# Nathan Strong

**Game of Life Simulator**

# Software Design Document

Name(s): Nathan Strong

Lab Section: COMP 150

Workstation: N/A

Date: 02/16/2024

**TABLE OF CONTENTS**

1. **INTRODUCTION**
   1. Purpose
   2. Scope
   3. Overview
2. **SYSTEM OVERVIEW**
3. **SYSTEM ARCHITECTURE**
   1. Architectural Design
   2. Decomposition Description
4. **DATA DESIGN**
   1. Data Description
   2. Data Dictionary
5. **COMPONENT DESIGN**
6. **HUMAN INTERFACE DESIGN**
   1. Overview of User Interface
   2. Screen Images
7. **REQUIREMENTS MATRIX**

# 1. INTRODUCTION

(The project’s flowchart is located here: [COMP 150 Final Flowchart](https://ufv1974-my.sharepoint.com/:u:/g/personal/nathan_strong_student_ufv_ca/EVHkU5a3nT9OjghGS6az2iAB6uRtOHs_4oPMuKQVfpon8g?e=0Ad6bZ))

* 1. **Purpose**

This document describes the design and implementation of an application which runs Conway’s Game of Life, a cellular automaton. The application also provides the ability to edit, save, and load grid states.

The purpose of this project is to create an interface for interacting with the cellular automaton, allowing for the exploration of patterns that can be found within it. It also aims to demonstrate some simple programming concepts, such as multi-dimensional arrays and file I/O.

## Scope

This project includes the following features:

* A GUI which displays the grid that the automaton runs on, with the following features:
  + Tiles which toggle state from alive to dead when clicked
  + An interface to control the rate of the simulation, or step through individual ticks
* A system which allows for the saving and loading of grids, with the following features:
  + A custom file format which stores the states of all cells in its simulation, with the suffix “.cgol”
  + Buttons on the GUI to trigger saving and loading.

## Overview

This document contains information about the specifications to which this application will be built. Its sections, and the details contained in them, are as follows:

* 1. Introduction
  + This section contains information regarding the general concept of the project, as well as details of this document itself.
* 2. System Overview
  + This section contains more detailed information regarding the functionality, context, and design of the project, as well as background information on a few concepts required to understand its purpose.
* 3. System Architecture
  + This section contains information on the implementation of the features described in Section 2, except for descriptions of the data used.
* 4. Data Design
  + This section contains descriptions of the stored data used to make the software function.
* 5. Component Design
  + This section contains an even more in-depth description of the implementation of specific parts of the program, detailing individual functions and algorithms used.
* 6. Human Interface Design
  + This section contains an explanation of the interface which the program uses to allow the user to interact with it.
* 7. Requirements Matrix
  + This section contains the requirements which the project must fit, and the parts of the project which fulfill them.

# 2. SYSTEM OVERVIEW

## 2.1 General Project Aims

This project’s aim is to allow the user to simulate Conway’s Game of Life acting on patterns of their design. It aims to do this in a way that is both efficient and intuitive to use, and to allow the user full control over features of the simulation such as the speed at which it runs.

## 2.2 Description of the Game of Life

Conway’s Game of Life is a simple cellular automaton. This means that it operates in steps over a grid of cells, and that at each step each cell runs some computation on its neighbors to determine what its state should be in the next step. In the case of the Game of Life specifically, the possible states are alive and dead. In each step, each cell counts the number of its neighbours which are alive (Neighbours, in this case, are any cell adjacent horizontally, vertically, or diagonally to the cell in question), and whether the cell itself is alive. If the cell is dead and it has exactly 3 living neighbours, it becomes alive. If the cell is alive and has 2 or 3 living neighbours, it remains alive in the next step. Otherwise, the cell is dead. This relatively simple set of rules can produce surprisingly complex results when computed over a large enough grid, even going so far as to be Turing-complete. A more detailed description of the Game of Life can be found in this article: <https://conwaylife.com/wiki/Conway%27s_Game_of_Life>.

