicity (*chaotic*).

Any stable or oscillating structures that appear in class 3 cellular automaton are quickly destroyed by the surrounding noise (Illachinsky, 2001).

It is difficult to create a two-panel graphic for such automata; to clearly illustrate the chaotic nature of them, many states need to be shown. Instead, try to imagine a cellular automaton that has rules that allow it to almost “randomly” choose which cells become active or inactive in a given generation.

#### 2.4.4: Class 4 Cellular Automata

In class 4 cellular automata, evolution leads to complex localised structures, but after long periods of time may lead to a class 2 type behaviour (Ilachinski, 2001) (Wolfram, 1984).

The best possible example of a class 4 cellular automaton is John Conway's Game of Life, which is visualised by default in the supplied program.

Such "complex localised structures" are present in the Game of Life, and are discussed later on. For instance, the well-known "glider", which are patterns of cells that propagate in one direction through the Game of Life, are examples of such structures.

Furthermore, when running the visualiser program, you will see that the automaton tends to end in states where all that is left are oscillating structures (class 2 type behaviour), which are also discussed later on.

### 2.4.5: Reversibility

Cellular automata may also be *reversible*. A cellular automaton is reversible if, for every possible configuration of the cellular automaton, there is exactly one past configuration (also known as a *preimage*). This means that such an automaton can always be time-reversed to its initial state with confidence.

Consider as example the cellular automaton (with two states, “active” and “inactive”) with the rule: “every inactive cell becomes active in the next generation if and only if the cell to its immediate left is active in the current generation”:

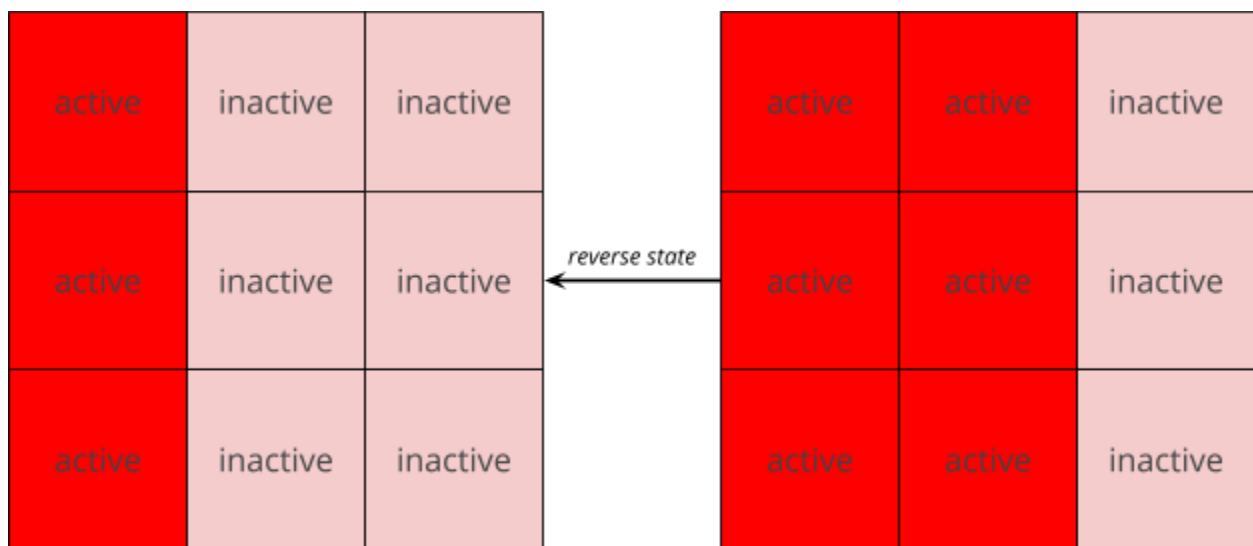


Figure 10: The 2D cellular automata described with the rule given above it.

If a cellular automaton is reversible, its time-reversed behavior can also be described as a cellular automaton itself (an alternative definition for a reversible cellular automaton from Jarkko Kari (1990): “A cellular automaton is called reversible if it has an inverse automaton, that is a cellular automaton that makes the system retrace its steps backwards in time”).

For cellular automata of two or more dimensions, reversibility is unfortunately undecidable. (Jarkko Kari, 1990).

#### 2.4.6: Totality

Cellular automata may also be *totalistic*.

The state of each cell in a totalistic cellular automaton is represented by a number, and the value of a cell at time  $t$  depends only on the sum of the values of the cells in its neighbourhood (which may include the cell itself) at time  $t - 1$  (Wolfram, 2002) (Illachinski, 2001).

If the state of the cell at time  $t$  depends on both its own state *and* the total of its neighbours at time  $t - 1$  then the cellular automaton is considered *outer totalistic* (Illachinski, 2001).

The Game of Life is an example of an outer totalistic cellular automaton, with the cell values 0 ("dead") and 1 ("alive"). Its rules will be considered later, and the reasoning behind why the Game of Life is outer-totalistic will be considered then.



## 2.5: Rule Representation for Cellular Automata

In the context of a Java program, rule representation for cellular automata could be achieved in many ways (such is the nature of coding: there are usually several solutions to any given problem).

Take the following representation of a cellular automaton's grid of cells and that automaton's neighbourhood definition:

*Note that information on the rule implementation of lifelike cellular automata as simulated in the included program is under the subsection [Lifelike Cellular Automata](#).*

included	included	included
included	cell c	included
included	included	included

*Figure 11: An example cellular automaton grid with the neighbourhood of the given cell  $c$  highlighted and labelled "included".*

Because rules in cellular automata always depend on the neighbourhood of a given cell, the first thing to do is to fetch the neighbourhood of the cell being considered.

In Figure 11, we consider all cells that are directly connected to a cell  $c$  as being in  $c$ 's neighbourhood ( $c$  is not included). Implementing this is as simple as looking at each cell in the neighbourhood of  $c$  and saving its state into a variable or an array.

Once the states of the neighbourhood cells have been found, rules can be applied to  $c$  using that information.

Finding the number of active cells in a given cell  $c$ 's neighbourhood in pseudocode:

```
function getActiveNeighbours(cell c){
    int activeNeighbourCount;
    int inactiveNeighbourCount;
    for every cell  $i$  in neighbourhood of cell  $c$  {
        // for every cell in the neighbourhood of the cell being
        // considered
        if ( $i.getState() == \text{"active"}$ ) {
            // if the cell is active, increment
            // "activeNeighbourCount"
            activeNeighbourCount += 1;
        }
        else {
            // otherwise, increment "inactiveNeighbourCount".
            inactiveNeighbourCount += 1;
        }
    }
}
```

Fetching neighbourhoods in this way is sufficient to implement totalistic cellular automata. However, non-totalistic automata might depend on specific cell locations being active or inactive (or if there are more states, they might also be considered too).

To this end, we could also save the position of each cell along with its state to allow non-totalistic rules to be implemented. For instance, a rule might depend on the state of specific cells that are immediately to the left of a cell *c*, for which simply summing the number of active neighbours would not be sufficient to implement.

```
function getActiveNeighboursNonTotalistic(cell c){
    array nestedNeighbourArray;
    for every cell i in neighbourhood of cell c {
        if (i.getState() = "active") {
            // if the cell is active, save "active" to the array.
            array currentUpdateArray = [i.position(),
            "active"];
            neighbourArray.add(currentUpdateArray);
        }
        else {
            // otherwise, save "inactive" to the array.
            array currentUpdateArray = [i.position(),
            "inactive"];
            neighbourArray.add(currentUpdateArray);
        }
    }
}
```

Because both the positions and states of neighbourhood cells are saved, non-totalistic rules can now be implemented.

For instance, the rule "if cell left\_1 is active then cell *c* is active in the next generation" can be implemented using "getActiveNeighboursNonTotalistic".

However, if the function "getActiveNeighbours" was used, it would be impossible to implement because it would not be possible to check if the specific cell "left\_1" was active or not. Instead, it would only be possible to see how many active neighbours a given cell *c* had.

cell 1 (inactive)	cell 2 (active)	cell 3 (inactive)
cell 4 (active)	cell c (inactive)	cell 6 (inactive)
cell 7 (inactive)	cell 8 (active)	cell 9 (inactive)

Figure 12: A state of the automaton grid shown in Figure 11.

Let us now consider an example automaton based on the neighbourhood definition given in Figure 11 (all cells directly connected to cell c are included). Figure 12 shows a state of this automaton, and each cell is labelled with a name. Each cell can be in one of two states, “active” or “inactive”. In this automaton, the cell c is not included in its own neighbourhood.

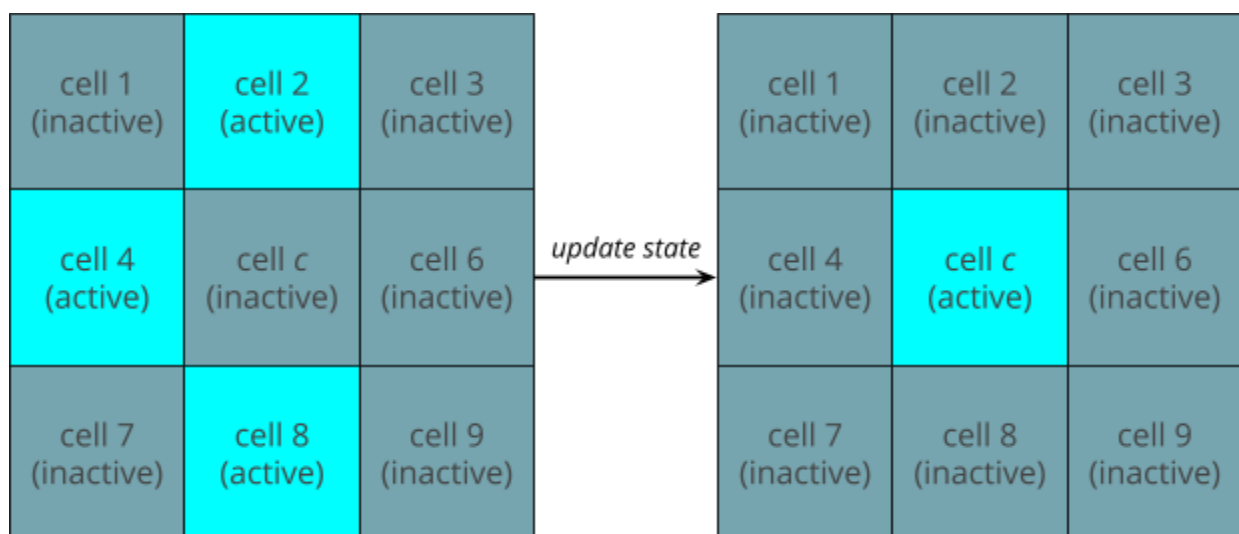


Figure 13: The grid in Figure 12 paired with rule 1 and rule 2 updating its state.

In figure 13, the rules being applied are:

- RULE 1: if a cell  $c$  has 3 or more active neighbours in the current generation, then it will be active in the next generation.
- RULE 2: if a cell  $c$  has fewer than 3 active neighbours in the current generation, it will be inactive in the next generation.
  - *think of this rule as an "inverse" of rule 1.*

The only cell in Figure 13 that has 3 active neighbours is cell  $c$ . Therefore, it is the only cell that is active when the automaton updates its state.

All other cells have 2 or 1 active neighbours. Cell 4 has 2 active neighbours in cell 2 and cell 8, for example.

In pseudocode, these rules could be modelled as:

```
function updateRule(currentGeneration){
    for every cell  $c$  in currentGeneration {
        // for every cell in the current generation of the
        // automaton...
        if (getActiveNeighbours( $c$ ) >= 3) {
            // if there are three or more active neighbours, set
            // the cell to be active in the next generation.
            nextGeneration.setCellActive( $c$ );
        }
        else {
            // otherwise, the cell remains inactive in the next
            // generation.
            nextGeneration.setCellInactive( $c$ );
        }
    }
}
```

excluded	included	excluded
excluded	cell c	excluded
excluded	excluded	excluded

Figure 14: Introducing a new automaton grid and a different neighbourhood definition.

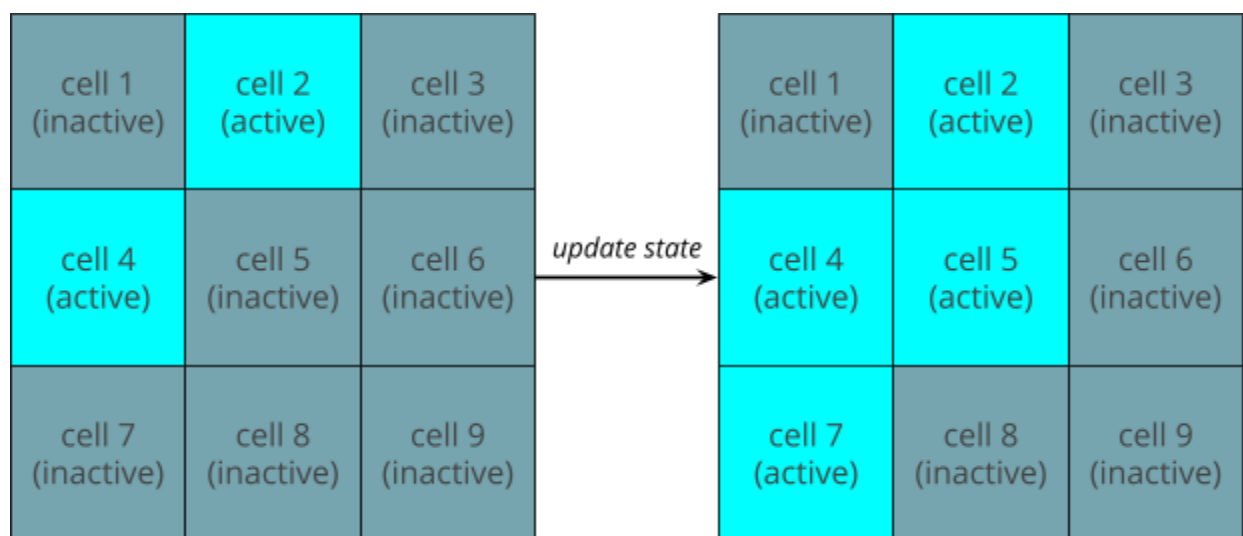


Figure 15: an automaton based on Figure 14 paired with rule 3 updating its state.

In figure 15, the rule being applied is:

- RULE 3: if a cell has an active neighbour directly above it in the current generation, then it will be active in the next generation.

Both cell 5 and cell 7 are inactive and have an active neighbour directly above them in the left state. Therefore, when the automaton updates its state, cell 5 and cell 7 become active.

Because the rules for this automaton only consider one cell (the cell above cell  $c$ ), the neighbourhood in Figure 14 reflects this. When considering if a cell  $c$  is to be made active in the next generation, only the cell directly above  $c$  is considered.

In pseudocode, these rules could be modelled as:

```
function updateRule(currentGeneration){
    for every cell  $c$  in currentGeneration {
        // for every cell in the current generation of the
        // automaton...
        if (getActiveNeighbours( $c$ ) == 1) {
            // if the neighbour appears above, set the cell to be
            // active in the next generation.
            nextGeneration.setCellActive( $c$ );
        }
        // there is no else condition as the only rule is
        // the one that is defined by the above if statement.
    }
}
```

We still write “if (cell.getActiveNeighbours() == 1)”. Even though the neighbourhood is only of size one, we do not write something like “if (cell.isNeighbourActive() == true)” because that reduces the flexibility and expandability of the resulting program.

Writing the rule definition in the way that it is means that if the neighbourhood definition ever changes, it is simple and easy to update the “updateRule” function.

Another important note is that although this rule could very well be modelled non-totalistically, it has been demonstrated that it can be modelled using a totalistic approach. In fact, the rule that is implemented in pseudocode is actually different from rule 3, but it clearly has the same effect as rule 3.

## 2.6: Lifelike Cellular Automata

First, know that lifelike cellular automata can be in any of the four generic classes described earlier in [Different Types of Cellular Automata](#). The most important thing to note is that lifelike cellular automata are simply a subset of 2D cellular automata.

The program implemented simulates all lifelike cellular automata (except for ones where cells are in their own neighbourhood, which are again a further specialisation of lifelike cellular automata).

A cellular automaton is lifelike (in the sense of being similar to Conway's Game of Life) if and only if it meets the following criteria (Eppstein, 2010):

1. the cellular automaton must be [two-dimensional](#).
2. each cell of the automaton has exactly two states (i.e. "alive" or "dead").
3. the neighborhood of each cell is the [Moore neighborhood](#) of radius 1.
  - note that the neighbourhood of a cell might include itself, but this is not a necessary property.
4. the automaton must be [outer-totalistic](#).

Outer-totalistic rules are used to model rules that simulate real cellular life. For instance, if a cell has lots of live neighbours, it might die as if the simulation is arguing that the cell is living in an overpopulated environment.



Because of these criteria, all lifelike cellular automata can be modelled using exactly three rules (or rather, two rules and the knowledge if neither apply):

1. **BIRTH RULE:** if the number of live neighbours that a *dead* cell has is in a given subset of  $[0, 9]$ , then that cell will be alive in the next generation, as if by reproduction.
  - this given subset might be  $\{1, 2, 3\}$ . In this case, if a dead cell has 1, 2 or 3 living neighbours, then it will become alive in the next generation.
2. **SURVIVAL RULE:** if the number of live neighbours that a *live* cell has is in a given subset of  $[0, 9]$ , then that cell will be alive in the next generation, as if it were living in a suitable habitat.
  - this given subset is often (but not strictly) different from the one given for the birth rule. For example, if this rule's subset was  $\{5, 6\}$ , then living cells with 5 or 6 neighbours would stay alive in the next generation.
3. **DEFAULT RULE:** if neither of rule 1 or rule 2 apply, then any cell, regardless if it is alive or not, will be dead in the next generation.
  - for the subsets given above, this means that any cell with 4, 7, 8 or 9 neighbours (9 only if a cell's neighbourhood includes itself) would be dead in the next generation.

Clearly, these rules can model outer-totalistic cellular automata.

- a cell's future state depends only on its state and the state of cells in its neighbourhood - these rules model that.
- a further restriction for lifelike cellular automata is that the neighbourhood must be one that is a Moore neighbourhood of radius 1.

These rules are commonly written in the form “Bx/Sy”. Each of x and y is a sequence of distinct digits from 0 to 8, in numerical order (Eppstein, 2010).

- the presence of a digit  $d$  in the x-string means that a dead cell with  $d$  live neighbours becomes alive in the next generation.
- the presence of a digit  $d$  in the y-string means that a live cell with  $d$  live neighbours survives into the next generation of the automaton.

In the de facto RLE format<sup>3</sup> for storing lifelike automaton rules, this notation is used.

- furthermore, the implemented program uses this notation to allow advanced users to change the default rule configuration to any Bx/Sy rule.

For instance, if the rule “B3/S23” was used, then:

- a dead cell with three live neighbours would become alive in the next generation.
- a live cell with two or three neighbours would remain alive in the next generation.
- any cell that does not follow the above two rules would be dead in the next generation.

B3/S23 is the rule configuration that Conway’s Game of Life uses.

---

<sup>3</sup> some information on the RLE format can be found at [https://www.conwaylife.com/wiki/Run\\_Length\\_Encoded#:~:text=The%20Run%20Length%20Encoded%20\(or,in%20the%20MCell%20file%20format.](https://www.conwaylife.com/wiki/Run_Length_Encoded#:~:text=The%20Run%20Length%20Encoded%20(or,in%20the%20MCell%20file%20format.)

cell <i>a</i> (active)	cell <i>b</i> (active)	cell <i>c</i> (inactive)	cell <i>d</i> (inactive)
cell <i>e</i> (active)	cell <i>f</i> (inactive)	cell <i>g</i> (inactive)	cell <i>h</i> (active)
cell <i>i</i> (inactive)	cell <i>j</i> (inactive)	cell <i>k</i> (inactive)	cell <i>l</i> (inactive)
cell <i>m</i> (inactive)	cell <i>n</i> (inactive)	cell <i>o</i> (active)	cell <i>p</i> (active)

*Figure 16: a configuration of cells.*

Consider the configuration of cells shown in Figure 16. If the rule B3/S1 was used:

- cells *a*, *b* and *e* would die in the next generation.
  - they all have two neighbours, and since “S1” does not contain the number 2, then live cells with two neighbours must die.
- cells *f*, *k* and *l* would become alive in the next generation.
  - these cells are all dead and all have three live neighbours, and since the part of the rule “B3” supports that, they must become alive in the next generation.
- cells *o* and *p* will survive in the next generation.
  - they both have one neighbour, and this is supported by the part of the rule “S1”.
- no rules apply to any of the other cells, so they will all be dead in the next generation.
  - including the currently live cell *h*, which does not have exactly 1 neighbour as required in “S1”. This is the same logic that resulted in cells *a*, *b* and *e* dying.

### 2.6.1: Rule Representation for Lifelike Cellular Automata

Because lifelike cellular automata can always have their rules modelled in the Bx/Sy notation, the basis needed to represent rules in code is the storing of the x-string and the y-string. Like in the chapter [Rule Representation for Cellular Automata](#), we use Java as the background.

We can take each number in the x-string and y-string and add them to an array. The x-string will be stored individually in an array `birthSet`, and the y-string will be stored individually in an array `survivalSet`.

We can then count the number of neighbours for each cell in the automaton, and check this number against the relevant set. If the cell is alive, then we check it against `survivalSet`, otherwise the cell is dead and we check it against `birthSet`.

For instance, if the rule B3/S23 was used then:

- `birthSet` would be {3}.
- `survivalSet` would be {2, 3}.

Therefore, if a cell `c` had two neighbours and was dead, it would be checked against `birthSet`. Because `birthSet` does not contain 2, we would conclude that `c` should not be born in the next generation (it should remain dead).

However, if `c` was alive and had 2 neighbours, it would instead be checked against `survivalSet`. In this case, because `survivalSet` contains the element 2, we would conclude that `c` should survive in the next generation.

This results in the pseudocode:

```
function getLiveNeighbours(cell c){
    int neighbours = 0;
    for every cell i in Moore neighbourhood of c {
        if (i.getState() == "alive") {
            neighbours += 1;
        }
    }
    return neighbours;
}

function nextGeneration(cellular automaton automaton,
                        B-rule birthSet,
                        S-rule survivalSet){

    for every cell c in automaton {
        currentNeighbours = getLiveNeighbours(c);

        if (c.getState() == "dead" AND currentNeighbours is in
            birthSet) { // if birth rule applies.
            set c to be active in next generation;
        }

        else if (c.getState() == "alive" AND currentNeighbours
            is in survivalSet){ // if survival rule applies.
            set c to remain active in next generation;
        }
        else { // if no rule applies.
            set c to be dead in next generation;
        }
    }
}
```

This slice of pseudocode tests every cell in the automaton against the rules of the automaton.

## 2.7: Conway's Game of Life: a Lifelike Cellular Automaton

John Conway's Game of Life is a *non-reversible, outer-totalistic*, class 4 cellular automaton. It is simulated by default in the program delivered.

The Game of Life models its grid of cells as a grid of organic cells. Each cell is in exactly one of two states, alive or dead, and whether or not a cell "survives" (or is "revived") in the next generation is based on neighbourhood rules that are intended to simulate real organic cell life.

These simple rules can create complex and often beautiful structures, but they typically die out when the automaton is left to run for long periods of time (hence its classification as a class 4 cellular automaton).

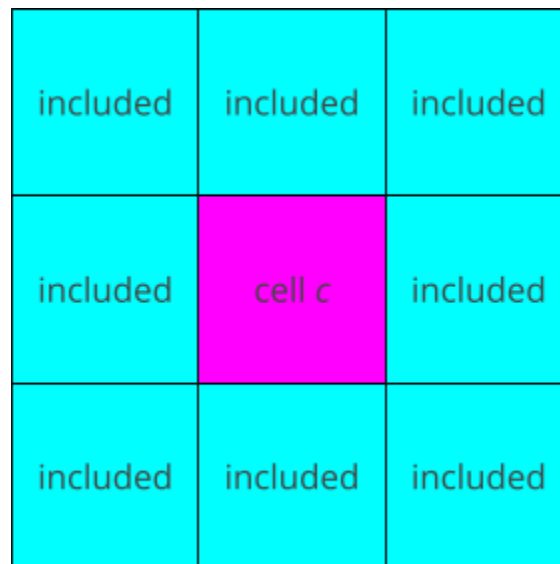
The Game of Life is defined with the following rules:

1. any live cell with fewer than two live neighbours in the current generation dies, as if by underpopulation.
2. any live cell with two or three live neighbours in the current generation stays alive in the next generation.
3. any live cell with more than three live neighbours in the current generation dies, as if by overpopulation.
4. any dead cell with exactly three live neighbours in the current generation becomes a live cell in the next generation, as if it were born.

To simplify these rules, they can be written in the form:

1. a live cell with fewer than two live neighbours or more than three live neighbours dies.
2. a live cell with exactly two or three live neighbours lives on.
3. a dead cell with exactly three live neighbours becomes alive.

These rules can be written as B3/S23 in the notation discussed under [lifelike cellular automata](#).



*Figure 17: The neighbourhood definition of the Game of Life (a Moore neighbourhood of radius 1, as it is a lifelike cellular automaton).*

### 2.7.1: Characteristics of the Game of Life

The Game of Life is *outer-totalistic*:

- a live (or active) cell can be modelled as in state 1, and a dead (or inactive) cell can be modelled as in state 0.
- the neighbourhood rules depend only on how many neighbours a given cell has, the location of these neighbours does not matter.
- any cell  $c$  is not a part of  $c$ 's neighbourhood.
- therefore, by definition, the Game of Life is outer-totalistic.

The Game of Life is *non-reversible*:

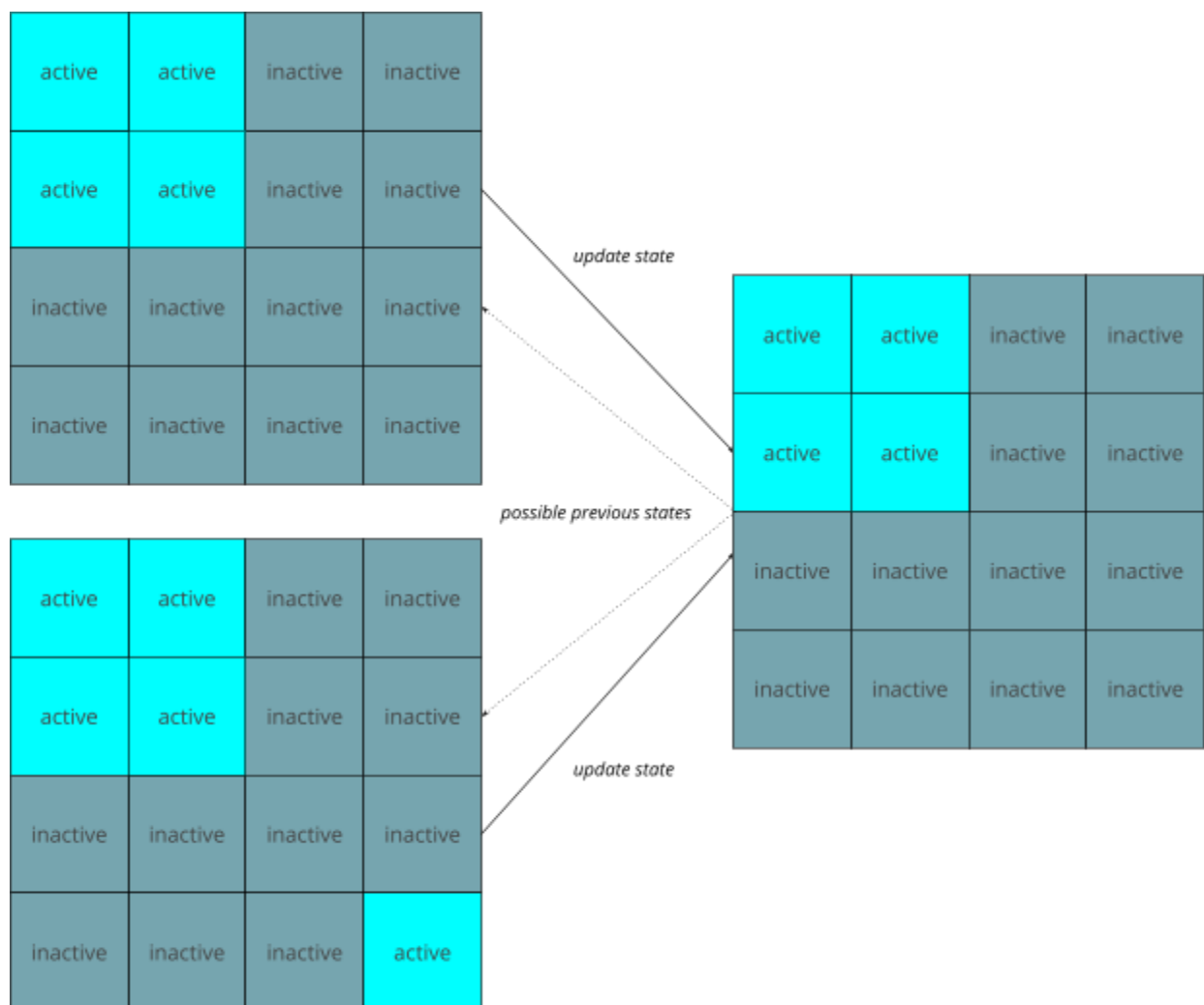


Figure 18: Proof that the Game of Life is non-reversible.



Consider Figure 18. The two states on the left both update to the state on the right (by definition of the rules of the Game of Life), which means that the state on the right has more than one preimage.

We have seen that for an automaton to be reversible, it must have exactly one preimage for every possible state of the automaton. Since this counterexample exists, the Game of Life is not reversible.

The Game of Life is in class 4:

- because the definitions of classes are qualitative in nature, it is difficult to formulate a proof in words as to how the Game of Life is in class 4.
- instead, it is best to run the program and simply see that the automaton creates complex structures that eventually die out into stable or oscillating patterns (as the definition of a class 4 automaton implies).

### 2.7.2: Interesting Patterns in the Game of Life

The Game of Life is well known for exhibiting many interesting and often beautiful so-called “life patterns”. These patterns are structures of live cells that are organised in such a way that they have special characteristics.

Some structures do not change through generations, and some structures oscillate between the same patterns over and over through generations. Some other structures actually move and propagate themselves through the grid of the cellular automaton (spaceships), and there are even structures that emit these moving structures (guns)!

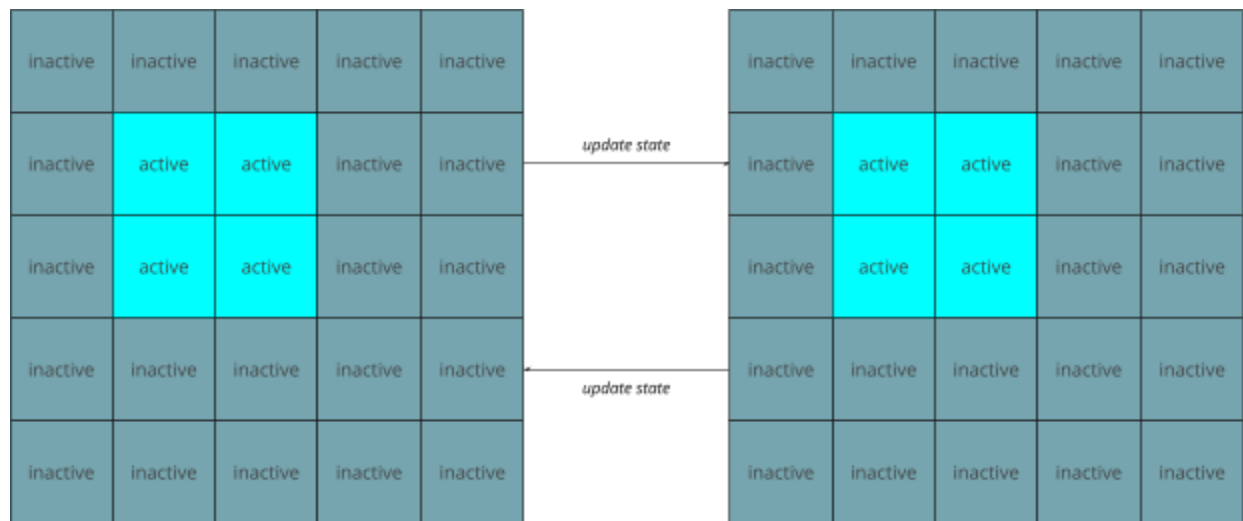
All these structures are categorised:

- “Still Life”
  - structures that do not change through generations.
  - these are in fact oscillators with a period of 1.
- “Oscillators”
  - structures that oscillate between the same patterns over and over through generations.
- “Spaceships”
  - structures that “move” through the automaton’s grid and propagate themselves through it.
- “Guns”
  - stationary structures that generate and emit spaceships.
- “Rakes”
  - moving structures that generate and emit spaceships - these are moving guns, or guns that are also spaceships.
- “Trains”
  - moving structures that leave behind a trail of live cells.

### 2.7.2.1: Still Life

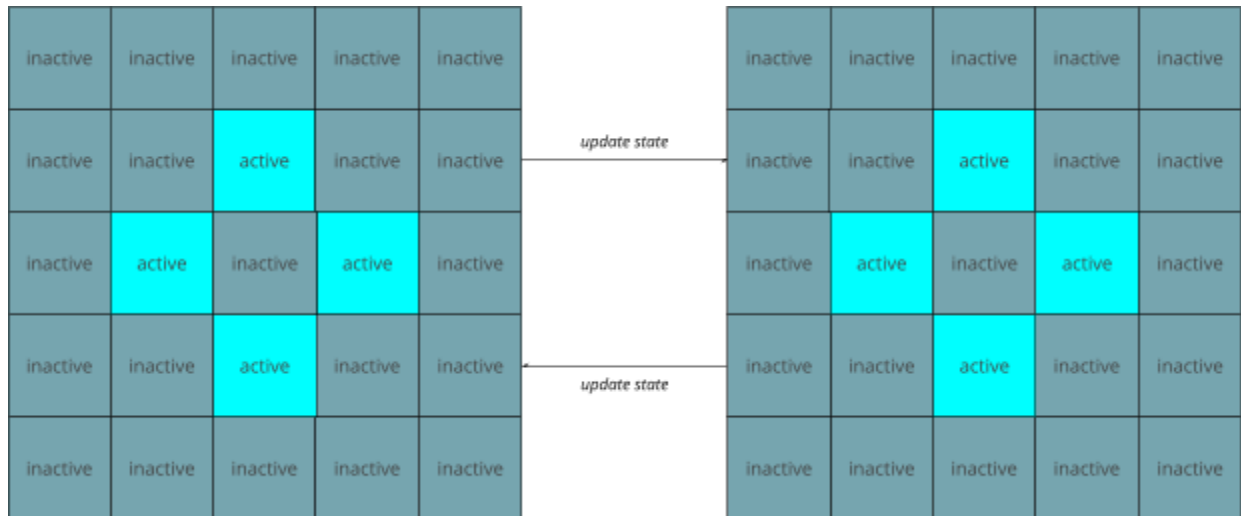
Still life structures are patterns of live cells that do not change through generations. In all still life structures, every cell will have two or three live neighbours.

Recall that in the Game of Life, if a live cell has two or three live neighbours, it lives on to the next generation. If all cells in a structure have this many neighbours (and no cells die or reproduce), then they will all live on, and therefore create still life structures.



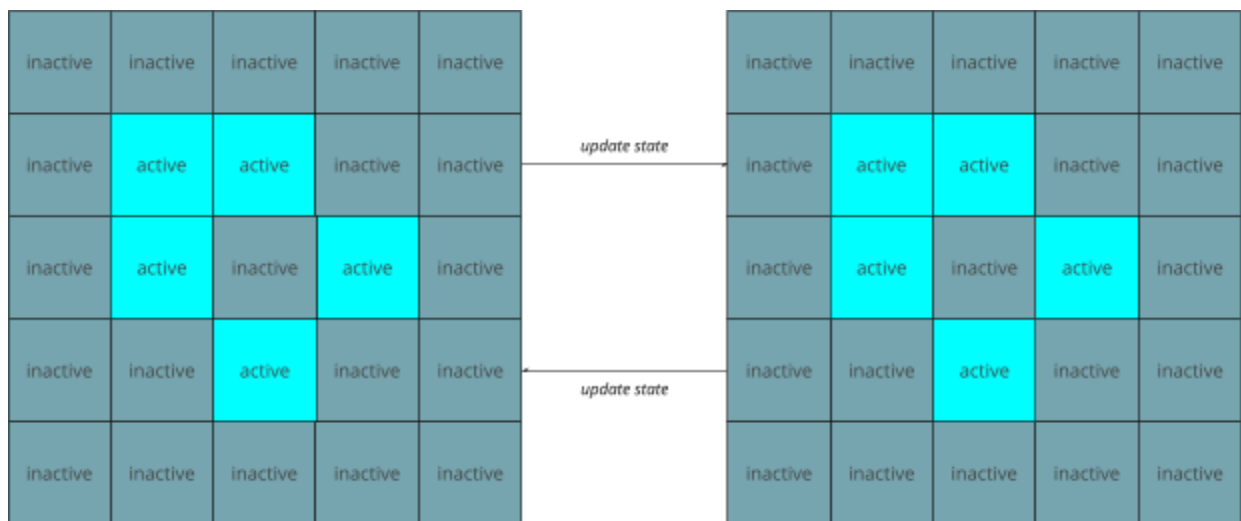
*Figure 19: The block, a 2x2 structure.*

The block is a common still life structure.



*Figure 20: The tub, a cross-patterned still life structure.*

The tub is another still life structure. All cells have two live neighbours.



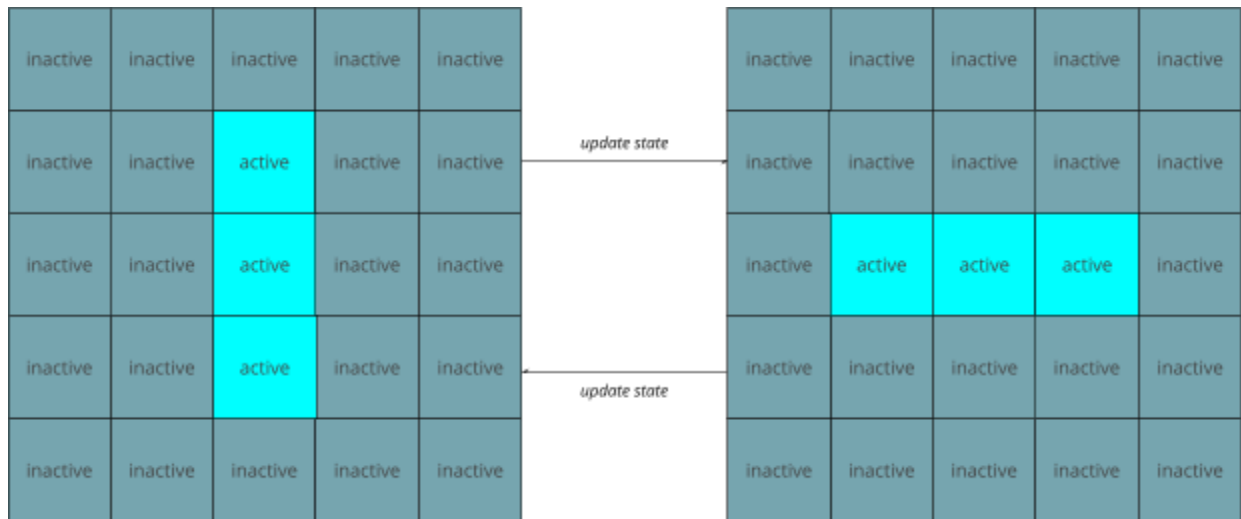
*Figure 21: The boat is the same as the tub, but with one more active cell.*

The boat is a structure where some cells have two live neighbours, and others have three live neighbours.

There are also some other, larger still-life patterns which are not pictured here (doing so would take up a lot of space!).

### 2.7.2.2: Oscillators

Oscillating structures are structures of cells that cycle between the same patterns over and over in the automaton for eternity.



*Figure 22: The blinker, an oscillating structure.*

The blinker is the most common oscillator. It cycles between the two patterns pictured above. Because it cycles between two patterns, it is said to have a period of two.



*Figure 23: The toad, another oscillating structure.*

The toad is another common period two oscillator.



*Figure 24: The beacon - an oscillating structure made up of two blocks connected diagonally.*

Finally, the beacon might be considered the third structure included in the three most commonly occurring oscillators in the Game of Life. Again, it has a period of two.

There are other oscillators that are larger and have longer periods, but these are best seen in the supplied program (because of their longer periods and great size). Also, it is worth noting that oscillators with periods greater than two are comparably rare.

### 2.7.2.3: Spaceships, Guns and More

Spaceships are structures that move through the automaton's grid.

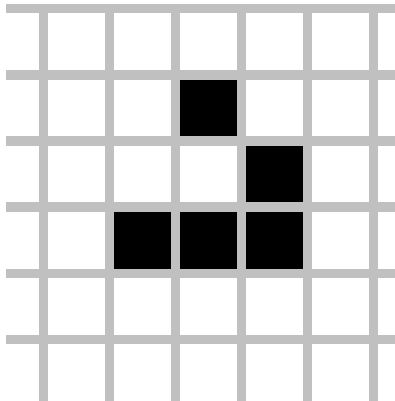
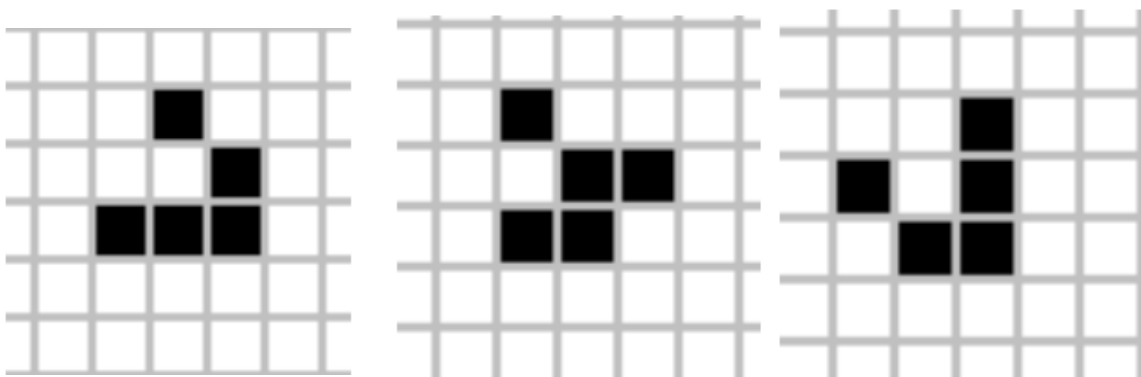


Figure 25: A glider<sup>4</sup> shown moving through the grid.

By far the most commonly occurring spaceship is the glider, shown in Figure 25. Gliders generate rather frequently during normal operation of the Game of Life, and are the best-known example of moving structures.

We can see that in Figure 25, the glider is moving through the grid of its own accord. This is a result of the rules of the Game of Life, allowing the glider to propel itself forward while maintaining its shape.

If Figure 25 is a still image:



---

<sup>4</sup> Rodrigo Silveira Camargo, Public domain, Wikimedia Commons,  
[https://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life#/media/File:Game\\_of\\_life\\_animated\\_glider.gif](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#/media/File:Game_of_life_animated_glider.gif)

The graphic above shows how the glider changes shape through generations, without the use of motion pictures.

The glider moves from the leftmost state to the rightmost state, and then from the rightmost state to the leftmost state. However, in every generation, the glider also moves diagonally one square. In this case, the glider is moving diagonally down and to the right.

To view the glider in actual motion, please visit the source.



It's also possible that gliders can be generated by *guns*. Guns are stationary structures that emit spaceships.



*Figure 26: A Gosper's Glider Gun<sup>5</sup> emitting gliders.*

The structure at the top generates moving structures (spaceships) that collide with one another. When this collision occurs, the cells are organised in such a way that the structure of a glider is created and released.

After the collision occurs, moving structures are sent back to the blocks and the cycle is repeated.

Again, if the still image is not enough to visualise the structure, please visit the source to see how the gun appears in simulation.

---

<sup>5</sup> By Lucas Vieira - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=101736>

There are also types of moving guns - these are rakes. While trains are structures that leave behind a trail of live cells, rakes are a special form of trains that leave behind a trail of spaceships.



*Figure 27: A selection of rakes<sup>6</sup>.*

Figure 27 shows a selection of rakes. Though they are quite small, multiple moving structures are seen that leave behind a trail of spaceships.

While rakes leave behind a trail of spaceships, there are other moving structures that leave behind non-moving structures. These types of structures are called trains or puffer trains.

It would be exceptionally rare for rakes, trains or guns to occur naturally from a randomly set initial state of the Game of Life. Therefore, it is best to create them in order to see them in action.

However, gliders, oscillating structures and still lifes do appear with good frequency randomly. Even still, larger spaceships (than the glider) and oscillating structures with long periods are unlikely to appear during a randomly-initialised run of the Game of Life.

Again, if the still image is not enough to visualise the structure, please visit the source to see how the rake appears in simulation.

---

<sup>6</sup> Simpsons contributor, Public domain, Wikimedia Commons, [https://commons.wikimedia.org/wiki/File:Rake\\_selection.gif](https://commons.wikimedia.org/wiki/File:Rake_selection.gif)

### 3: Project Requirements

Requirements were specified before development began. They were as follows:

1. There will be a function that allows the user to toggle the state of cells on the grid.
  - a. These cells can be drawn individually, and can also be “stamped” on with a few predefined life patterns, such as a glider.
  - b. Stamping is only applicable for turning cells on, not off.
2. The boundaries of the grid will wrap, so that a life pattern passing through any edge of the grid will reappear on the opposite edge.
  - a. This feature can be disabled.
3. All live cells can be cleared at once during the simulation.
4. The update rules (the neighbourhood rules of the program) will be customisable by the user, up to a certain reasonable limit.
  - a. Update rules will be limited in size to prevent “misuse” of the program - there is no reason for a user to be able to simulate a 2D cellular automaton with neighborhoods the size of the entire grid, for example.
5. The speed of the program should be controllable, perhaps with 0.5x speed, 1x speed, and 2x speed.
6. The user should be able to pause, play and single-step the simulation.
7. The number of cells in the simulation should be customisable before the simulation starts.
8. The starting state will be able to be initialised in several ways:
  - a. This includes a randomised start, a clear grid start and some “interesting” start states that feature oscillating life patterns.
  - b. A state of the automaton can be saved and later loaded to initialise the automaton again.
9. As cells age, they will be able to change colour to reflect how long that they have survived. When a cell dies, its colour will be reverted should it ever become alive again.
  - a. If a cell at location  $s$  dies in iteration  $i$ , but is reproduced in iteration  $i+2$ , that cell will appear as the “normal” colour.

- b. For this reason, the colour of cells will not be customisable - however the colour of the grid and/or background may still be customisable.
  - c. This feature can be disabled.
- 10. All features should be realised through a user interface, and a command-line input will not be necessary for the program to run.

All of the above ten specified requirements have been met in the implemented program.

- **Note for Requirement 4:** the “certain reasonable limit” was decided to be all lifelike cellular automata. When the project began, it was unknown how many cellular automata would be supported. The class of lifelike automata was ultimately selected to limit the scope of the project to a manageable amount.
- **Note for Requirement 5:** a slider was implemented to allow more than just three options for speed.
- **Note for Requirement 7:** the number of cells is customisable in the way that the user can choose a width and height, both in pixels, and can choose the size in pixels of a single cell. The number of cells follows as a function of those three parameters.