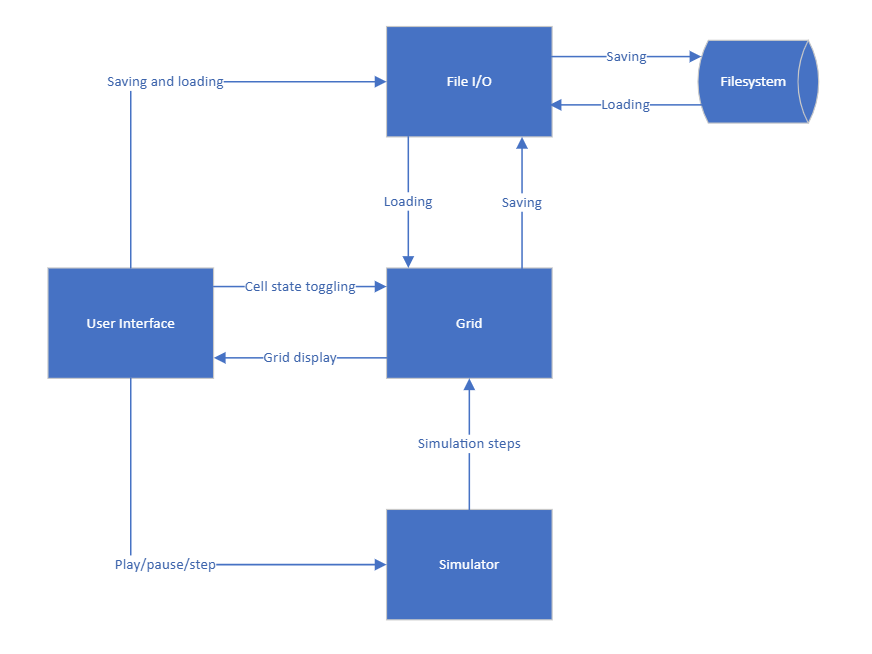
## 2.3 Other Features

This project will have several additional features built onto its Game of Life simulation, for quality of life and to allow for user interaction. These features will include the ability to change the rate at which the simulation runs (including pausing it), to change properties of the simulation such as dimensions of the grid, and to save and load grids using a custom file format.

# 3. SYSTEM ARCHITECTURE

## Architectural Design

The project will contain 4 major subsystems: The grid, the simulator, the user interface, and the file I/O system. The grid stores the current state of the automaton, and the simulator allows the grid to transition from one step to the next. The user interface allows the user to affect when the simulator runs a step, which files are saved or loaded, and to toggle which grid cells are alive or dead. The file I/O system takes care of the reading and writing grids from files, using a file format which stores the grid’s cell states as well as some details about it, such as size. Below is a diagram of the interactions between these subsystems.



## Decomposition Description

### The Grid

The grid is a 2-dimensional array of Boolean values denoting whether each cell in the simulation is alive or dead. It interfaces with all other subsystems in some capacity, though it has little functionality of its own.

### User Interface

The user interface controls the rendering of all relevant data to the screen. It displays the grid, as well as some text concerning its dimensions and other information. It also contains several buttons, which allow the user to interact with the step rate of the simulator and to save and load grids using the file I/O system. The user interface is also responsible for handling timings of step events, so it calls the simulator to do a step whenever necessary.

### The Simulator

The simulator takes care of running the automaton, using whatever step rate has been designated by the user. It carries out the necessary processes each time a step needs to be performed, reading from and writing to the grid as necessary to achieve this.

### File I/O

The file I/O system handles the saving and loading of the grid’s data to and from the filesystem of the computer. It does this whenever the user clicks one of the previously mentioned user interface buttons, requesting file paths as needed.

# 4. DATA DESIGN

## Data Description

The project needs to store only limited data to accomplish what it is supposed to. This data is entirely contained in the grid subsystem and consists of the 2-dimensional array which was mentioned in its description as well as a pair of integers for its size and a Boolean to determine whether the grid wraps around (with edges being treated as adjacent to their opposite sides) or not (with edges being treated as adjacent to zeroes).

## Data Dictionary

Some additional data is stored for the purpose of accurate display (colours, image URLs, pixel sizes, etc.) but it is not included here as it is irrelevant to the functionality of the program.

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| grid | ArrayList<ObservableList<boolean>> | The grid, which stores the entire state of the automaton. Can only be inserted into at indices within the bounds of width and height. The inner lists are observable so that the on-screen cells can know when a value is updated. |
| columns | int | The width of the grid, used when constructing it initially. |
| rows | int | The height of the grid, used when constructing it initially. |
| do\_wrap | boolean | Determines whether the grid treats edge cells as adjacent to their counterparts on opposite edges. |
| generation | SimpleIntegerProperty | Incremented once per tick, to show the user how many have passed. It’s a property (which is observable) so that the display can detect changes. |

# 5. COMPONENT DESIGN

## 5.1 The Grid

The grid has effectively no functionality in-built and relies entirely on other subsystems to make changes to itself. It does, however, contain all the data specified in section 4.