### 3.1: System Requirements

1. The program must be able to run on Windows 10.
  - a. The program will most likely work on Mac, Linux and older versions of Windows as well, as it will be developed using tools that are available on these platforms.
  - b. It will not be tested on these platforms, however.
2. The program will run smoothly on sensible hardware specifications.
  - a. This means that testing the program on very old hardware will not be necessary.

The system requirements criteria have been met. The program runs with a low CPU load and a low memory load, and is fully tested on Windows 10.

- **Note for Requirement 2:** the “sensible hardware specifications” demand a minimum resolution of 1280x720. [Also, nVidia's G-Sync should be disabled.](#)

## 4: Design

The project planned to use Java, for its versatility and efficiency. Other suitable options were considered, including video game engines and other programming languages.

Ultimately, it was concluded that the best choice to implement as many features as possible in the time frame provided was Java.

The main aim of the program is to simulate lifelike cellular automata in a fun and interesting way, even to users who do not understand such automata.

## 4.1: Development Approach

The project followed a rapid prototyping approach. In total, twenty-three prototypes were made and an additional one final package was made.

Whenever a feature was to be added, a new prototype was created. For instance, the first prototype only simulated Conway's Game of Life and supported no user input. The prototype that followed would add play / pause functionality, and the prototype that followed that would add another distinct feature.

Rapid prototyping was effective in ensuring that there was always a fall-back in case the project did not go to plan. If, for any reason, not all requirements were able to be met, there would at least be a fully-functional program available to deliver.

Moreover, it allowed for features to be implemented in any order: if a feature was too difficult to implement, it could always be postponed while development continued on a different part of the program. This was the case for the rule changing functionality, which was one of the last prototypes made.

A drawback of this development approach is that there were few time-based milestones. The idea was to complete as many requirements as possible before the deadline, but there was no Gantt chart to track progress or to define any type of rigid schedule.

Overall, the rapid prototyping approach proved highly effective for a solo development approach. While it lacked direct structure, this was a non-issue, as personal calendaring kept the project on track to be submitted on time.

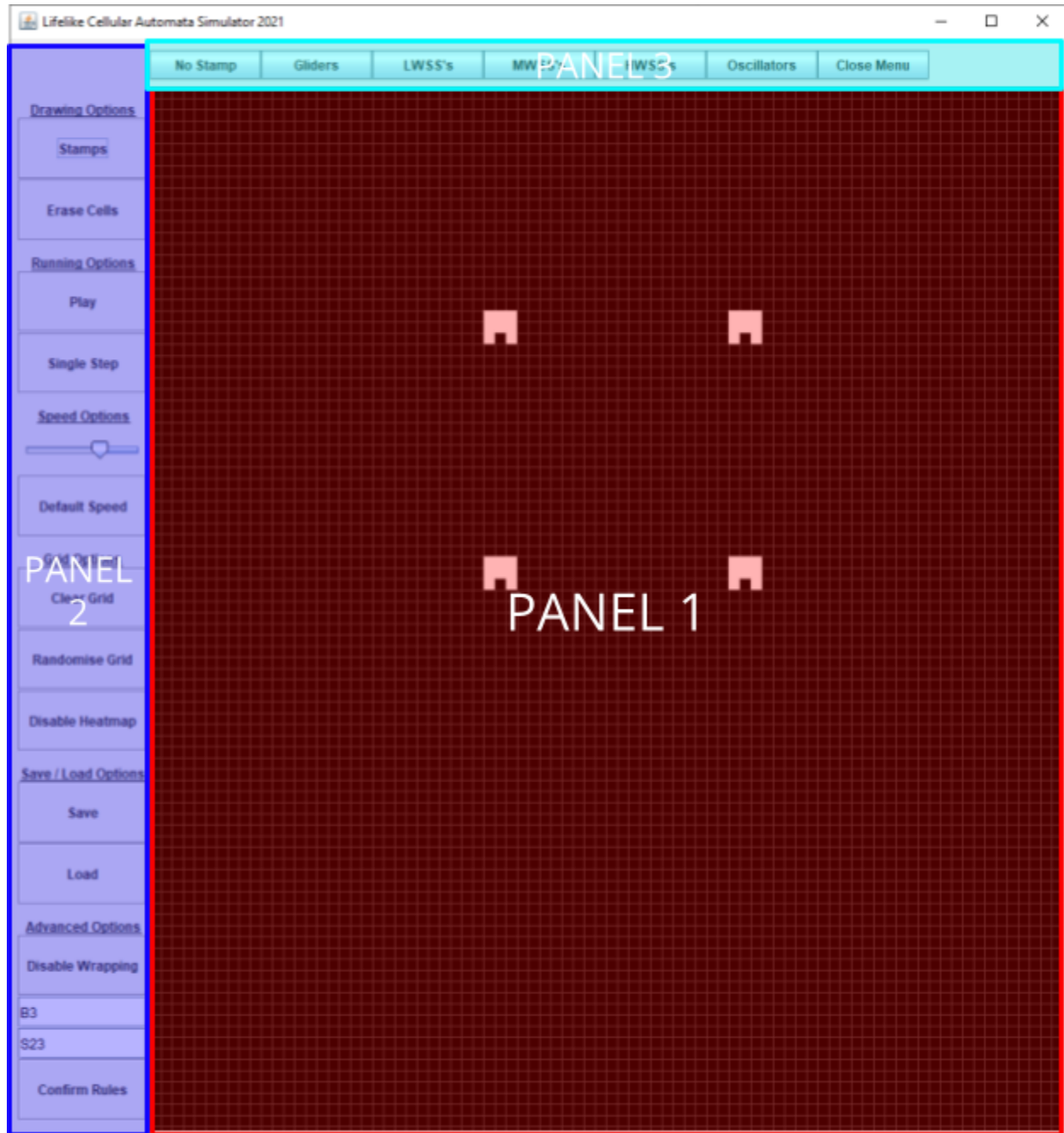
## 4.2: Panel Design (Overall Layout Design)

The program consists of four major panels:

1. Panel 1 is a *graphics panel*, which displays the cells of the automaton using pixels.
2. Panel 2 is a *toolbar panel*, which displays the buttons that the user interacts with. The toolbar panel is not a standard panel, but rather a `JToolBar`.
3. Panel 3 is a *stamp panel*, which displays the buttons that correspond to choices that the user has between stamps to use. The stamp panel's visibility is controlled by the toolbar panel. The stamp panel is not a standard panel, but rather a `JMenuBar`.
4. Panel 4 is an underpinning *content panel* that the other three panels use to define their location on screen.

The four panels are all placed onto a frame (a `JFrame`), which is the frame that programs appear on (at least when using Java and `WindowBuilder`). A `JFrame` is analogous to what all programs would appear on when running on a given operating system, which houses the minimise, restore and close buttons.

There is also a dialog box (`InitialPopup` in program code) that shows up when the program first starts, which follows design principles from the main program.



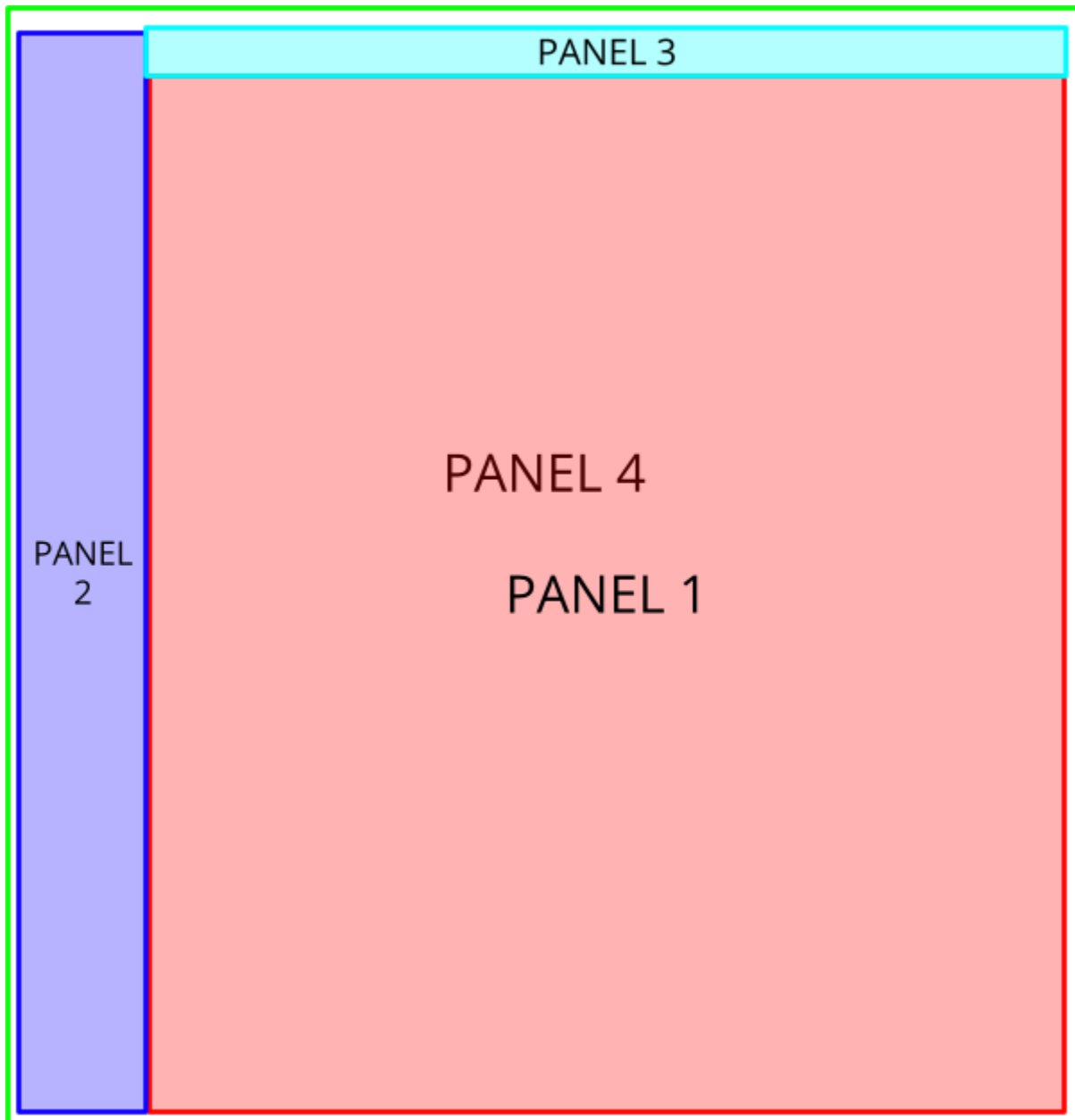
*Figure 28: A screenshot of the finished program, with three panels highlighted.*

Figure 28 shows how three of the four panels of the main program are placed. The program consists of four major panels, which are all placed onto one program frame.



The program frame is what contains the operating system-level controls, and the title of the program (“Lifelike Cellular Automata Simulator 2021”). The program frame is the JFrame of the program, and its controls and buttons are handled by the operating system of the machine being used to run the program.

The content panel, panel 4, is underneath the three panels highlighted in Figure 28.



*Figure 29: Depicting the layout of all four major panels, including the content panel that underpins the remaining three panels.*

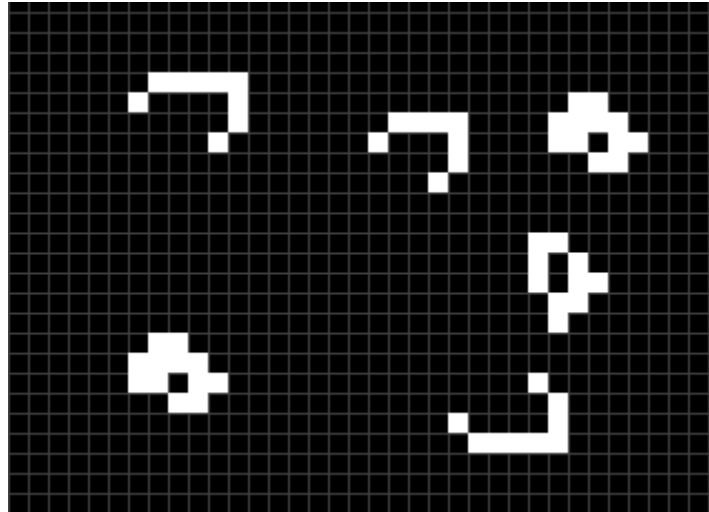
Panel 4 (green in Figure 29) underpins the three other major panels. That is, panel 2 is defined to appear on the western edge of panel 4. Panel 3 is defined to appear on the northern edge of panel 4. Finally, panel 1 is defined to appear in the centre of panel 4, while still respecting the position of panels 2 and 3.

This design layout was chosen before the prototype that first included panel 2 was made. Before this point, there was no planned design. This is another advantage of the rapid prototyping method: it allowed the project to remain agile and easy to manage.

After this layout was decided, the design of panel 2 was created.

### 4.3: Graphics Panel Design (Panel 1)

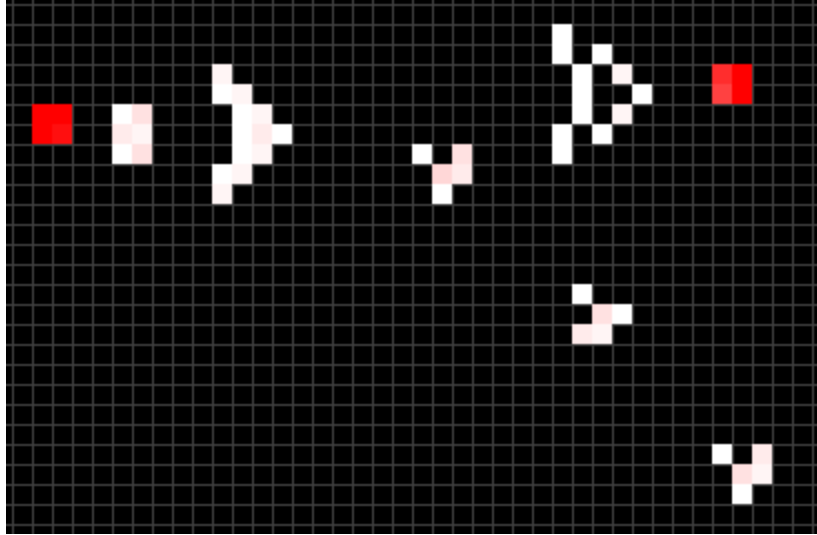
Since lifelike cellular automata have exactly two states for each cell, a simple two-colour scheme of black (for inactive cells) and white (for active cells) was chosen.



*Figure 30: A screenshot of the program simulating Conway's Game of Life, with live cells drawn in white and dead cells in black.*

A black background was chosen, and live cells would be drawn in white. Since a grid of cells was needed, gridlines were needed too. Gridlines were made gray so that they were visible around inactive cells without being bright enough to divert attention from active cells.

Although cellular automata have two states, a third colour was needed to support the heatmap functionality. That is, as cells age, they become coloured.



*Figure 31: A screenshot of the program with some cells redder than others.*

In Figure 31, the blocks are almost completely red. This indicates that they are old, and have been alive for quite a few generations. Similarly, the gliders have pink cells, which indicates that although they are not new, they are still not as old as the reddest cells.

Red was chosen for three main reasons:

1. Red is a de facto colour for heatmaps.
2. Red colour is highly different to the white colour of a normal cell, making it easy to identify how old a cell generally is.
3. Red and black (of the dead cells) together form a visually pleasing colour combination.

The graphics panel of the program was written entirely by hand. That is, no external tools were used (like WindowBuilder) to create the graphics panel.

This is simply because there are none available (as far as the planning phase found). The graphics panel is, by nature, a bespoke component of the program.

In its simplest definition, the graphics panel is an empty shape with a background colour (in this case, black). Every frame, grid lines are drawn<sup>7</sup>, and some parts of the screen are drawn over the black background to make certain cells of the grid appear white (or some shade of red).

This is the case as it is the most efficient graphics solution available, instead of directly modelling parts of the screen as actual cells. Directly indexing parts of the screen would be both inflexible for multiple resolutions and resource consuming to implement.

Java has no graphics tools that automate this drawing process, as far as the planning phase found. Any tools that may have been suitable were deemed more complex or less efficient than using nothing.

#### 4.3.1: Animation Design

The default animation delay is 75ms. That is, every 75ms, a new generation of the automaton being simulated will be drawn.

A 75ms delay gives approximately 13 frames per second. Commonly, film runs with 24 frames per second, which results in a smooth motion picture.

The slowdown to 13 frames per second makes it clear to see how generations are progressing, but without making the animation seem choppy. The goal was to make the program simulate a cellular automaton, not appear so smooth as to allow the human eye to blend frames together, missing important details that would not be considered important in a film setting<sup>8</sup>.

---

<sup>7</sup> Grid lines are drawn every frame as they are a more flexible solution than hard-drawing the grid. It allows for more sizes of cell and grid to be more easily supported.

<sup>8</sup> Furthermore, the user always has the option to speed up the simulation to up to 100 frames per second (10ms delay), or slow it down to as little as 5 frames per second (200ms delay).

## 4.4: UI Design and UX Design (Panels 2 and 3)

The GUI elements of the program (i.e. excluding panel 1) were planned to be implemented using [WindowBuilder](#) for Java. WindowBuilder was chosen for its simplicity and its lightweight nature, as well as its platform-independency.

Panel 2 was designed before the first prototype that included it was developed. The overall idea was to include buttons in a logical order. In the final prototype, these buttons were placed into button groups following user feedback recommending to do so.

Panel 3 was similarly designed just before the first prototype that included it was developed.

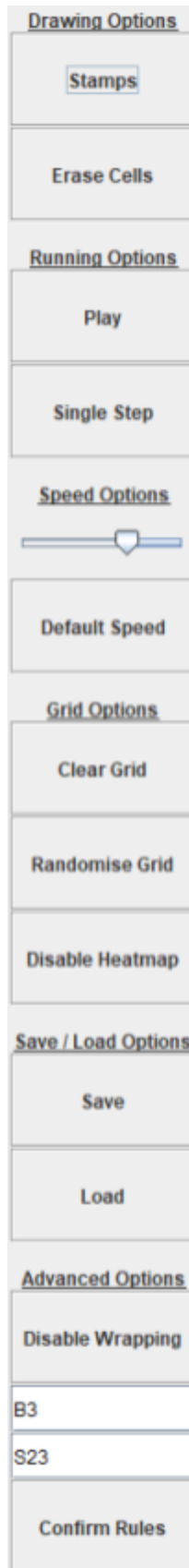
Still, [as previously mentioned in Panel Design](#), the layout for where the panels would be located was decided just before the first prototype that included panel 2 was developed.

#### 4.4.1: UI / UX Design of Panel 2 (Toolbar)

The toolbar is a JToolBar object, which are objects designed to house buttons and controls that the user can interact with to influence the rest of the program. Hence, they were the obvious choice for the type of panel to use for the toolbar panel.



Figure 32: Panel 2 (toolbar, left) and panel 1 (graphics, right) shown together.



As seen in Figure 33, the toolbar panel is very simple.

It is important to note that the program uses WindowBuilder's System Look'n'Feel functionality. This means that the program will appear differently on different operating systems; in the way that it will match the general design of the operating system it is being run on.

The program here matches the default Windows 10 appearance. When being run on MacOS, the program will appear to match system dialogs and windows of MacOS (due to COVID-19, it was not possible to secure a MacOS machine to retrieve screenshots).

This choice was made to further simplify the UI design - making this choice means the program will always fit the computer of the user, minimising the amount of UI work required.

The buttons on the toolbar are simple, large and clear. They are designed to be as obvious as possible. For instance, it is very obvious what the Play / Pause button does - it plays and pauses the simulation.

Furthermore, buttons are grouped in a way that is designed to be as obvious as possible. Every group is obvious in what it does, save for the "Advanced Options" group.

Buttons in the "Advanced Options" group are designed for use by advanced users, and what they do will not be immediately apparent to a casual user. This is more of a UX design choice than it is a UI design choice.

Figure 33 (left): The toolbar panel.



Figure 34 (right): The toolbar panel as in Figure 33, but with options flipped (e.g. "Play" becomes "Pause").



“Stamps” is in the same category as “Erase Cells”. “Erase Cells” toggles between the user drawing live cells when using their mouse, or erasing live cells when using their mouse. This is seen in Figure 34, where the button has been flipped and now displays “Draw Cells”.

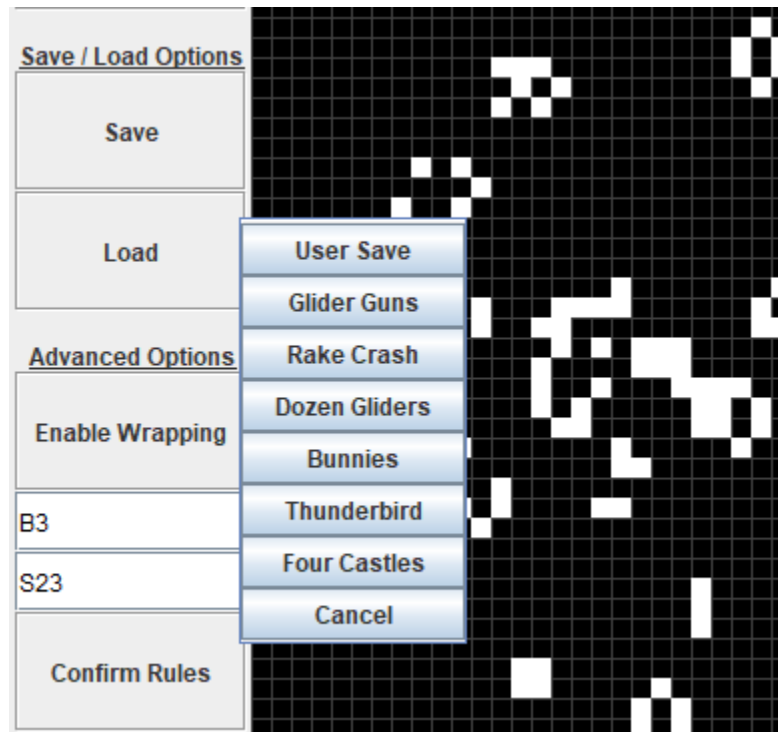
The category as a whole is named “Drawing Options” to reflect the fact that both buttons influence what the user does when they draw on the screen using their mouse.

This theme continues for all categories, such as “Save / Load Options” controlling saving and loading operations, and “Running Options” controlling how many generations that the program simulates per unit time.

The “Advanced Options” group demands that the user have some knowledge of cellular automata. The user can enable or disable wrapping, which changes the grid of the graphics panel to behave as a 2D grid or as a 2D torus.

Advanced users may also want to change the rules of the lifelike cellular automaton being simulated. It uses the de facto format discussed under [Rule Representation for Lifelike Cellular Automata](#).

#### 4.4.1.1: Popup Menus (Panel 2, Toolbar)



*Figure 35: Popup menu for loading files.*

When the “Load” button is used, the popup menu in Figure 35 is shown. The popup menu is a type of JPopupMenu object, and thus WindowBuilder uses a different button style than on a JToolBar, similarly to how Windows 10 would display menus and toolbars differently from each other.

The popup menu in Figure 35 gives seven choices to load (and a cancel option), which include all seven options included with the program. The buttons contrast with the toolbar enough to draw attention to them, but not so much to distract from the main simulation.

This popup design is used whenever a popup menu is required.

A limitation of this design is its expandability. If several dozen load configurations were created, then this popup menu would become cumbersome and ineffective. Fortunately, for the amount of load files that currently exist, this is a non-issue.

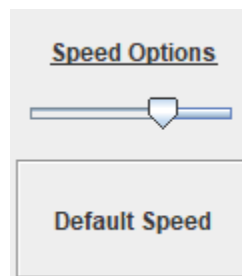
However, if the program were to be expanded, it would be wise to replace this popup menu with a file browser. The user would interact with the “Load” button,

and a file browser would be presented to them, asking them to select a file to load from.

This would require the program to check save files before loading them in some way. While this feature was never planned in the project requirements, it was suggested by several users. At this point, it was unfortunately too late to implement.

When the “Save” button is interacted with, the “User Save” element of the popup in Figure 35 will load whatever configuration the automaton was currently running when the “Save” button was used.

#### 4.4.1.2: Sliders (Panel 2, Toolbar)



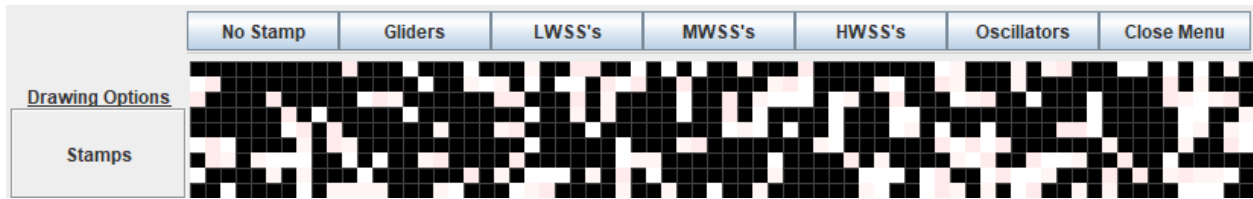
*Figure 36: The speed slider.*

The goal, again, is simplicity. The slider here is inverted (high values are on the left). This means that when the user drags the slider to the left, the simulation slows down. This choice was made as it makes logical sense that the lower end of the slider should signify a lower speed.

The slider itself has no number that displays the current update speed. This choice was made to dissuade users from looking for “nice numbers”, like 100 or 50, and instead to look for what value the slider should be at to present an appealing animation speed to them.

Because there is no number, it may be difficult for a user to return to the default speed that the simulation runs at. When the “Default Speed” button is used, the slider returns to its default value of 75ms.

#### 4.4.2: UI / UX Design of Panel 3 (Stamps)



*Figure 37: The stamp panel, shown above the graphics panel and to the right of the toolbar panel.*

The design decisions for panel 2 generally follow into the design decisions for panel 3. Panel 3 was designed after panel 2 was designed.

Panel 3 (the stamp panel) only appears when the “Stamps” button on panel 2 is interacted with. Otherwise, it is hidden. When the user selects a stamp to use, the panel is hidden from view. The panel is hidden by default to maximise the amount of screen space that is available for drawing the simulation.

The stamp panel is a JMenuBar object, under the context of WindowBuilder. It is a menu that displays several options. Each button has its own popup menu that shows up when it is activated, and the popup menu design follows from the popup menu in panel 2.

Because the stamp panel is a menu object, it uses the same button style as the popup menu for the “Load” button (that is shown in [Figure 35](#)).

The buttons are named after the patterns that they house. The “Gliders” button houses all four orientations of a glider pattern, and similarly for “LWSS’s”, “MWSS’s”, and “HWSS’s”. The “Oscillators” button displays a short selection of different oscillating patterns in Conway’s Game of Life.

A close menu option is presented, although the user can click the “Stamps” button on panel 2 (the toolbar) to close it. They can also use “No Stamp” to draw single cells.

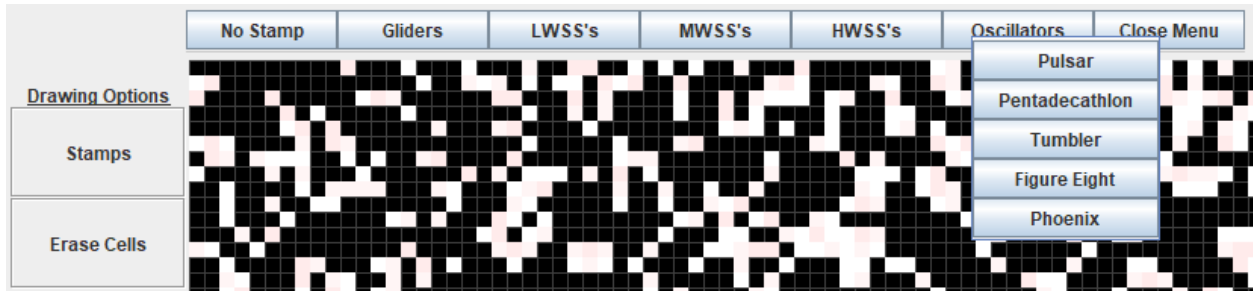


Figure 38: The popup menu for the “Oscillators” button in the stamp panel.

When the oscillators button is interacted with, a popup menu that shows five options is presented. It follows the same design as the popup menu for the “Load” button that is shown in [Figure 35](#).

However, no “cancel” button is given. This is because the user can click anywhere off the menu to close it, including any other button. A cancel button here would be an unnecessary use of space, as selecting a stamp is not damaging to the state of the automaton in the way that loading a file is.



Figure 39: The result of clicking on the screen after the “Up-Right” option of “Gliders” is selected.

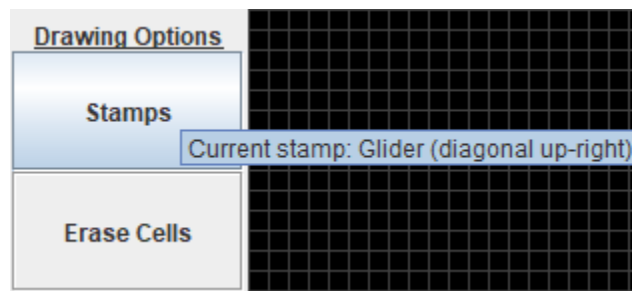
When the user selects a stamp by choosing a category, such as “Gliders”, and then choosing a specific choice using the popup menu, such as “Up-Right”, the program draws the chosen pattern onto the screen whenever the user clicks on the screen with their mouse.

The entirety of panel 3 is designed to be easily understood after using it once. When the user runs the program, they can easily understand how panel 3 works by simply using it (and similarly for panel 2).

The only thing for a user to learn about (or rather discover) is what each stamp pattern looks like. That is, the menus are designed to be as clear as they can.

#### 4.4.3: Tooltips

Tooltips are used to further describe the functions or current state of buttons in the program.



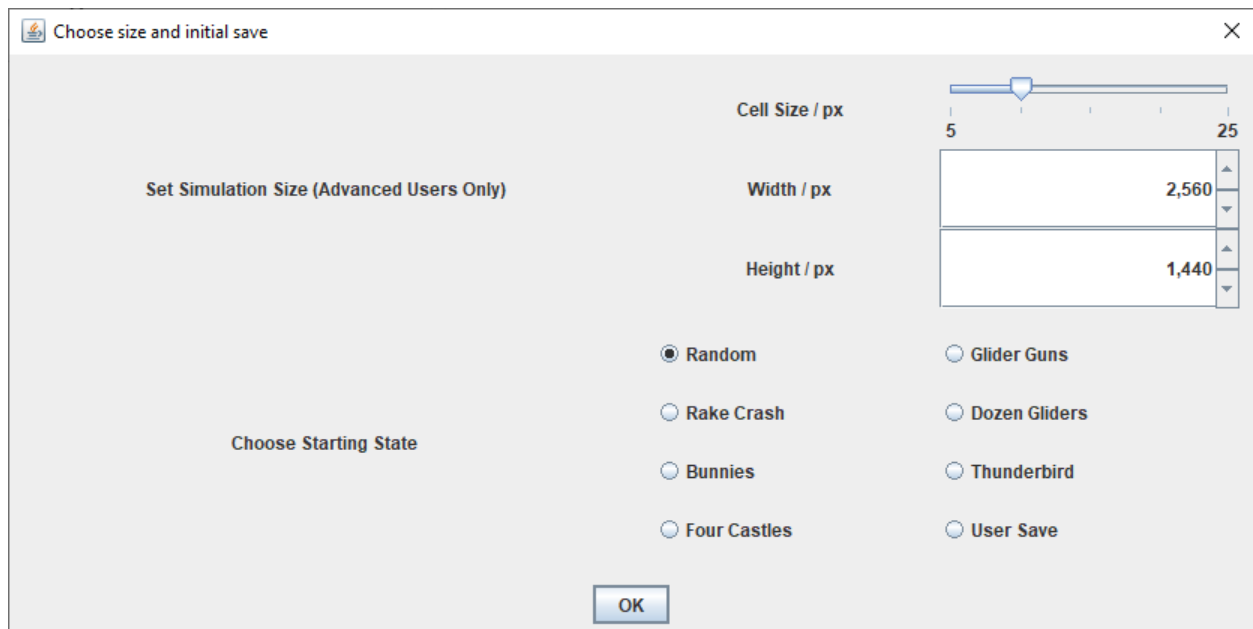
*Figure 40: A tooltip for the “Stamps” button on panel 2 (the toolbar panel).*

The tooltip shown in Figure 40 shows the user what the currently selected stamp is. In this case, the user would draw a glider that moves diagonally up and right when clicking on panel 1 (the graphics panel).

## 4.5: InitialPopup Design

The design for this Swing JDialog object was made just before the first prototype that included it was developed. Since this dialog box was actually created after all functionality in the main program was developed, its design principles follow from those of panel 1, panel 2 and panel 3.

The overall idea was to make the dialog simple and straightforward.



*Figure 41: Initial dialog box that shows when the program runs.*

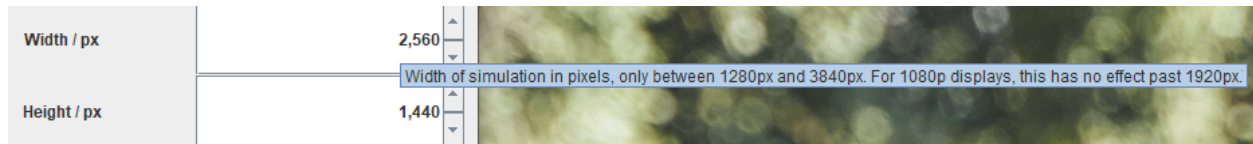
When the program is started, the dialog box shown in Figure 41 is presented to the user before the simulation starts.



*Figure 42: The size options of Figure 41.*

The size options are all logically measured in pixels. Only five different cell sizes are presented to ensure that the edges of the grid will always match to the edge of the program's boundaries.

Tooltips are also implemented here to communicate the minimum and maximum values for Width and Height.



*Figure 43: The width tooltip.*

This tooltip tells the user that the minimum value is 1280px and the maximum value is 3840px (i.e. a 4K resolution screen's width in pixels). The minimum value exists to prevent the user from making the simulation unusually small by accident.

If the user is using a 1080p display, the maximum value will be 1920px (the width of a 1080p display in pixels).

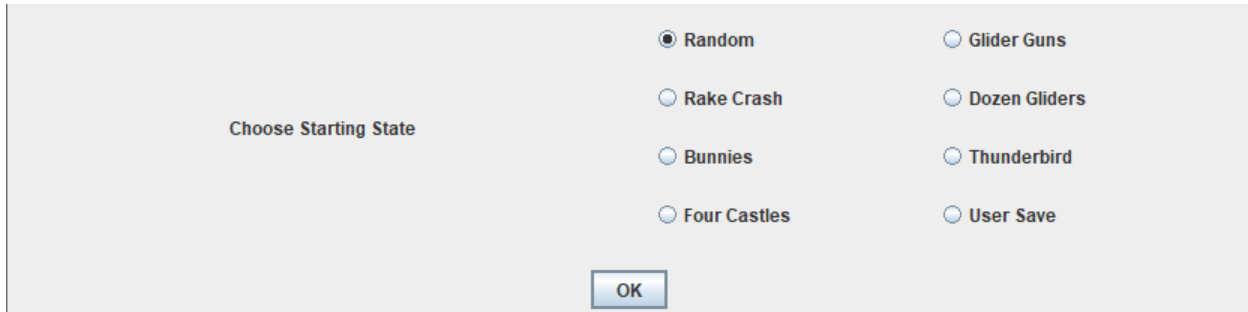
If the user enters an invalid value, it will reset to the previous value. The user can only enter numbers. Furthermore, they are able to use the up / down arrows to increment or decrement the value by 1, or they can type directly into the box.

The default value of the width and height boxes is the width and height of the monitor that the user is currently running. This has not been tested on multi-monitor setups, but according to documentation, it should work as expected.

In Figure 43, the program is being run on a 1440p display, so the default values are 2560px and 1440px for width and height respectively. If the display were 1080p, then the values would instead be 1920px and 1080px respectively.

The cell size is always defaulted to 10px, but can be any one of the five values shown (5px, 10px, 15px, 20px, 25px).





Choose Starting State

☒ Random      ☐ Glider Guns

☐ Rake Crash      ☐ Dozen Gliders

☐ Bunnies      ☐ Thunderbird

☐ Four Castles      ☐ User Save

OK

*Figure 44: The starting state options of Figure 41.*

The user can choose to initialise the program using any one of the save files supplied, including one that they have saved themselves by using “User Save”. They can also initialise the simulation randomly, where each cell is randomly determined to be active or not when the simulation starts.

This part of the dialog uses radio buttons that are mutually exclusive. The user can load any of these files during simulation if they wish, using the “Load” button discussed earlier as part of panel 2.

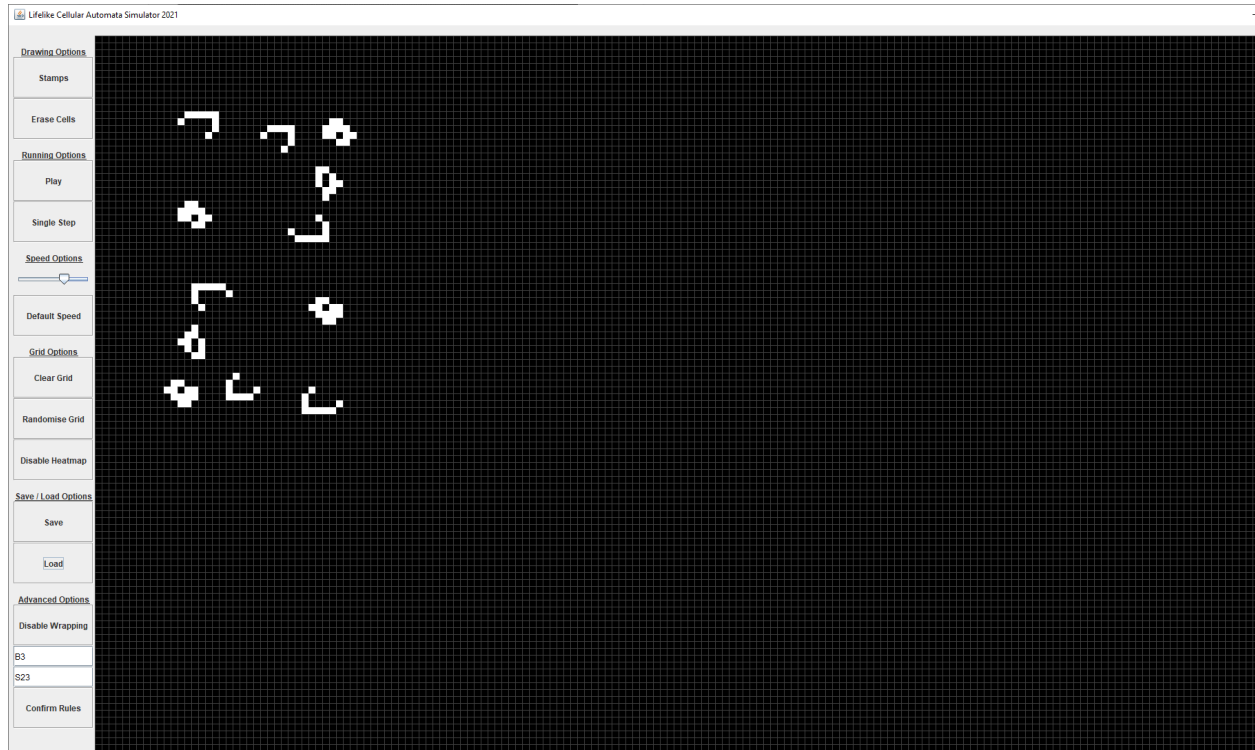
The “OK” button is self-explanatory: after all desired options have been set, interacting with “OK” closes the dialog and starts the simulation.



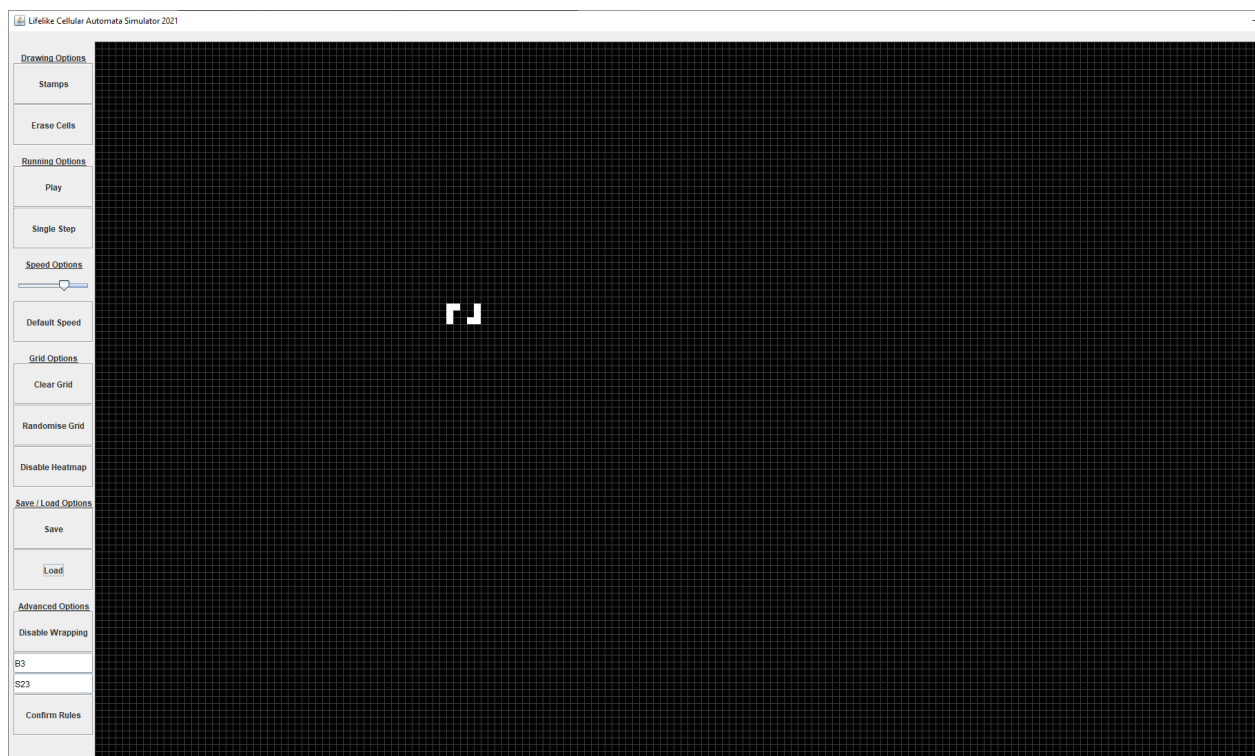
Figure 45: Random start state (panel 3 is hidden).



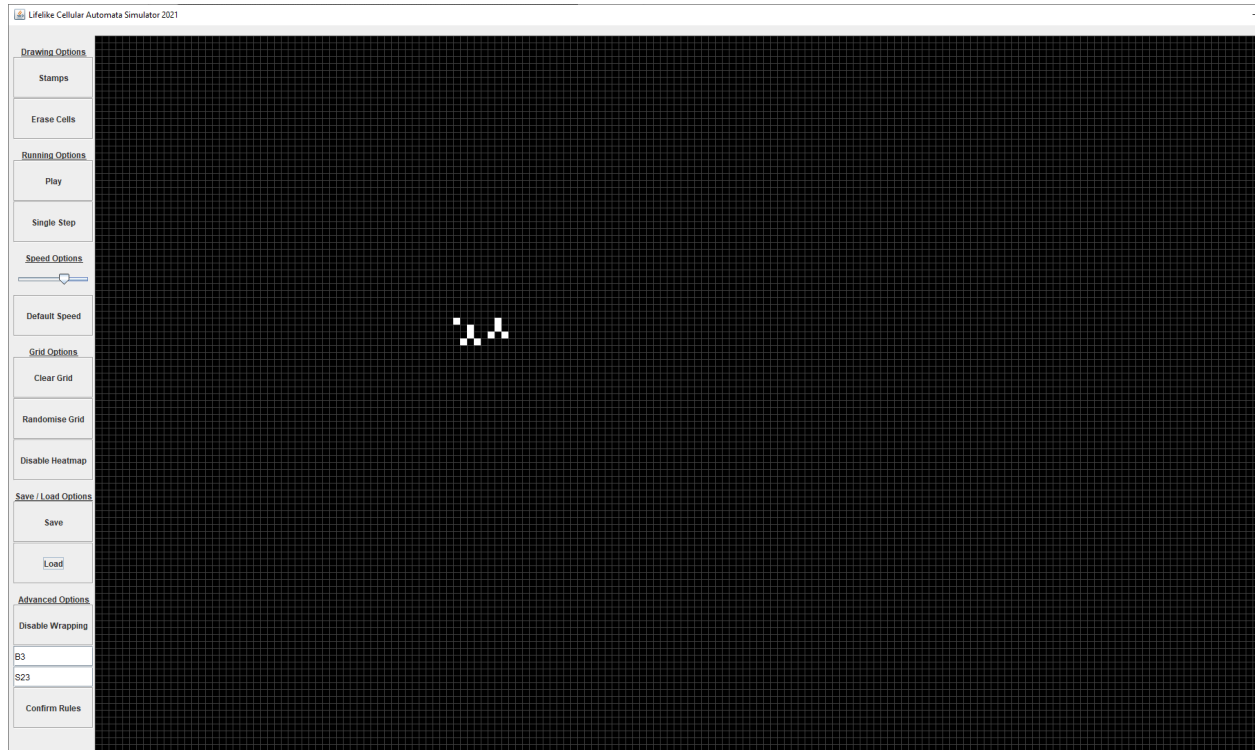
Figure 46: Glider Guns start state.