## 5.2 The Simulator

The simulator needs only one function, called step. This function iterates over all cells and tests whether they should be alive or dead in the next step. It uses a buffer array to do this, as this determination relies on the states of a cell’s neighbours and altering them mid-computation would invalidate the results. Below is some pseudocode detailing this procedure:

function step() {

let grid\_copy = a copy of the current grid

for each row (with y position) in the grid:

for each cell (with x position) in the row:

let neighbours = 0

for dx in -1, 0, 1:

for dy in -1, 0, 1:

// Some check should be involved to ensure that

// no indexes are made that would go outside the

// normal confines of the grid.

if grid\_copy[y+dy][x+dx]:

neighbours += 1

grid[y][x] = neighbours == 3 || (neighbours == 4 && cell)

// This is equivalent to the standard rules (n=3 || [n=2&&alive])

// since we include the cell itself when counting living neighbours.

}

This function is called whenever the clock detects that there has been sufficient delay since the last tick.

## 5.3 User Interface

The user interface contains the main function of the program, and it is the core of its functionality. The user interface’s main function is some minimal setup code (mostly for the grid) followed by the initialization of the window and the running of the main loop. It works as follows:

function main() {

set up the grid and its related values

create a screen object and give it all the components it needs (buttons, etc.)

assign each clickable component a function that lets it interact with the grid or

other values in the way it needs to.

display the screen to the user and let them interact with it

once the user presses exit, end the program

}

## 5.4 File I/O

The file I/O system handles interaction with the computer’s filesystem and the grid, and it does so by using two functions: save and load. The save function takes the grid, creates a binary representation of it that can be saved to disk, and does so. The load function does the inverse, taking binary data and reconstructing a grid from it. The two functions work like this:

function save(filename) {

open a file with the given name

write (the number of rows and columns in the grid)

write (1 if wraparound is enabled else 0)

write (generation index)

write (the binary representation of the grid itself)

save and close the file

}

function load(filename) {

let bytestream = the contents of the given file

rows = the first int in the bytestream

columns = the next int in the bytestream

wraparound = the next int in the bytestream

generation\_index = the next int in the bytestream

grid = a 2d array of the size we just read

for each row in the grid:

for each cell in the row:

cell = the next int in the bytestream

close the file

}

The specification for a .cgol file is as follows, with “1 int” representing the amount of space an integer takes up when written to a file:

|  |  |  |
| --- | --- | --- |
| **Name** | **Size** | **Description** |
| Rows | 1 int | The number of rows the grid uses. |
| Columns | 1 int | The number of columns the grid uses. |
| Do Wraparound | 1 int | Whether the grid has wraparound enabled or not. 0 if no, 1 if yes. |
| Generation Index | 1 int | The index of the last tick to be run on the grid. |
| Grid Data | Variable | 1 int for every cell in the grid, which is 0 if the cell is dead and 1 if it’s alive. The cells go in order starting from the top left and moving down, then shifting right when the bottom is reached. |

# 6. HUMAN INTERFACE DESIGN

## Overview of User Interface

The user interface is controlled and accessed entirely by its respective component, and so it contains the following elements (as previously specified):

* A display of the grid, with cells which toggle their state when clicked (middle).
* Buttons to pause and play the simulation, as well as a slider for the number of ticks per second (top left).
* Buttons to save and load grids using the file I/O subsystem (bottom left).

## Screen Images

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

# 7. REQUIREMENTS MATRIX

## 7.1 Requirements

|  |  |
| --- | --- |
| **Requirement Code** | **Description** |
| *REQ-1: Simulation* | |
| *REQ-1.1* | The program must accurately simulate Conway’s Game of Life. |
| *REQ-1.2* | The program must display its simulation in a convenient and easy-to-view format. |
| *REQ-2: User Inputs* | |
| *REQ-2.1* | The program must allow the user to generate grids of a given size filled with random cells. |
| *REQ-2.2* | The program must allow the user to view and run grid configurations from external locations, by way of a file-reading system. |
| *REQ-3: Filesystem Saving/Loading* | |
| *REQ-3.1* | The program must use a custom .cgol file format for saving and loading its grids and should be capable of processing and creating files of that format. |
| *REQ-4: Efficiency* | |
| *REQ-4.1* | The program must run at a fast enough speed that the user does not notice any delay on reasonably sized grids. |
| *REQ-5: Simulation Timings* | |
| *REQ-5.1* | The program must allow the user precise control over when steps are taken in the simulation, with play, pause, step, and speed controls available. |

## 7.2 Component Matrix

|  |  |
| --- | --- |
| **Component** | **Associated Requirements** |
| Grid | REQ-1.1 |
| User Interface | REQ-1.2, REQ-2.1, REQ-2.2, REQ-5.1 |
| Simulator | REQ-1.1, REQ-4.1 |
| File I/O | REQ-2.2, REQ-3.1 |