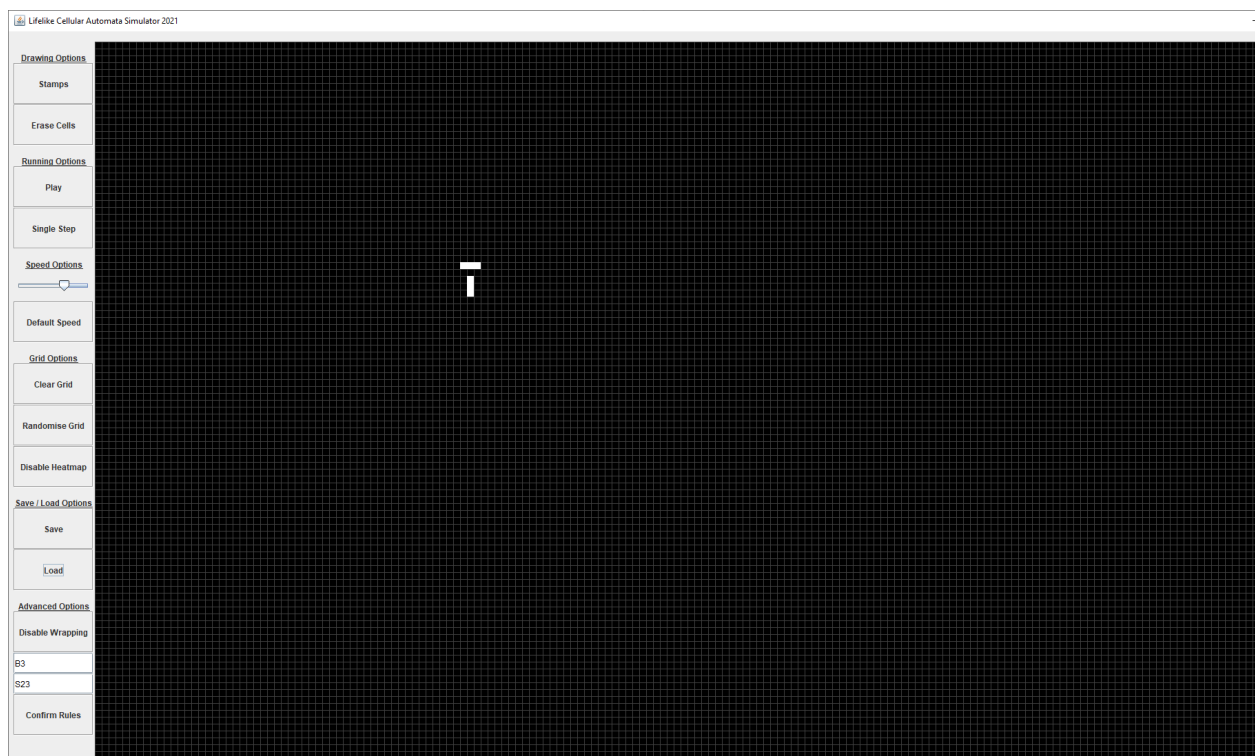
*Figure 47: Rake Crash start state.*



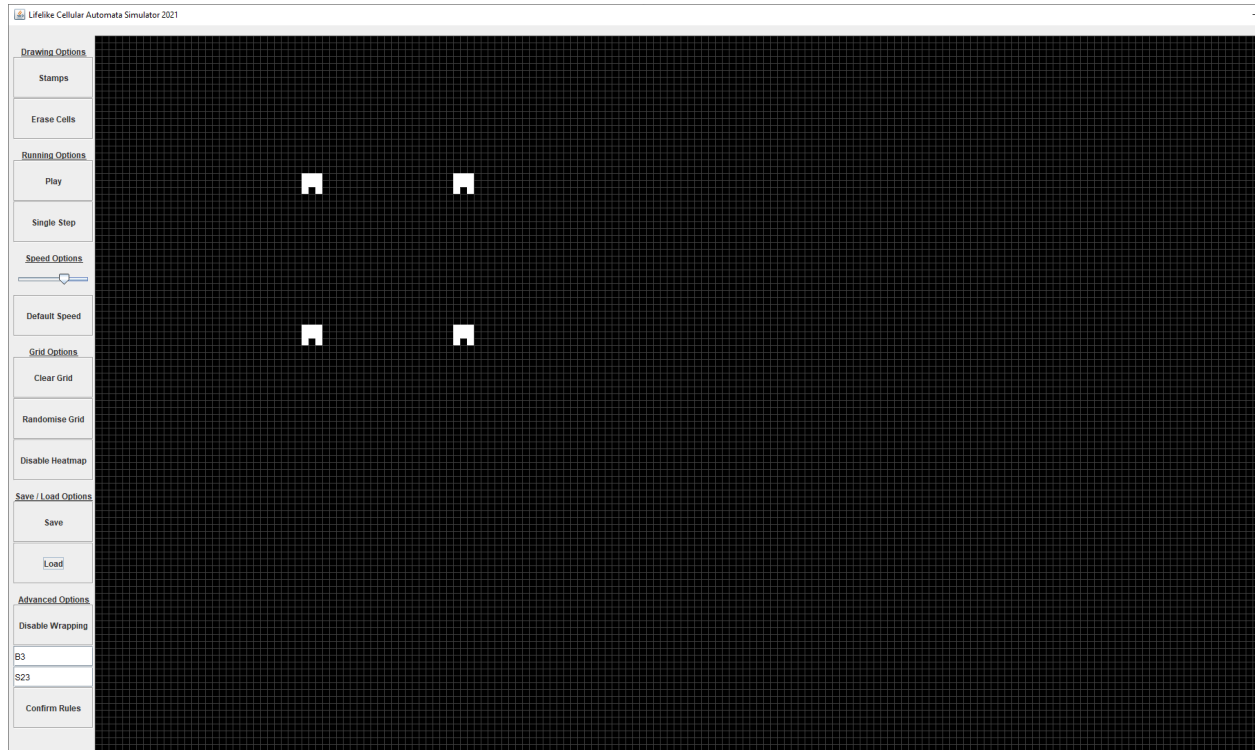
*Figure 48: Dozen Gliders start state.*



*Figure 49: Bunnies start state.*



*Figure 50: Thunderbird start state.*



*Figure 51: Four Castles start state.*

## 5: Implementation and User Manual

### 5.1: Program Structure and Class Diagram

The design section outlined that a rapid prototyping method was to be used to develop this program.

Java is an object-oriented programming language, but developing in a heavily planned object-oriented fashion would be contradictory to the concept of rapid prototyping. Creating large class diagrams and taking a highly planned approach for each prototype would not have been consistent with the concept.

As a result, no UML diagrams were created before any of the prototypes. Instead, after the last prototype was developed, one final class diagram was created to govern the development of the end deliverable.

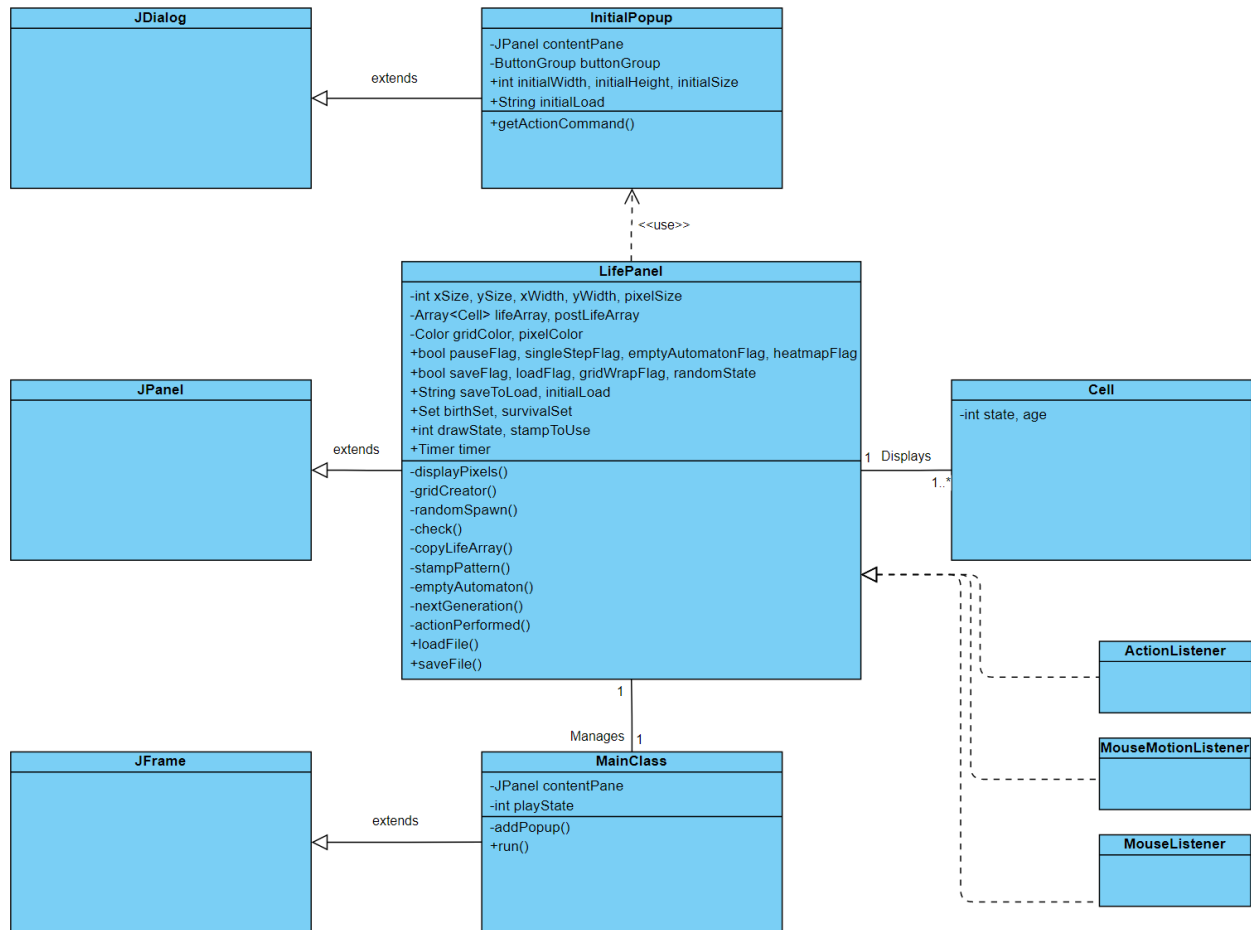


Figure 52: The class diagram for the program.

Figure 52 gives the class diagram for the program, which is also supplied [standalone](#) (both through the [Imgur](#) link and as a .PNG file included under the folder “Diagram”).

Imported classes are displayed as black boxes.

`InitialPopup` is the first window that the user sees when the program is run. `LifePanel` uses data from it in its constructor to determine its own size and initial configuration.

`MainClass` also uses `InitialPopup` to determine the size of the `JFrame`, but the size of the `JFrame` used is dependent on the size of the `JPanel` used. This explains why no association is directly present between `MainClass` and `InitialPopup`.

`LifePanel` displays and manages `Cell` objects by storing them in an array `lifeArray`. Since the simulation is based on cells and their states, `LifePanel` is essentially the heart of the simulation and the program as a whole. `LifePanel` maintains total power over all `Cell` objects, and can set their instantiated attributes to any value.

`Cell` objects have a `state` and an `age`. `state` is either 1 or 0, and if it is 1 then the cell is considered active. In this case, the cell will be drawn to the screen. Its `age` determines what colour it appears as when it is drawn.

`LifePanel` does implement some listener classes, which are used for mouse input. When the user draws with their mouse, live cells are drawn to the `LifePanel` by directly editing the `lifeArray`.

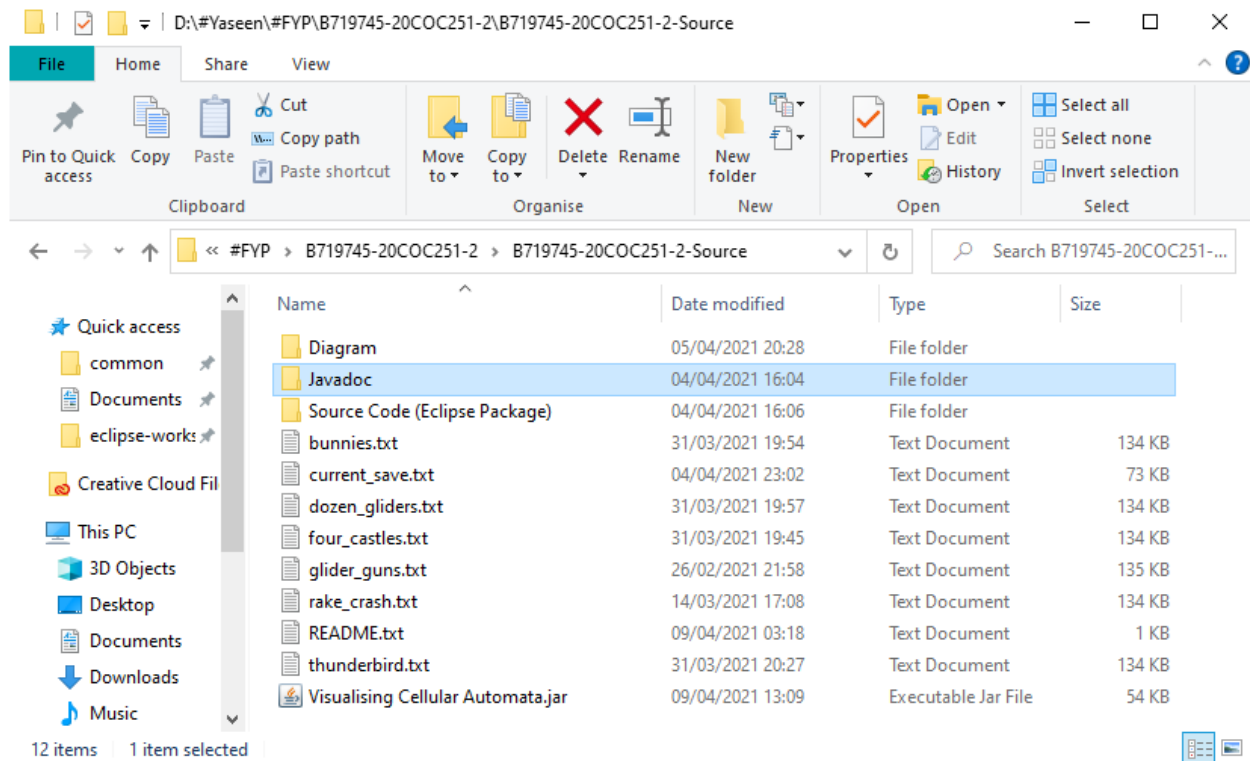
`MainClass` is the class that houses the UI of the program. The UI components present in `MainClass` influence the behaviour of `LifePanel` and, by extension, `LifePanel`'s associated cells. For instance, a user can use `MainClass` to signal the program through the use of the public flag `pauseFlag` to pause or play the simulation.

Both `MainClass` and `LifePanel` are handled in a static fashion, so only one instance of each exists in the diagram. `InitialPopup` is also only instantiated once. The only class that is instantiated several times is the `Cell` class, of which many objects are governed by the `LifePanel` class.



## 5.2: Javadoc

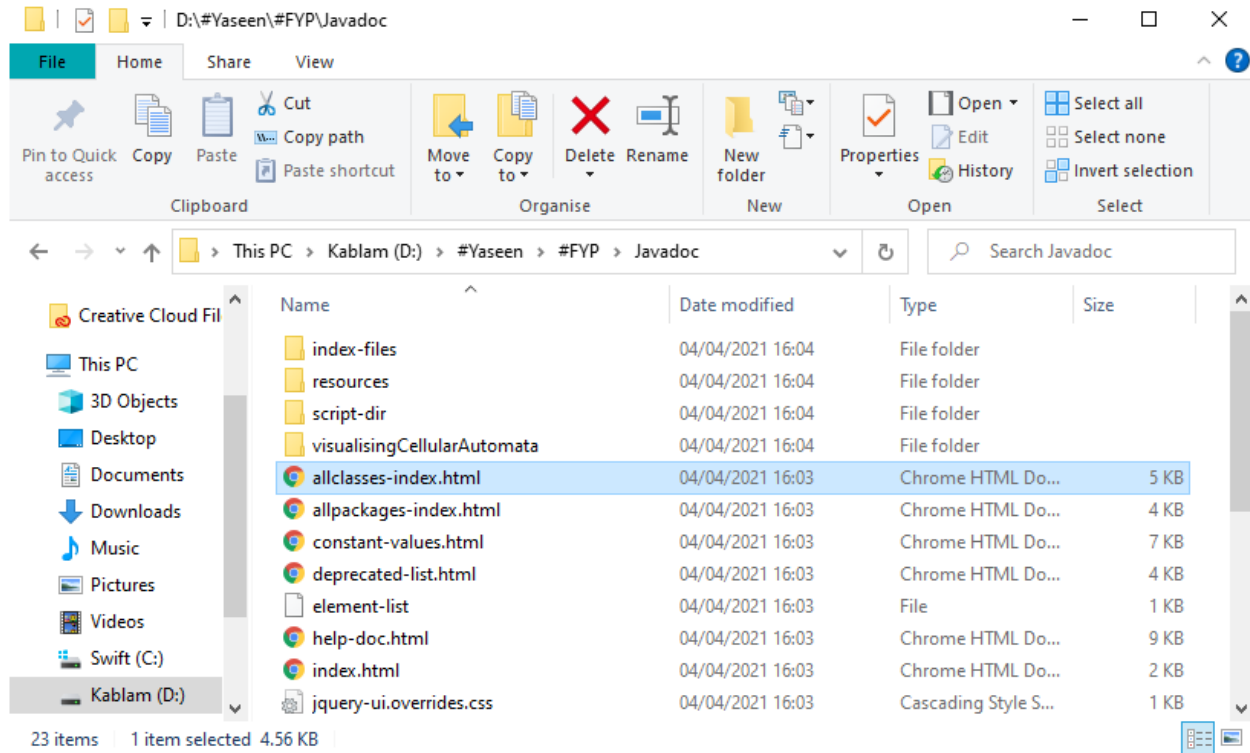
An invaluable tool to better understand the implementation of this program is the supplied Javadoc. It outlines and describes all classes, fields, methods and major elements of the entire program<sup>9</sup>. This chapter is intended as a supplement to the Javadoc, not necessarily as a complete replacement of it.



*Guide 1: The folder in which the Javadoc is located.*

First, navigate to the included “Javadoc” folder and open it. This folder will be found in the source folder, “B719745-20COC251-2-Source”.

<sup>9</sup> It does not cover specific UI components in detail, like buttons and spinners - instead, the fields and variables they control are (and are similarly discussed in this chapter).



*Guide 2: The file to open in order to view the Javadoc.*

When the file "allclasses-index.html" is opened, the Javadoc will open in your default browser. From here, an overview of the entire implementation is provided.

## 5.3: Tools

As discussed in [Design](#), the main tools used were Java and WindowBuilder.

### 5.3.1: Eclipse (the IDE)

The IDE used was [Eclipse](#)<sup>10</sup>. Eclipse is an extremely lightweight IDE with few automatic features adding to the weight of Java code, making it highly effective for implementing bespoke programs and interfaces - like panel 1 (the graphics panel).

### 5.3.2: WindowBuilder

[WindowBuilder](#) is a plugin for the Eclipse IDE. It is a tool that automatically generates Java code when the user creates a design using a GUI themselves. To put it simply, the user creates a GUI by using a GUI.

WindowBuilder follows similar design principles to Eclipse. It is lightweight and versatile: generated WindowBuilder code does not require any additional custom libraries to compile and run. This makes it an effective tool for use.

WindowBuilder was used to generate a Swing GUI, while also automatically generating AWT components where Swing was not applicable. Swing and AWT are discussed under the following [Libraries and Imports](#) section.

---

<sup>10</sup> Eclipse offers many types of IDE. The one used was the Java program development IDE.

## 5.4: Libraries and Imports

Developing a program with a GUI and graphics panels involves using several external libraries. These libraries and imports form the backbone of the many of the methods in the program. Three out of four classes in the program are actually a class extension of imported classes.

### 5.4.1: Swing

Swing is the imported library that handles most of the UI. Swing components include primitives like buttons and sliders, as well as more complicated components including menus, toolbars and popup menus.

Swing is a platform independent tool that generates lightweight components. Because of this, it is perfect for this program.

However, it does not include some elements of backend code for UI elements. That means that some things, like button listeners, are handled by the default Java library of AWT.

WindowBuilder automatically generated Swing and AWT code where necessary. UI components were generated using Swing, and backend elements that Swing was not designed for used AWT methods and objects.

Swing also imports the timer object that is used to update the simulation at regular intervals.

*Table 1: Imported Swing Elements*

imported Swing element	description
<code>javax.swing.JFrame</code>	The JFrame is the main backbone of the program. Its buttons are controlled by the operating system of the machine being used to run the program.
<code>javax.swing.JPanel</code>	JPanels are placed on JFrames, and

	other components appear on JPanels. Panel 1 is a JPanel.
<code>javax.swing.JToolBar</code>	JToolBars are placed on JFrames, and are used for toolbars. Panel 2 is a JToolBar.
<code>javax.swing.JMenuBar</code>	JMenuBars are placed on JFrames. Panel 3 is a JMenuBar.
<code>javax.swing.JPopupMenu</code>	An implementation of a popup menu. They are small windows that popup when a related component, such as a button, is interacted with. Many buttons in panels 2 and 3 use these objects.
<code>javax.swing.JFormattedTextField</code>	A text field that allows for user input. The rule change fields are these objects.
<code>javax.swing.JSpinner</code>	A text field that allows for user input of numbers, dates or other discrete data. A spinner has interactable arrows that allow the user to increment or decrement the input. The user sets the dimensions of the simulation with these objects.
<code>javax.swing.SpinnerNumberModel</code>	The “rules” that a JSpinner object abides by. For instance, the model might be to input dates or integers.
<code>javax.swing.JButton</code>	A simple button that has listeners attached to it in order to function.
<code>javax.swing.JSlider</code>	A slider that supports a range of values. Simulation speed is controlled with these objects.

<code>javax.swing.JLabel</code>	A single line of text that can not be interacted with. Used to “name” buttons to the user.
<code>javax.swing.JRadioButton</code>	Radio buttons are usually grouped together in a <code>ButtonGroup</code> to allow for mutually exclusive choices to be presented to the user (as in <code>InitialDialog</code> ).
<code>javax.swing.ButtonGroup</code>	A button group places several buttons (of any type) into a group that abide by some common rules, such as mutual exclusivity.
<code>javax.swing.SwingConstants</code>	Swing constants are used in Swing methods. For instance, <code>SwingConstants.BOTTOM</code> would align a component to the bottom of something else.
<code>javax.swing.border.EmptyBorder</code>	A border that is used in the programming for padding components.
<code>javax.swing.event. ChangeListener</code>	One of the few listeners that Swing has. This listener listens for a change in value - implemented in <code>JSliders</code> .
<code>javax.swing.event.ChangeEvent</code>	A <code>ChangeEvent</code> occurs when a <code>ChangeListener</code> is triggered.
<code>javax.swing.timer</code>	An object that periodically generates an interrupt, which allows for methods that listen for that interrupt to be called on a regular basis. This regular interrupt allows the simulation to update itself at regular intervals.

### 5.4.2: AWT

[AWT](#) contains all of the classes for painting graphics. This library is the “default” library for Java UI and graphics development, and as such the program outright requires some of its classes to function properly (with no alternative available).

Where Swing objects or methods do not exist, AWT objects or methods are used instead. In general, AWT objects or methods are used for lower-level tasks of the program, most notably for painting graphics.

There is no table here for AWT imports, as such a table would be much too large to display here for its worth to the reader (instead, see [Appendix 2](#) for the table). The program as a whole can be considered “based” in some way on AWT, considering that AWT is used to paint graphics.

It is sensible to consult the [AWT documentation](#), the supplied Javadoc, the code itself, and most importantly [Appendix 2](#) to understand the parts of AWT that are used.

### 5.4.3: Java.io.\* and Java.util.\*

These imports helped achieve some of the basic functionality of the program.

*Table 2: Imported .io and .util elements.*

imported element	description
<code>javax.io.BufferedReader,</code> <code>javax.io.BufferedWriter,</code> <code>javax.io.FileReader,</code> <code>javax.io.FileWriter</code>	Used for reading and writing files, which is used in the program for saving and loading functionalities.
<code>javax.util.Set,</code> <code>javax.util.Random</code>	Set objects are used to store the rule strings for the cellular automaton being simulated. Random objects used to initialise or create random states of the automaton being simulated.

## 5.5: Important Features

Most methods in the program can be understood by consulting the supplied Javadoc. This chapter serves as a supplement to the Javadoc.

### 5.5.1: Graphics (Panel 1)

`paintComponent()` is the imported method (from AWT Graphics) that is used to draw graphics to panel 1. `paintComponent()` calls the custom method `displayPixels()` to draw colour on certain parts of the screen.

`paintComponent()` is called whenever something is made visible, so it must be manually triggered so that it is called whenever something is changed in panel 1.

In this regard, the custom method `nextGeneration()`, which is detailed in its own subchapter, calls the AWT Graphics method `repaint()`, which in turn triggers `paintComponent()`.

The parts of the screen that are coloured are determined by the current state of the automaton being simulated. If a given cell is currently alive, then the part of the screen that it is associated with will be coloured. Otherwise, the black background and gray gridlines will be visible in that place.

`Cell` objects are stored in the array `lifeArray`, through its population of `Cell` objects. If a given cell that is stored in `lifeArray` has a `state = 1`, then it will be drawn to the screen. The location that a cell belongs to on-screen is dependent on its index in `lifeArray` - i.e. index `x = 0, y = 0` is the first entry of `lifeArray`.

#### method `displayPixels`

```
if the cell at lifeArray[i][j] has state == 1 {  
    // colour its associated part of the screen in.  
    // this is dependent on the dimensions of the automaton.  
}
```

`Cell` objects have two attributes: `age` and `state`. `state` determines whether or not the cell should be drawn to the screen, while `age` determines the colour it will



appear as when it is drawn to the screen. The higher the `age` of a cell, the more red it will appear. A cell with an `age` of zero will appear purely white.

```
setColor(255,  
         MAX(0, (255-(lifeArray[i][j].age*10))),  
         MAX(0, (255-(lifeArray[i][j].age*10)))  
        );
```

As `age` increases, the green and blue elements of the RGB colour that the graphics object uses to colour the screen decrease. Minimum bounds prevent memory usage from getting unnecessarily high. This solution is efficient and simple.

### 5.5.2: Automaton Simulation Logic (`check()` and `nextGeneration()`)

The graphics methods let the program draw cells in `lifeArray` to the screen. What remains is to actually populate `lifeArray` with `cell` objects and create new generations of it.

For this, we require a new array of the same type as `lifeArray`. This array is `postLifeArray`. `postLifeArray` is written to based on information in `lifeArray`.

`lifeArray` has the information of all cells in the simulation. Rules are applied to each cell in `lifeArray` based on the [Rule Representation for Lifelike Cellular Automata](#). The result of this rule application is then saved to `postLifeArray`.

To summarise what must occur for a new generation of the automaton to be brought about:

1. Cells need to have the number of live neighbours that they have enumerated. This is done using `check()`.
2. Rules then need to be applied to cells based on the number of live neighbours they have and their own current state, determining their new state in the next generation. This is done using `nextGeneration()`.

#### method check

```
function check(cell c)
int neighbours = 0;
int x = x-location of c in lifeArray;
int y = y-location of c in lifeArray;

if(gridWrapFlag) {
    // efficiently check neighbours of cell c modelling the
    // grid as a torus using the modulo operator to enable grid
    // wrapping.
    neighbours += state of the cell located at
                    lifeArray[(x + xWidth - 1) % xWidth]
                    [(y + yHeight - 1) % yHeight];
    // repeat for all 8 possible locations. This is for the
    // top-left neighbour.
}
else {
    // efficiently check neighbours using try / catch to
    // prevent the use of if statements to see if a cell is on
    // the edge of the grid.
    try {
        neighbours += state of the cell located at
                        lifeArray[x - 1][y - 1];
    }
    catch (error if array index out of bounds) {
        pass;
    }
    // repeat a try/catch cycle for all 8 possible neighbour
    // locations. This code is only for the top-left neighbour.
}
```

*Code 1 (check): The neighbour enumeration method used.*

gridWrapFlag controls if grid wrapping is enabled. xWidth is the amount of pixels in the x-dimension. yHeight is similar to xWidth, but for the y-dimension.

The first thing to do is to enumerate the number of neighbours that a cell has. A focus was placed on efficiency in check() in order to make the program usable on weaker hardware specifications.

Code 1 shows the implementation used (although only partly - the real implementation checks all 8 neighbours, while Code 1 only checks the top-left neighbour).

If grid wrapping is enabled, then the 2D grid of the automaton being simulated will actually be modelled as a torus. In order to minimise the number of if statements used, an efficient implementation of the modulo operator ("%") is applied to make the standard 2D array behave as a 2D torus.

Proof for this can very easily be visualised in the program by placing oscillators at any edge cell in the simulation<sup>11</sup>.

If grid wrapping is disabled (`gridWrapFlag` is false), then there is no escaping the case in which an index-out-of-bounds exception is encountered. If, for instance, a cell on the first row is checked, then looking for the cell at `lifeArray[x - 1][y - 1]` will throw an error.

This is circumvented in the most efficient way possible by using a try / catch cycle. The program assumes that all neighbours exist, and if one does not, it forgives the error thrown when attempting to index the nonexistent neighbour. This is a far more efficient solution than using an if statement to verify if a neighbour cell exists or not.

Overall, the `check()` method completely eliminates in-line if statements. A naive solution would be to use if statements to check if a cell is an edge cell or if a neighbour exists, but this concept has been improved on here.

Further information on the `check()` method (and its full implementation) are discussed both in the Javadoc and the source code.

---

<sup>11</sup> It is also possible to derive a mathematical proof, but is not included here in the interests of space.

method nextGeneration

```

function nextGeneration(array lifeArray,
                        Set birthSet,
                        Set survivalSet){

  for every cell c in lifeArray {
    currentNeighbours = check(c);

    if (c's state == "dead" AND currentNeighbours is in
    birthSet) { // if birth rule applies.
      state of cell c in postLifeArray = 1;
    }

    else if (c's state == "alive" AND currentNeighbours is
    in survivalSet){ // if survival rule applies.
      state of cell c in postLifeArray = 1;
      age of cell c in postLifeArray =
        MIN(255, age of cell c in lifeArray + 1);
      // increment age as the cell has survived
      // through a generation. Max value of 255 to
      // minimise the memory use of the program.
    }
    else { // if no rule applies.
      state of cell c in postLifeArray = 0;
      age of cell c in postLifeArray = 0;
      // the cell must die, so its age and state are
      // reset to 0.
    }
  }
}

```

*Code 2 (nextGeneration): The rule application method used.*

birthSet and survivalSet store the x-string and the y-string of Bx/Sy notation respectively. They are set through the UI with formatted text fields. The Set object (actually a HashSet) offers very efficient search; although its maximum size is only eight, so this has a minimal impact on performance.

Once all cells have been iterated through, `postLifeArray` will now represent the state of the automaton in the generation that is immediately after the one represented in `lifeArray`.

To summarise:

1. `lifeArray` is drawn to the screen using the methods discussed under Graphics.
2. All cells in `postLifeArray` have their attributes, `state` and `age`, set based on information from `lifeArray` using the methods discussed here.
3. `lifeArray` copies the values in `postLifeArray`, and eventually these values are drawn to the screen as in step 1, therefore drawing the next generation of the automaton.
4. Cycle between steps 1 through 3 until otherwise instructed (e.g. program exited).

### 5.5.3: Communication (Flags) and Updating

Communication between `MainClass` and `LifePanel` is based on the use of public static fields. The majority of these fields are boolean flags.

Every time the automaton is to be updated, a set of flags will be checked. The value of these flags is handled by `MainClass` through the user, while the operations that result are handled by `LifePanel`.

method `actionPerformed`

```
function actionPerformed(ActionEvent e) {  
    // If the user prompts a save should occur, do so.  
    if(saveFlag) {  
        saveFile();  
        saveFlag = false;  
    }  
    // If the user prompts to load, do so.  
    if(loadFlag) {  
        loadFile();  
        loadFlag = false;  
    }  
    // If the user wants to clear the simulation, do so.
```

```
if(emptyAutomatonFlag) {
    emptyAutomaton();
    emptyAutomatonFlag = false;
}
// If the user wants to randomise the simulation, do so.
if(randomState) {
    randomSpawn();
    randomState = false;
}
// If the user has not paused the simulation, make the next
// generation and display it.
if (NOT pauseFlag) {
    nextGeneration();
}
// if the user has paused the simulation, but wants to
// single-step it, then they can do so:
else {
    if(singleStepFlag) {
        nextGeneration();
        singleStepFlag = false;
    }
}
repaint();
// repaint() triggers paintComponent() to redraw panel 1.
// this always occurs after nextGeneration() is called,
// which updates the state of lifeArray.
}
```

*Code 3 (actionPerformed): displays how the program handles communication between MainClass and LifePanel.*

`actionPerformed()` is a private method of `LifePanel`. `actionPerformed()` is triggered whenever an action occurs in the program, most usually when the `timer` object generates its regular interrupt. However, interacting with the UI or drawing on panel 1 can also trigger an `ActionEvent`.

Every boolean flag is a public static field of `LifePanel.MainClass` therefore can change the values of these flags freely, and does so based on user input.

When a button in panel 2 is pressed, its effects are communicated to `LifePanel` in this way.

*Table 3: Public static boolean flags used in the program.*

flag	associated button	effect
<code>saveFlag</code>	"Save"	Saves the current state of the simulation to a file called "current_save.txt".
<code>loadFlag</code>	"Load"	Loads the file that is specified by the public static string <code>saveToLoad</code> , which is handled by the "Load" button.
<code>emptyAutomatonFlag</code>	"Clear Grid"	Calls <code>emptyAutomaton()</code> to clear the simulation of all cells (or rather, reset their attributes to 0).
<code>randomState</code>	"Randomise Grid"	Calls <code>randomSpawn()</code> to randomise the states of all cells in the simulation.
<code>pauseFlag</code>	"Play / Pause"	Pauses or plays the simulation.
<code>singleStepFlag</code>	"Single Step"	Generates and displays one new generation of the simulated automaton, and then returns the simulation to a paused state.
<code>heatmapFlag</code>	"Enable / Disable Heatmap"	Determines what the colour that the Graphics object that draws to panel 1 should use. If enabled, cells will become

		redder as they age.
<code>gridWrapFlag</code>	"Enable / Disable Wrapping"	Used in <code>check()</code> to either model the simulation grid as 2D grid or as a 2D torus.

#### 5.5.4: Communication (Non-Flags)

In addition, there are some communications that do not use flags. These include:

*Table 4: Public static non-boolean fields used in the program.*

(Class) static field	associated button	effect
<code>(Timer) timer</code>	"Speed Options" slider	Directly changes the value of <code>timer</code> , a field of <code>LifePanel</code> . This timer generates routine interrupts to trigger <code>actionPerformed()</code> .
<code>(int) stampToUse</code>	First, "Stamps" button in panel 2 is pressed to show panel 3.  Panel 3 then handles setting the value of this field.	Used in a switch statement called by <code>mouseClicked()</code> of <code>LifePanel</code> to determine what the program draws when the user clicks somewhere on panel 1.
<code>(int) drawState</code>	"Draw / Erase Cells"	Used in mouse listeners of panel 1 to determine what the state of cells in the grid should be set to when they are clicked by the user.
<code>(Set) birthSet</code> <code>(Set) survivalSet</code>	"Confirm Rules" and its associated text fields	Used to determine the elements of the sets



		birthSet and survivalSet, which are used to apply rules in nextGeneration().
--	--	--

Again, all of this communication is handled by static variables between classes.

### 5.5.5: `MouseListener` Methods

The UML class diagram indicates that `LifePanel` implements three listener classes: `actionListener`, `MouseListener`, and `mouseMotionListener`. The first is used in `actionPerformed()` and the latter two are used between three imported methods: `mousePressed()`, `mouseClicked()`, and `mouseDragged()`.

*Table 5: `MouseListener` methods used in the program.*

<b>mouseEvent listener in <code>LifePanel</code> Class</b>	<b>What does the user do to trigger this method?</b>	<b>effect upon activation</b>
<code>mouseClicked</code> ( <code>MouseEvent e</code> )	Pressing and releasing the primary mouse button.  Also, its effect is dependent on the current value of <code>stampToUse</code> , which is controlled by the user through panel 3.	Gets the current position of the cursor. Places onto the simulation the pattern that is indexed by <code>stampToUse</code> at that location.  For instance, if <code>stampToUse = 10</code> , then the program draws a glider that moves diagonally up and left on panel 1 at the location that the user clicked.  Stamps are stored in code in methods that are called by the parent method

		<code>stampPattern()</code> . For instance, if <code>stampToUse = 10</code> , then <code>stampPattern()</code> would call <code>stampGlider()</code> .
<code>mousePressed</code> ( <code>MouseEvent e</code> )	Pressing the primary mouse button. If the mouse is not moved before it is released, then <code>mouseClicked()</code> is invoked.	<p>Gets the current position of the cursor. Sets the state of the cell at that location to be equal to <code>1 - drawState</code>.</p> <p>This means that if <code>drawState = 0</code>, live cells are drawn into the simulation. If <code>drawState = 1</code>, then live cells are erased from the simulation.</p>
<code>mouseDragged</code> ( <code>MouseEvent e</code> )	Pressing the primary mouse button, and dragging the cursor while the button is held.	<p>Does the same thing as <code>mousePressed()</code>, but at all points on the cursor's path.</p> <p>If the user drags their mouse around the simulation, a smooth line of live cells will be drawn if <code>drawState = 0</code>.</p> <p>Otherwise, any live cells in the path of the drag will be erased when <code>drawState = 1</code>.</p>

These listeners are implemented into the `LifePanel` class. The user does not control drawing to the simulation through `MainClass`, but rather directly through `LifePanel`.

### 5.5.6: `InitialPopup` Implementation

`InitialPopup` is a class that displays a Swing `JDialog`. This dialog communicates the user-selected values to both `MainClass`<sup>12</sup> and `LifePanel`.

There are four options that the user can manipulate in `InitialPopup`.

*Table 6: Fields of the `InitialPopup` class.*

field in <code>InitialPopup</code>	field options	effect
<code>initialSize</code>	$\in \{5\text{px}, 10\text{px}, 15\text{px}, 20\text{px}, 25\text{px}\}$	<p>Sets the <code>pixelSize</code> of <code>LifePanel</code> by manipulating the value of <code>pixelSize</code> used in the constructor of <code>LifePanel</code> that the <code>main()</code> method uses to start the simulation.</p> <p>The value passed in this constructor sets the value for <code>pixelSize</code> of the simulation, which determines how big each cell appears.</p>
<code>initialWidth</code>	$\in [1280, 3840]\text{px}$	<p>Sets the <code>xSize</code> of <code>LifePanel</code> by setting the value of <code>xSize</code> used in the constructor of <code>LifePanel</code> that <code>main()</code> uses to start the simulation.</p> <p>Also used to determine the restored-down width</p>

<sup>12</sup> The communication to `MainClass` is simply so that `MainClass` is synchronised with `LifePanel`; it is not major in any regard.

		<p>of the <code>MainClass</code> (i.e. the program's <code>JFrame</code> as a whole). Again, this is done by manipulating the value of the <code>MainClass</code> constructor called in <code>main()</code>.</p> <p>The maximum value is bounded by the vertical resolution of the monitor that the program is being run on, and is hard-capped at 3840px.</p>
<code>initialHeight</code>	$\in [720, 2160]\text{px}$	<p>Similar to <code>initialWidth</code>, but for height instead.</p> <p>Its maximal value is bounded by the vertical resolution of the monitor that the program is being run on, and is hard-capped at 2160px.</p>
<code>initialLoad</code>	String value for the filename of the eight saves in the program.	<p>Directly changes the value of <code>initialLoad</code> (variable with the same name, as if they are shared) in <code>LifePanel</code>.</p> <p>If this string is non-empty, then when <code>LifePanel</code> is instantiated it will set its <code>loadFlag</code> value to true, and therefore load the save file specified by <code>InitialPopup</code>'s <code>initialLoad</code> value to</p>

		<p>start the simulation with. <code>lifeArray</code> will be initialised based on <code>initialLoad</code>.</p> <p>If this string is empty, then when <code>LifePanel</code> is instantiated it will initialise <code>lifeArray</code> randomly.</p>
--	--	--

The user can specify the number of cells in the simulation by setting the dimensions of the simulation and the size of each cell. It clearly follows that the number of cells is equivalent to

$((\text{initialWidth} \times \text{initialHeight}) / \text{initialSize}).$

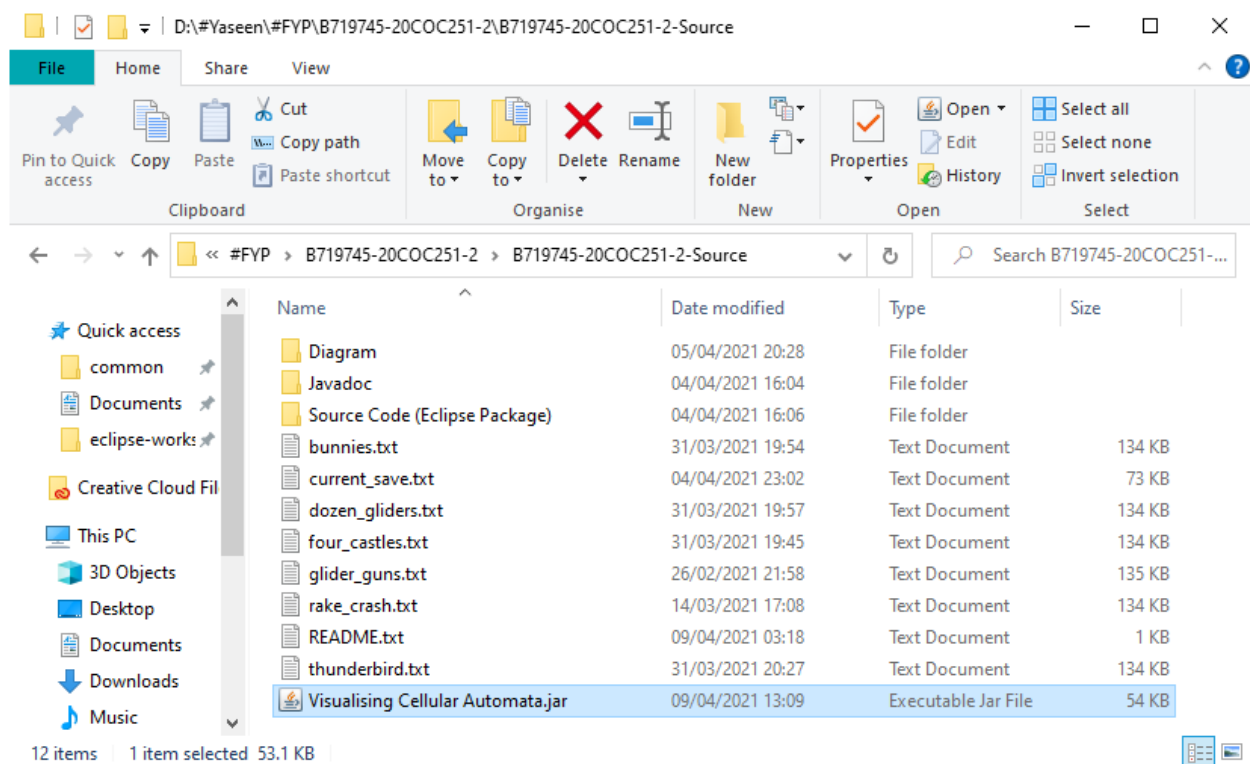
## 5.6: User Manual

To run this program, you must have some JDK or JRE installed. Typically, it is best to install the latest version available, but the program should be compatible with any Java JDK or JRE that is [Java SE 1.8](#) or later (which is JDK 8).

Furthermore, the minimum system resolution required is 1280x720. While the program will run on smaller displays, it will run cropped.

[If your system is using nVidia G-Sync configured with borderless windowed mode enabled, you should either blacklist this program or disable G-Sync.](#)

### 5.6.1: Starting Up



*Guide 3: The runnable .jar file that launches the program. Always ensure that the .txt files are in the same directory as the .jar file.*

In the source folder, run “Visualising Cellular Automata.jar”. Alternatively, import the Eclipse package (found in the “Source Code (Eclipse Package)” folder) into Eclipse and run it using the IDE.

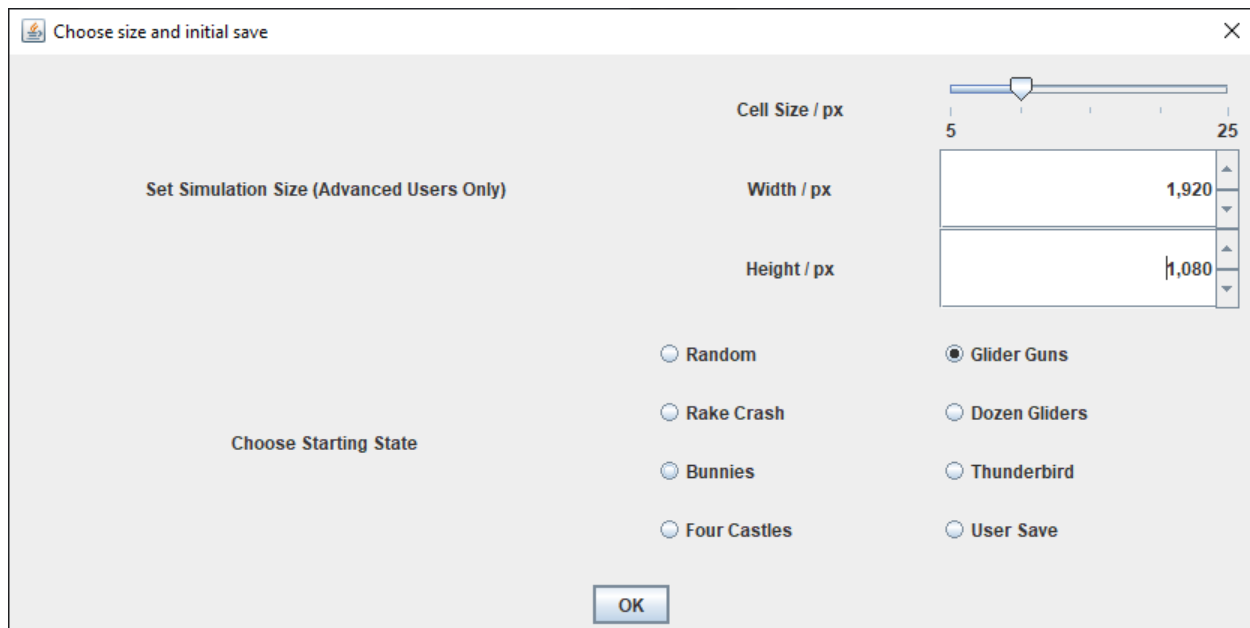


Figure 53: The window that presents itself when the program is run.

The window in Figure 53 is the first thing you'll see when you start the program. There are a total of four options:

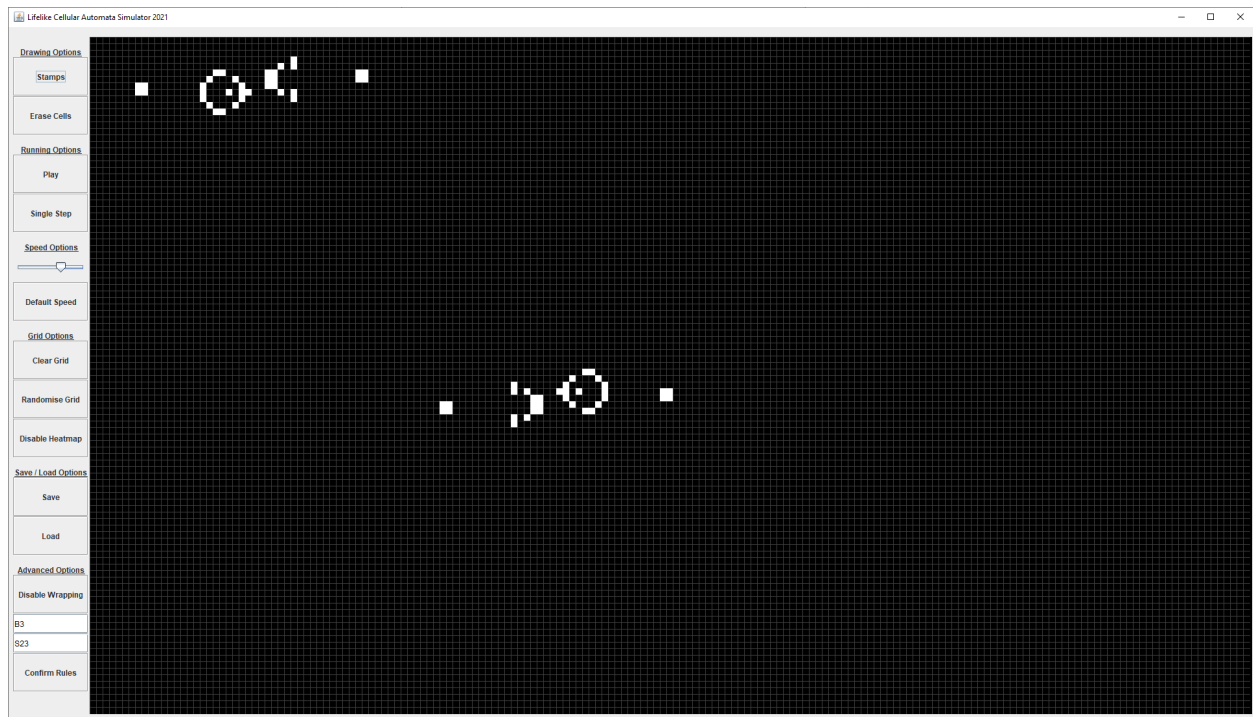
1. **Cell size in pixels:** set the size, in pixels, that each cell will take on the screen.
2. **Simulation width in pixels:** set the width of the simulation in pixels. This has a minimum value of 1280px and a maximum value of whatever your screen resolution is<sup>13</sup>.
3. **Simulation height in pixels:** set the height of the simulation in pixels. This has a minimum value of 720px and a maximum value of whatever your screen resolution is<sup>13</sup>.
4. **Starting state:** these eight radio buttons let you choose what the initial content of the simulation will be. Note that all starting states are compatible with the minimum dimensions of 1280px by 720px<sup>14</sup>.

Generally speaking, you will want to keep the first three options as default. The starting state option is fun to play around with, and you should try seeing them all.

<sup>13</sup> Width has a maximum of 3840px regardless of display resolution, and height has a maximum of 2160px.

<sup>14</sup> All starting states are intended to be compatible with 1280x720 resolutions as a minimum.

Know that you can always change the “starting state” of the simulation by using the included load feature, which is discussed later.



*Figure 54: What is presented to you when you launch the program using the options shown in Figure 53.*

The simulation starts paused. In order to begin it, press the “Play” button on the toolbar that is on the left of the screen. Alternatively, you can use the “Single Step” button to view the animation in single-frame steps.



## 5.6.2: Drawing

There are two ways of drawing:

1. Clicking on the grid. This draws whatever stamp you have selected to it (stamps are discussed shortly).
2. Clicking and dragging across the grid. This draws a smooth line of cells on the grid. Alternatively, you can use the “Erase Cells” button to erase whatever cells you click and drag over.



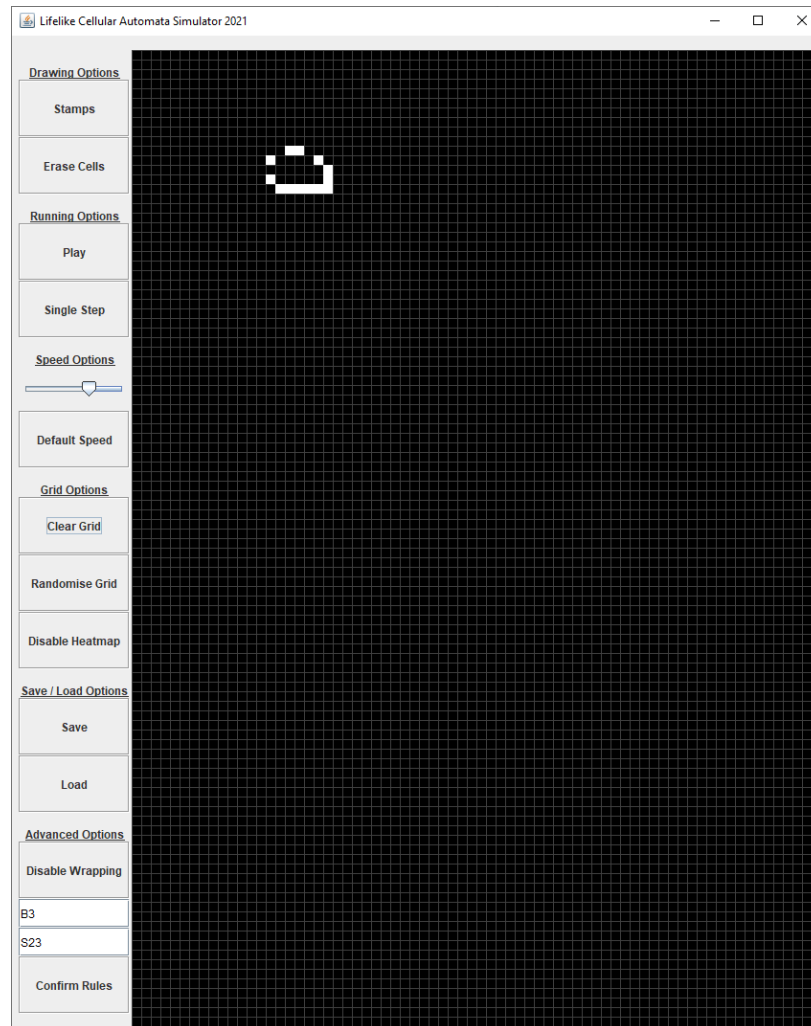
Figure 55: From left to right: clicking on the grid with no stamp selected, clicking on the grid with the up-left Glider selected, clicking and dragging the mouse without pressing “Erase Cells”.



*Figure 56: The Stamp menu is shown after pressing the “Stamps” button.*

To select a stamp that you would like to use, press the “Stamps” button. This will display a menu at the top of the program as shown in Figure 56.

You may now interact with any of the options from Gliders to Oscillators, and choose a pattern that you would like to draw. If you are unaware of what the patterns are, just play around with the different choices present!

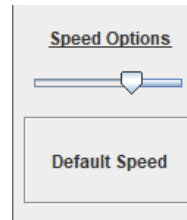


*Figure 57: The result of clicking the grid with the “Left-to-Right” variant of the HWSS selected as your stamp.*

After selecting a stamp, the menu with stamp choices will automatically hide itself. You can always reopen the menu using the “Stamps” button, and select a different stamp (or select no stamp at all).

Finally, you can interact with the “Erase Cells” button on the left to delete cells as you click and drag over them instead of drawing cells as you click and drag over the grid.

### 5.6.3: Speed of Animation

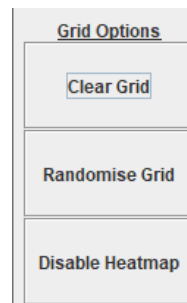


*Figure 58: The speed options that the program has.*

You may interact with the speed slider to speed up or slow down the simulation. Make sure that the simulation is currently playing to select a value that you're happy with.

To return to the default value after using the slider, press the "Default Speed" button.

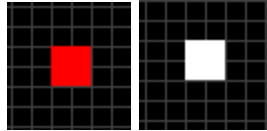
### 5.6.4: Grid Options



*Figure 59: The grid options that the program has.*

**Clear grid:** all the cells in the simulation are cleared.

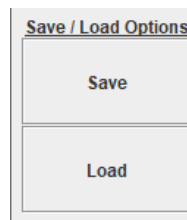
**Randomise grid:** the state of the simulation is randomised. That means that cells are randomly placed in the grid.



*Figure 60: a block of cells displayed with the heatmap enabled (left) and with the heatmap disabled (right).*

**Disable heatmap:** when the heatmap is enabled, cells get redder as they “age”. Cells age as they remain in the simulation for extended periods of time. Disabling the heatmap means that cells will always stay white in colour.

#### 5.6.5: Save / Load Options



*Figure 61: The saving and loading options of the program.*

You can save and load preset configurations of the program.



*Figure 62: The load options.*

When pressing the Load button, the menu in Figure 62 is presented to the user.

The “User Save” is written to every time that the Save button (in Figure 61) is used. This means you can load configurations that you have made yourself.

The rest of the options are also available when starting the program. These load options, when played in the simulation, lead to interesting and beautiful patterns in the Game of Life (the Game of Life is what the program is simulating).

Note that when loading a save from an automaton with a greater size than the one currently being simulated, only the top-left most values that fit will be loaded into the simulation.

### 5.6.6: Advanced Options

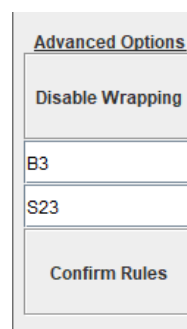


Figure 63: The advanced options.

**Disable wrapping:** normally, when a pattern reaches the edge of the screen, it wraps around to the other edge of the screen. For instance, if a glider moved through the left edge of the screen, it would reappear on the right edge of the screen. If wrapping is disabled, the edge of the screen will behave like a wall, and any pattern that reaches it will crash into it instead of passing through it.

**Rules:** the text fields let you change the rules that the automaton uses based on the notation discussed under [Rule Representation for Lifelike Cellular Automata](#). Some interesting rulesets include:

Table 7: Some interesting Lifelike Cellular Automata rules, excerpted from Wikipedia<sup>15</sup>.

Rule	Name	Description
B2/S	Seeds	All patterns are phoenixes, meaning that

		they never survive through a generation. Configurations will typically “explode” in an epic fashion.
B35678/S5678	Diamoeba	Forms large diamond-patterns with chaotically fluctuating boundaries.
B36/S23	HighLife	Similar to the default ruleset of Conway’s Game of Life

More interesting rulesets can be found on the Internet, such as on Wikipedia [here](#)<sup>15</sup>. The default ruleset used is B3/S23, which is Conway’s Game of Life.

---

<sup>15</sup> [https://en.wikipedia.org/wiki/Life-like\\_cellular\\_automaton](https://en.wikipedia.org/wiki/Life-like_cellular_automaton).

## 5.7: Problems Encountered

Many problems were encountered during development of this program. Some notable problems and how they were overcome are briefly detailed here.

### 5.7.1: Drawing Graphics

First was actually creating the graphics for the program. This was by far the most difficult part to implement as no Loughborough University module taken had detailed how to draw graphics.

Research into the `paintComponent()` and `repaint()` methods and the `Graphics` class eventually resulted in success. Ultimately, the solution is quite simple: after the automaton is updated (when `lifeArray` is updated, triggered by regular interrupts from a `timer`), call `repaint()`. This redraws the graphics.

### 5.7.2: Rule Changes

Second was the inclusion of the rule change functionality. Originally, this problem appeared much more complex than it was. Rules were being stored differently than they are now (in the Bx/Sy notation; and as a result, they were logically impossible to work with.

After researching lifelike cellular automata (and cellular automata as a whole) and the Bx/Sy notation, storing and applying rules became significantly easier. The difficult problem became a simple problem of storing the x-string and y-string of the Bx/Sy notation and matching it to some user input. This was done using sets.

### 5.7.3: Communication Between Classes

Thirdly was the communication in the program. Choosing an efficient method of communication was difficult, but using public static flags (the first choice considered, with others including using communication based on instantiated classes) seemed to work best.

All in all, this implementation was not difficult in code, but rather difficult in choice. That is, it was hard to choose what solution to use, not how to implement one.



## 6: Testing

Generally, testing was performed on a per-prototype basis. Components added to a prototype would be component-tested, and then after all features were added, the prototype was tested as a whole (this is system testing).

This system testing varied from prototype to prototype. When a prototype modified `check()` or `nextGeneration()`, performance testing was conducted to ensure that the program was still running efficiently and smoothly.

This means that the final prototype was fully system tested. When the UML diagram was created and the final program was developed, more extensive testing was done.

All components were re-tested to ensure that they worked correctly, and the entire final program was given its own system test to ensure it behaved as expected.

Unfortunately, due to restrictions imposed by COVID-19, tests could only be performed on a Windows 10 machine. However, since cross-platform tools were used to build the program, there is no reason why the tests would not be valid for MacOS and Linux systems.

## 6.1: Performance Testing

Performance testing was done in two ways; one of which was qualitative and the other being quantitative.

The quantitative test was run on one machine (namely, the one used to develop the program).

The machine used had the specifications:

- OS: Windows 10 Pro (20H2)
- CPU: Ryzen 9 5900X @ 3.7GHz
- GPU: nVidia GeForce RTX 3090 (MSi Gaming X Trio OC model)
- RAM: 32GB DDR4 @ 3600MHz (16-18-18-38 CAS latency)
- Ryzen Balanced Power Plan

More importantly, the test variables were as follows:

- Program ruleset: B012345678/S012345678.
  - the grid of cells would be immediately filled up, and cells would age to their maximum value to maximise the computation and memory load.
- Program timer value: 10ms.
- Program heatmap: enabled.
- Program grid-wrapping: enabled.
- Program dimensions: 10px per cell, 2560px width, 1440px height.
- Test duration: 1 hour.

*Table 8: Performance test results.*

Test	Result
RAM Maximum Load	388MB
CPU Maximum Load	2.2%
GPU Maximum Load	20%

These tests certainly confirm that the program runs efficiently. CPU and RAM loads are very low, which was a focal point of the `check()` and `nextGeneration()` functions.

The GPU maximum load, while appearing high, is not of concern. The system wants to maximise the use of the GPU, so its load appears abnormally high. In general, this test is not useful and in hindsight it actually should have been omitted completely. This concept is confirmed by the user tests.

There is no “upper acceptable limit” assigned to these tests, but rather a qualitative assumption that the performance is adequate.

## 6.2: Functionality Tests

These tests aimed to ensure that functionality of the program worked. Component tests tested each component, and all of the tests in Table 9 put together formed a full system test when combined with performance tests.

*Table 9: Functionality tests for the program.*

Test	Final Program's Result
<b>Does the program launch?</b>	Pass
<b>Do all of the options in InitialDialog function as expected?</b> (this includes setting the pixel size, simulation width, simulation height, and initial configuration)	Pass
<b>Does the simulation logic work?</b> (i.e. is Conway's Game of Life simulated when using B3/S23)	Pass
<b>Do the drawing options work?</b> (this includes Stamps, drawing live cells, and erasing live cells)	Pass
<b>Do the running options work?</b>	Pass

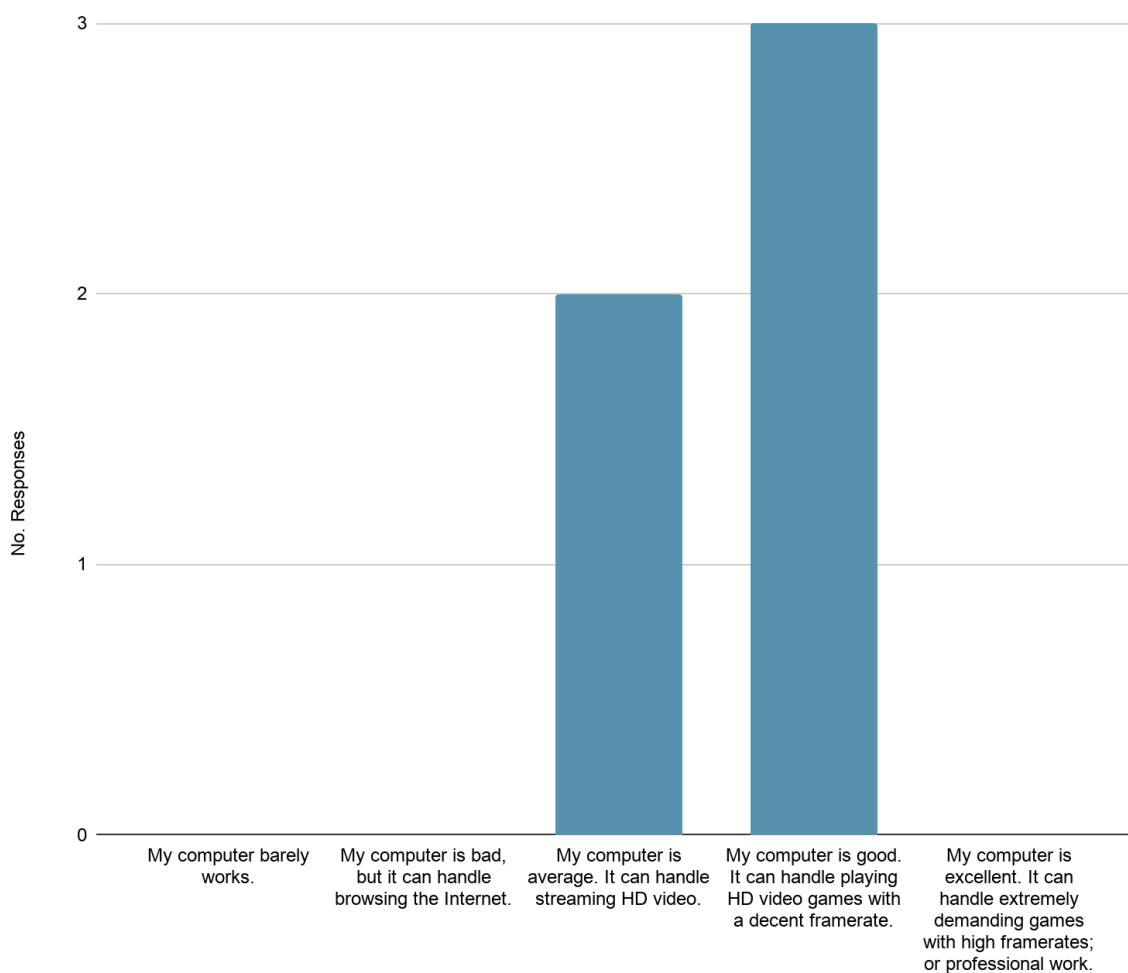
(this includes Play / Pause and single stepping the simulation)	
<b>Do the speed options work?</b> (this includes changing the timer and resetting it to default speed)	Pass
<b>Do the grid options work?</b> (this includes clearing the grid, randomising the grid, and disabling / enabling the heatmap)	Pass
<b>Do the save / load options work?</b> (this includes saving a custom file, loading a custom file, and loading pre-determined files)	Pass
<b>Do the advanced options work?</b> (this involves changing the rules to B2/S and seeing if Seeds is simulated)	Pass
<b>Does the program close on command?</b> (this ensures that the program is not interfering with the machine in some unknown way)	Pass

## 6.3: User Tests

User surveys were conducted to ensure that the program was performing as expected. This is especially useful as limitations imposed by COVID-19 meant that it was impossible to test the program on other machines.

Five people, external to the project, were surveyed. All five participants did not use nVidia's G-Sync technology.

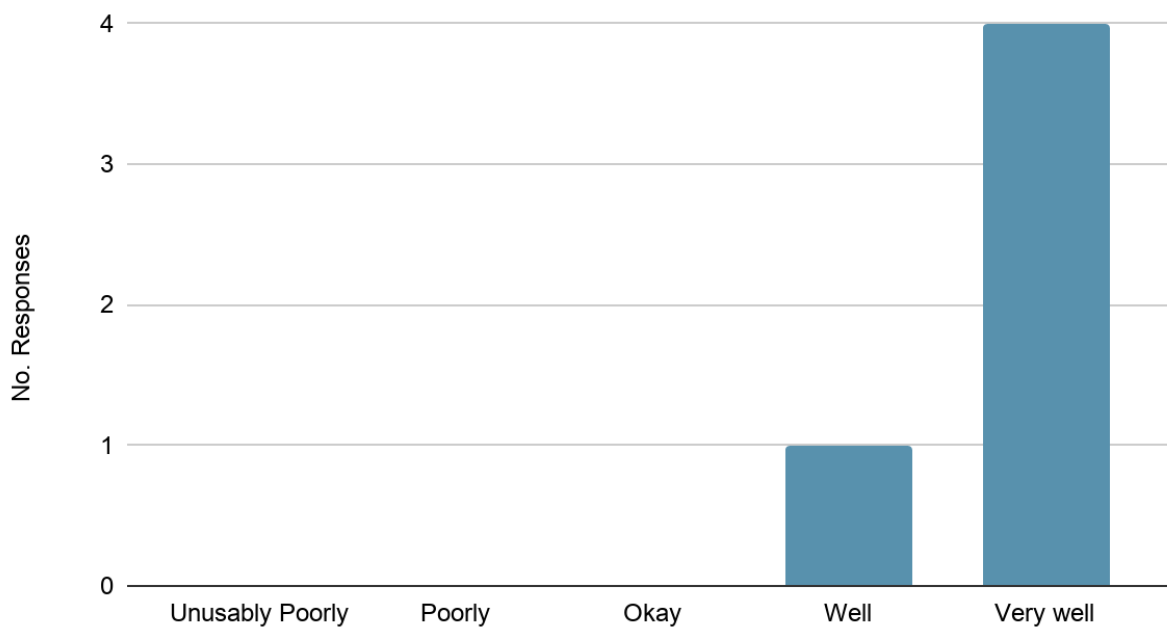
What would you say that the quality of your computer is?



*Graph 1.*

Participants generally have strong computers. It is probably not too far-fetched to assume that the average user's computer can handle streaming HD video, such as those on YouTube.

### How well did the program run?



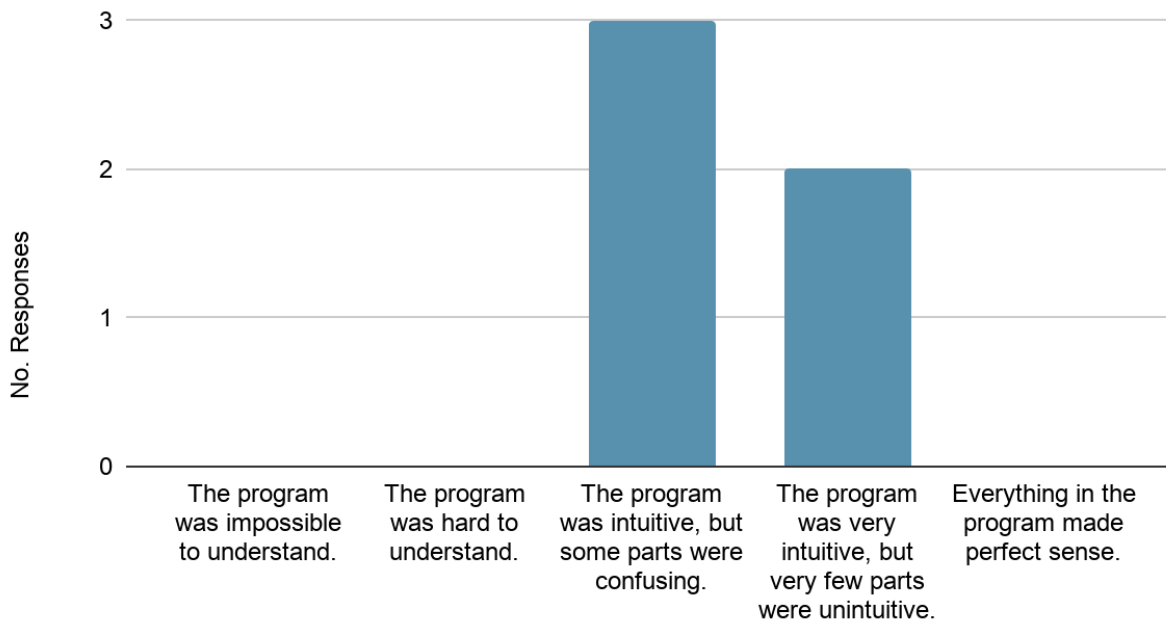
*Graph 2.*

Graph 2 gives the response to the question “How well did the program run?”. The general response is that the efficiency of the program is very good, but the hardware specifications of the participants were not fully recorded.

- The participant that responded “Well” answered that they had a “good computer”.

Furthermore, this question indicates that the program was bug-free. It is unlikely that a respondent would answer that the program ran well if they were unable to get it to work because of a bug. The lack of mentions of bugs in user comments further consolidate this.

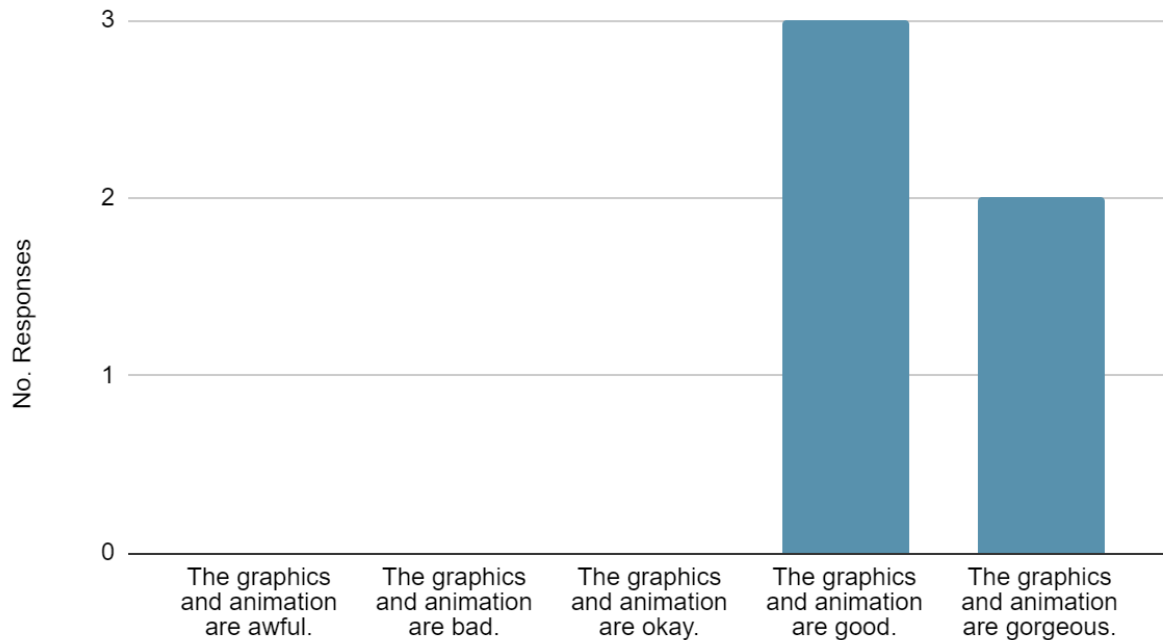
### How intuitive was the program to use?



*Graph 3.*

Graph 3 shows that the program should probably have its UI improved. The best possible response is most likely received no responses due to the “Advanced Options” section. However, in general, the intuitiveness of the program is satisfactory. Because this program deals with quite the abstract concept of cellular automata, this can be considered a good response.

Ignoring the UI, how aesthetically pleasing did you find the graphics and animation of the program (including the heatmap functionality)?



*Graph 4.*

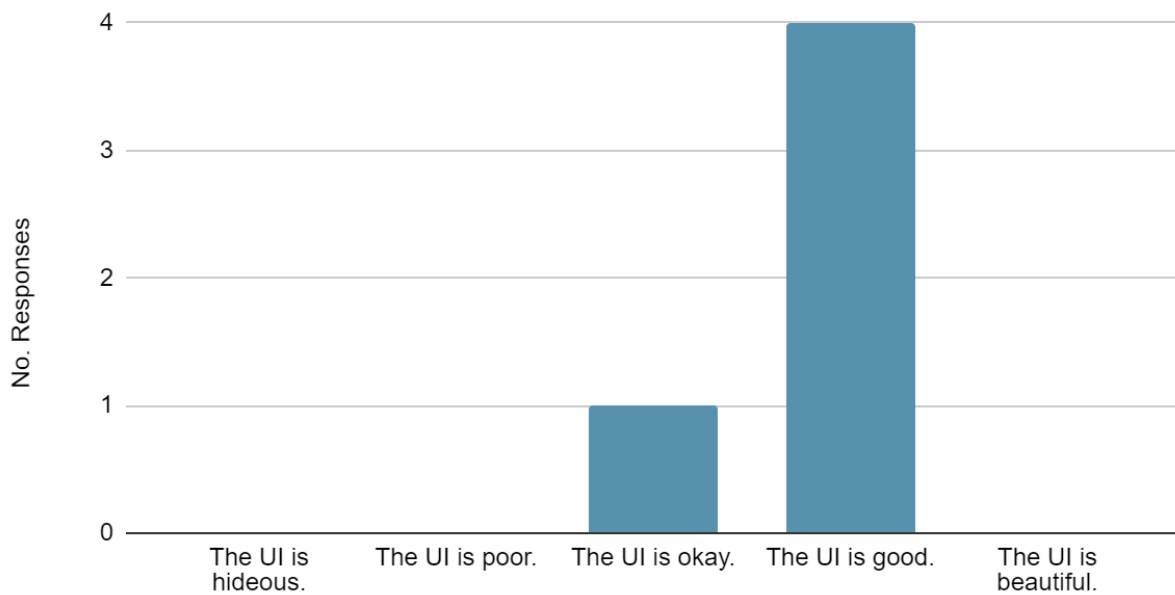
Graph 4 indicates that the animation and graphics (panel 1) was very well executed. It is important to note that since the participants had a personal relationship with the reviewer that these results may be skewed positively, even if the survey asked that they be as objective as possible<sup>16</sup>.

---

<sup>16</sup> The survey was also anonymous.



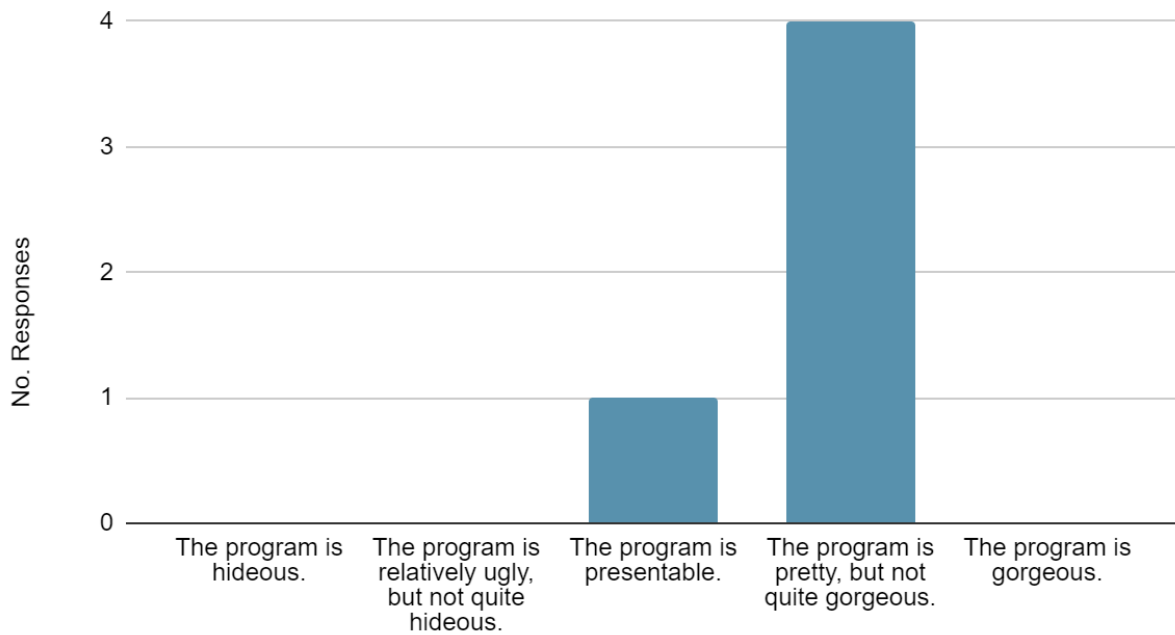
Ignoring the graphics, how aesthetically pleasing did you find the UI of the program (the buttons and popups)?



*Graph 5.*

Graph 5 indicates again that the UI is a point for improvement in the program. While the UI is not considered ugly, it can certainly be improved. However, it is clearly more than satisfactory.

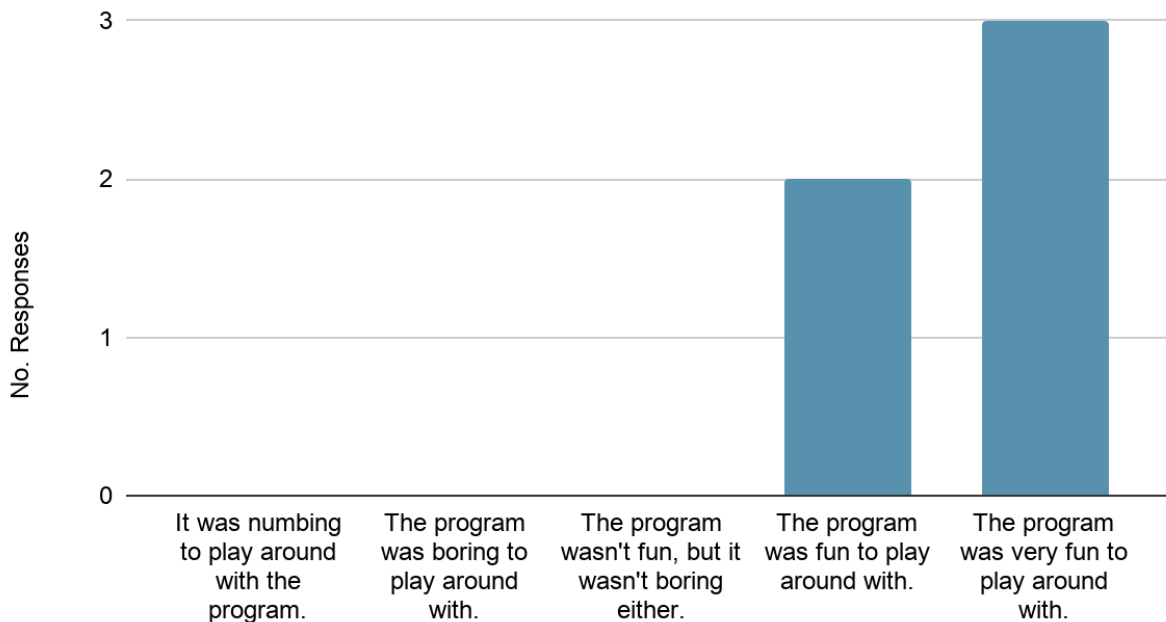
Overall, how aesthetically pleasing did you find the program?



*Graph 6.*

Clearly, the UI has dragged down the responses of this question. Still, the response is strongly positive.

## Was the program fun to use?



Graph 7.

The program is clearly enjoyable to use. Seeing as it is a simulation, it is an obvious goal for it to be fun; a goal that it has evidently achieved.

### 6.3.1: User Comments

The survey also included an optional field to leave comments. All five participants left a comment.

*"The programs [definitely] fun to play with, but the UI could be improved for sure. It isnt awful though"<sup>17</sup>*

The first comment clearly indicates that the UI is a point for improvement. It is assumed that they are talking about the aesthetics and not the functionality, considering that this person found the program "very intuitive" and rated the aesthetics of the UI as "okay".

---

<sup>17</sup> Spelling error "definitely" corrected.

*"I love this program, the animation is weirdly entrancing. I wish loading and saving was better."*

Another great addition to this program would be to make the save / load functionality to use a file browser instead of a popup menu. Doing so would address the concern outlined in this user comment.

*"I like it alot but I think you need to add a feature that rewinds the game!"*

Unfortunately, because many lifelike cellular automata (like the Game of Life) are non-reversible, this is impossible to implement without greatly increasing the memory load of the program. Furthermore, a feature such as this would take a noticeable amount of time to implement; however, it is certainly an interesting concept.

The main takeaway here is that the user enjoyed the program.

*"I love playing with this so I cant really fault it very much [to be honest]. However I would like a way to change the size of the game while its happening instead of restarting every time."<sup>18</sup>*

Currently, resizing the simulation while it is running is not possible. This would be a great feature to implement if more time was allotted, and would certainly be supported by the current code with some minor refactoring.

Overall, this feedback is positive in the way that the participant "loved playing with" the simulation.

*"I think the ui should be made a little more professional looking. I also reckon there should be some more choices to load because theyre a lot of fun to use, and maybe some more stamps as well because theyre great too."*

---

<sup>18</sup> Originally "tbh", meaning "to be honest".

Again, the aesthetics of the UI is a point of criticism. The participant also states that more stamps and predefined load states should be included. The flexibility of the current code will allow this with ease, so this request could easily be served with more time allotted.

Positive feedback regarding the stamps and predefined loads indicate that the program's main goal was a success: to simulate lifelike cellular automata in a fun way.

To summarise, the user feedback on the program was positive. It seems as though the program is successful in its aim to simulate lifelike automata in a fun and accessible way.

However, the user feedback is limited with both its small sample size and the fact that the participants were friends with the reviewer. Still, the feedback is certainly still valid; and participants were clearly not afraid to point out weaknesses in the comment section.

## 7: Conclusions

It can be strongly argued that the program has met its main goals. These were:

1. To simulate lifelike cellular automata in a fun and accessible way, even to users who are not well-versed with automata in general.
2. The program must be efficient.
3. The program must be well-built and expandable.

User surveys proved that the program is fun and accessible.

The performance testing and the user surveys together prove that the program runs efficiently and smoothly.

The program is written in an expandable way, which will be discussed briefly under the future expansions segment that follows. The UI is modular and best-practices have been followed to ensure that the program code is clear and maintainable.

### 7.1: Future Expansion

#### 7.1.1: File Browser

Currently, when the “Load” button is pressed, a popup menu is presented to the user.

An improvement to this would be to use a file browser when the load button is pressed. Similarly, this could be implemented for the “Save” button.

This would greatly increase the amount of files that the program can save and load, and improve the intuitiveness of the UI. Currently, only one save file can be saved by the user, and it is always overwritten. This clearly is not ideal, and this solution would benefit the program as a whole greatly.

Fortunately, the code is flexible enough that such a change would only require minor refactoring. Namely, the buttons would have their popup menus removed

and their `actionListeners` edited to launch a file browser. Then, the `saveFile()` and `loadFile()` methods would have to be edited to work with a file browser (and to verify files before loading them). Finally, `InitialPopup` must be changed to allow a file browser as well.

### 7.1.2: Rewind Feature

The program would save previous iterations of `lifeArray`, in an encrypted form (to make it compact), allowing the user to rewind the simulation with the minimum possible impact on memory usage.

Implementing such a feature would require an encryption method to efficiently store `lifeArray` states, a new button to facilitate this functionality and another button to return to the very first state saved (or perhaps a state that the user “checkpoints”).

### 7.1.3: Improved UI

`WindowBuilder` is lightweight and versatile. The UI generated can easily be changed, most likely by using `WindowBuilder`’s Look’n’Feel feature to improve the general aesthetics of the program.

While concepts of HCI (human-computer interaction) have been implemented, more could be applied to make the UI better - such as by implementing the file browser discussed.

### 7.1.4: Increased Neighbourhood Sizes

Increasing the neighbourhood size would allow the program to simulate more than just lifelike cellular automata. In turn, non-totalistic rules could be implemented, allowing for all possible cellular automata to be simulated (only limited by size).

This would involve changing the `check()` method and implementing a way of controlling it using the UI.

### 7.1.5: Increased Size (and Resizable Simulation)

The program could implement a method to infinitely generate the grid (or perhaps generate an extremely large grid). The program can then zoom in and out on the grid, greatly increasing the scale of the simulation.

Currently, the program can not resize the simulation without a restart - this change would fix this and improve the feel of the program as a result. However, some interesting aspects of grid-wrapping would be lost if the grid was made so large.

Unfortunately, this “infinite grid” change would also greatly increase the scope of the program. Implementing such a feature would require significant planning, but existing logic methods and UI methods would still work effectively.

Making the grid resizable during the simulation is a smaller change, and would not require the same planning that infinitely generating the grid would.

In general, the methods and fields that define the simulation like `gridCreator()` and `lifeArray` would have to be modified. Some methods may have to be overhauled or added - such as using a mouse listener to allow the scroll wheel to zoom in or out.

### 7.1.6: HashLife

HashLife is an algorithm created by Bill Gosper (1984), detailed in his paper "Exploiting Regularities in Large Cellular Spaces".

HashLife is a time-efficient algorithm for computing the long-term fate of a given starting configuration of the Game of Life. HashLife takes an extremely large amount of memory and can not simulate the automaton step-by-step; but it could be used as an optional function to determine the fate of predefined start states or states created by the user.

It uses a quadtree structure and memoization (saving answers in recursion to look them up later) to compute the result of the given Game of Life configuration (Gosper, 1984).



Unfortunately, HashLife is a complex algorithm with a very complex implementation. Adding the buttons to start HashLife would be the easy part: actually implementing the algorithm, especially in a restrictive language like Java, would most likely be very difficult. C++ would be a more suitable choice in this regard, but then the entire program would have to be rewritten.

Implementing HashLife is certainly possible, but it would take noticeable planning and development time. Furthermore, considering its extremely large memory load, it may not even be feasible to do so. This is especially likely when considering other features such as the “infinite grid” feature.

Ultimately, the idea of HashLife, while very interesting, would have to be researched more deeply to reach a real conclusion as to whether or not its implementation is feasible. After all, HashLife’s results can be “replicated” so to speak by simply increasing the CPU load by increasing the simulation speed to reach the “fated state” quickly.

## 7.3: Summary

Several practices, from software engineering to programming, have been used to create a program that simulates cellular automata. Studies from many walks of Computer Science have been used to realise this project in an effective and efficient manner.

Algorithm design was implemented to make the program efficient, HCI was used to make the UI as intuitive as possible, and object-oriented programming and functional programming modules were used to make the program expandable.

Software engineering was used to create the design and set out an effective plan to ensure that the project went smoothly.

# Appendix 1: nVidia G-Sync Incompatibility

It has been found that this program is partially incompatible with nVidia G-Sync, when G-Sync is configured to run for windowed applications.

G-Sync is a proprietary firmware that is available on some monitors that links with compatible nVidia graphics cards. G-Sync allows the monitor to match its refresh rate to the frame rate being output by the installed nVidia graphics card.

Typically, G-Sync is used in conjunction with a gaming computer to create a smooth experience without any frame tears when playing video games. As such, G-Sync automatically detects what the user is doing (if they are playing a video game or not), and activates or deactivates itself accordingly.

G-Sync is currently recognising this program as a video game by default. This, in turn, reduces the frame rate of the monitor to match the frame rate of the application. The frame rate of the application seems to be determined by the update rate of the automaton, but it is certainly not a 1:1 ratio.

Most modern-day monitors run at 60Hz, which translates to 60 frames per second. In testing, monitors do not run at 60Hz when G-Sync is active with this automaton open. G-Sync forces the monitor's refresh rate to drop to as low as 20 frames per second, and if the automaton updates below this speed (with respect to the unknown ratio), G-Sync automatically deactivates itself.

When G-Sync is turned off, the low framerate problem disappears entirely. Adding a rule that prevents G-Sync from working on the program also alleviates the issue (this is supported by all G-Sync enabled monitors).

## Appendix 2: AWT Imports

Table 10: AWT imports.

imported AWT element	description
<code>java.awt.Graphics</code>	a <code>Graphics</code> object is what is used to draw to the screen. Panel 1 is based on using a <code>Graphics</code> object to draw cells to the screen based on the state of <code>lifeArray</code> .
<code>java.awt.Color</code>	Used to set the colour that a <code>Graphics</code> object uses to draw to the screen.
<code>java.awt.ActionListener</code>	Listens for “action” on some other object. A button can feel action when it is interacted with, for example.
<code>java.awt.event.ActionEvent</code>	An <code>ActionEvent</code> is triggered when an <code>ActionListener</code> is triggered. These listeners are used to allow the user to interact with the UI.
<code>java.awt.MouseAdapter</code>	This abstract class has to be extended to implement <code>MouseListeners</code> . <code>MouseAdapters</code> themselves have empty methods for receiving mouse events.
<code>java.awt.MouseListener</code>	Listens for discrete mouse input on some other object. A mouse button being pressed would trigger this listener.
<code>java.awt.MouseMotionListener</code>	Listens for continuous mouse input on some other object. The mouse being clicked and dragged would first trigger a <code>MouseListener</code> , but then a <code>MouseMotionListener</code> .

<code>java.awt.MouseEvent</code>	A <code>MouseEvent</code> is triggered when a <code>MouseListener</code> or a <code>MouseMotionListener</code> is triggered. These listeners are used to allow the user to draw to the screen.
<code>java.awt.FocusAdapter</code>	This abstract class has to be extended to implement <code>FocusListeners</code> . <code>FocusAdapters</code> themselves have empty methods for receiving keyboard focus events.
<code>java.awt.FocusEvent</code>	A <code>FocusEvent</code> is triggered when a <code>FocusListener</code> is triggered. These listeners are used to allow the user to interact with the UI: in this case, focus is typically used to auto-highlight a textbox when it is selected.
<code>java.awt.BorderLayout</code>	A way of setting the layout of a component.
<code>java.awt.GridLayout</code>	A way of setting the layout of a component.
<code>java.awt.FlowLayout</code>	A way of setting the layout of a component.
<code>java.awt.Toolkit</code>	Used to get the dimension of the monitor that the user is using. Used to automatically set the dimensions that the simulation should start with (but can be overridden by the user).
<code>java.awt.Dimension</code>	A dimension is an object that encapsulates the width and height of another object in a single object.
<code>java.awt.Component</code>	An object having a graphical representation that can be displayed on the screen and that can interact with

	the user. <code>MainClass</code> uses this.
<code>java.awt.EventQueue</code>	A platform-independent class that queues events, both from the underlying peer classes and from trusted application classes.

# References

Weisstein, Eric W. (unknown). "Elementary Cellular Automaton." MathWorld - A Wolfram Web Resource. Retrieved from <https://mathworld.wolfram.com/ElementaryCellularAutomaton.html>.

Najera, Jesus. (2020). "Elementary Cellular Automaton: A Theory On How Simple Structures Generate Complex Systems". Cantor's Paradise - a publication by Medium. Retrieved from <https://www.cantorsparadise.com/elementary-cellular-automaton-e27e3d1008d9>.

Weisstein, Eric W. (unknown). "Moore Neighborhood". MathWorld - A Wolfram Web Resource. Retrieved from <https://mathworld.wolfram.com/MooreNeighborhood.html>.

Weisstein, Eric W. (unknown). "Cellular Automaton". MathWorld - A Wolfram Web Resource. Retrieved from <https://mathworld.wolfram.com/CellularAutomaton.html>.

Wolfram, Stephen. (2002). "A New Kind of Science". Stephen Wolfram LLC. p60, p876.  
*accessed at* <https://www.wolframscience.com/reference/notes/876b>

Sarkar, Palash. (2000). "A Brief History of Cellular Automata". Indian Statistical Institute. p1.  
*accessed at* [http://www.agentgroup.unimore.it/Zambonelli/didattica/cas/L8/CA\\_history.pdf](http://www.agentgroup.unimore.it/Zambonelli/didattica/cas/L8/CA_history.pdf)

Buckley, William R. (2000). "On the Complete Specification of a von Neumann 29-State Self-Replicating Cellular Automaton". p9.  
*accessed at* [https://www.researchgate.net/publication/235726266\\_On\\_the\\_Complete\\_Specification\\_of\\_a\\_von\\_Neumann\\_29-State\\_Self-Replicating\\_Cellular\\_Automaton](https://www.researchgate.net/publication/235726266_On_the_Complete_Specification_of_a_von_Neumann_29-State_Self-Replicating_Cellular_Automaton)

Donald Bren School of Information and Computer Science. (unknown). "Wolfram's Classification of Cellular Automata". Retrieved from <https://www.ics.uci.edu/~eppstein/ca/wolfram.html>.

Ilachinski, Andrew. (2001). "Cellular Automata: A Discrete Universe". World Scientific. p12-13, p44-45.

Wolfram, Stephen. (1984). "University and Complexity in Cellular Automata". Physica D. p1-35.

*accessed at*

<http://new.math.uiuc.edu/im2008/dakkak/papers/files/wolfram.universityofca.pdf>

Kari, Jarkko. (1990). "Reversibility of 2D cellular automata is undecidable". Physica D. p1-3.

*accessed at* <https://www.sciencedirect.com/science/article/abs/pii/016727899090195U>

Eppstein, David. (2010). "Growth and Decay in Life-Like Cellular Automata". Springer, London.

*accessed at* [https://doi.org/10.1007/978-1-84996-217-9\\_6](https://doi.org/10.1007/978-1-84996-217-9_6)

Gosper, Bill. (1984). "Exploiting Regularities in Large Cellular Spaces". Physica D.

*accessed at* <https://www.lri.fr/~filliatr/m1/gol/gosper-84.pdf>