Java Programming

**Name:** Colm Kearney

**Assignments:** Assignment 1 – Finish Conway’s Game of Life from template provided.

Contents

1.0 Assignment 1 4

1.1 Program Summary 4

References: 13

‌1.2 Issues Encountered & Solutions 14

1.3 Testing 17

Test 1: Start button with user inputted pattern 17

Test 2: Start and random button 20

Test 3. Save and load button 23

Test 4. Second Load test. 25

Appendix 1 29

Assignment 1 29

Source Code: ConwaysLife.Java 29

# 1.0 Assignment 1

**Part 1:**

Finish Conway’s Game of Life, based on the template and code snippets provided. The tasks will be as follows:

|  |  |
| --- | --- |
| - | Add game states (Playing and Not Playing). |
| - | When the game is in the 'not playing' state, render two rectangles as ‘buttons’: one for Start and one for Random.  Modify the mousePressed method so that it checks for clicks on the buttons' regions.  The buttons you create should have the following functionality:  Start: switches the game state to ‘playing’  Random: randomises the game state |
| - | When in the 'playing' state, apply the rules of Conway’s Game of Life at each repaint. |
| - | Implement game state Loading and Saving (via ‘buttons’ as above).  Finally, how will you encode the game state as a string?  Explain your answer. |

## 1.1 Program Summary

To begin this question I reviewed the existing template and added a Boolean playing to control the game state later in the program and array called buttons to hold the buttons in. I also created the rectangle buttons in the constructor. This can be seen in the screenshot in Fig. 1.1 below:

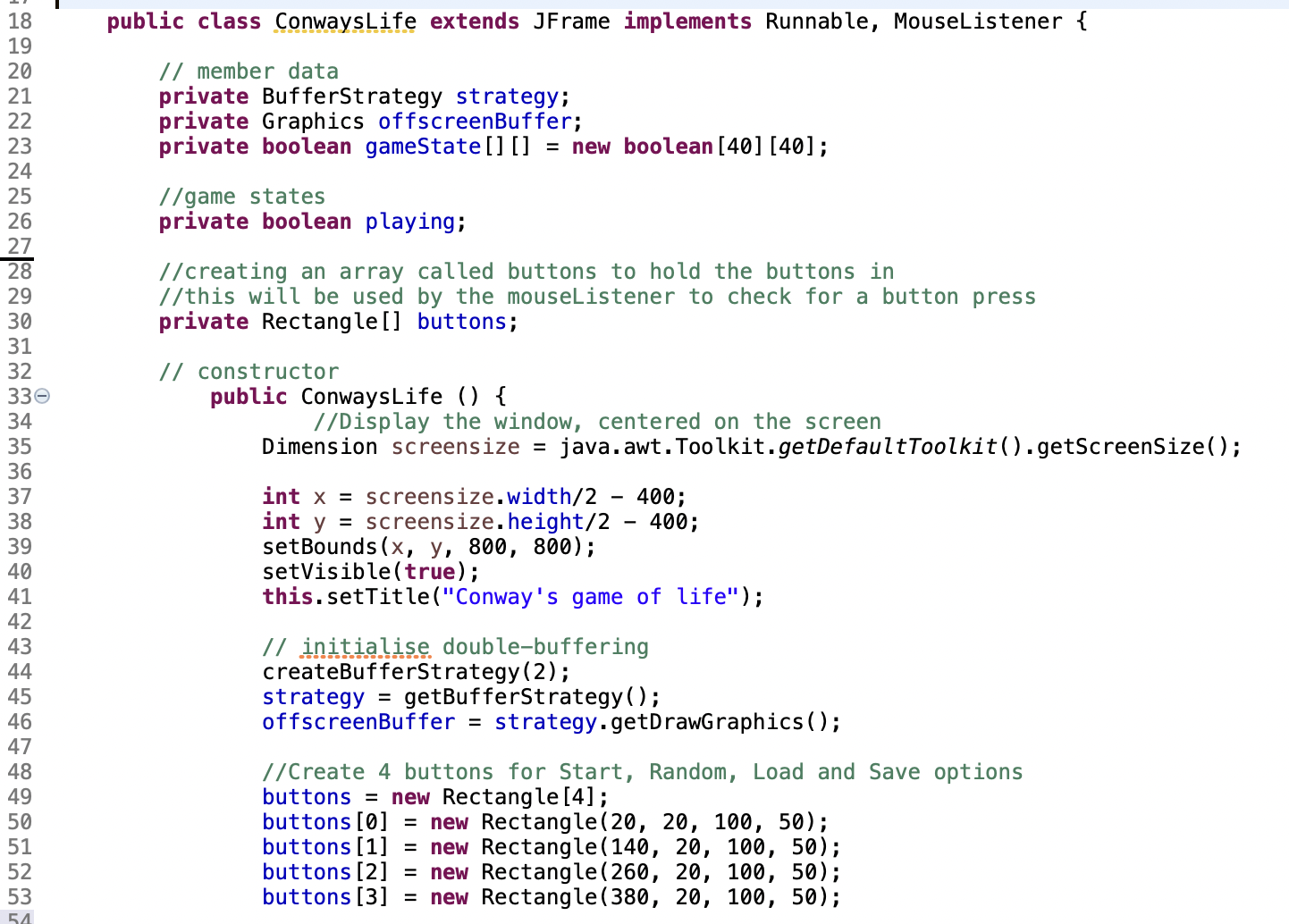


Fig. 1.1 Variables and Constructor.

Before approaching the list of tasks outlined in the question, I first needed to complete the code for initializing the game state outlined in slide 15 of the section 2 notes for this week. The sample code in the slides can be seen in Fig.1.2 below:

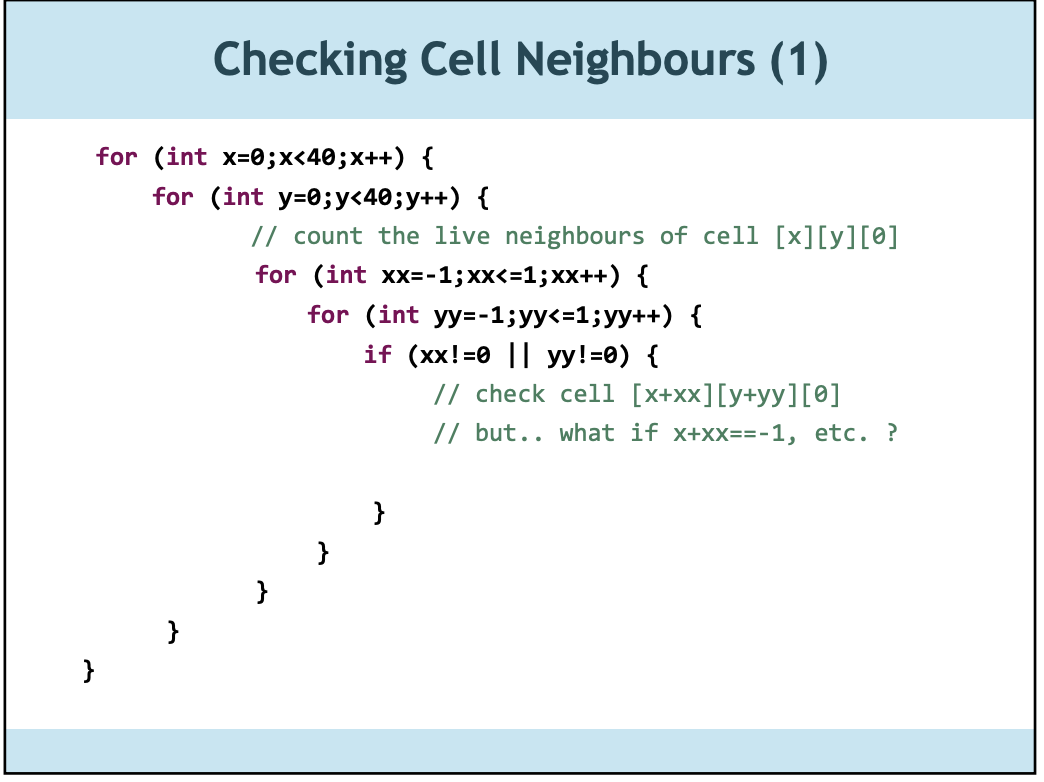


Fig. 1.2 Original sample

The notes below this outlined the following requirements:

* We need to make sure the value won’t go above or below the boundaries of our board (less than 0, greater than 39).
* we need to set up logic to check if cell [xx][yy][0] is alive, keep track of how many surrounding cells are alive total, and then we need to decide if cell [x][y] should be alive or dead

I have done this by implementing the following code in the constructor method:



Fig. 1.3

The code above in Fig. 1.3 is as the notes mention the core of the Game of Life game, as it calculates the new state of the game board for each generation. The first for loop to count the live neighbors was provided and I expanded upon this. The outer for loop iterates over the rows of the board, while the inner for loop iterates over the columns of the board. For each cell, the code counts the number of live neighbors by checking the eight adjacent cells (up, down, left, right, and diagonals) and incrementing a counter for each live neighbor.

After counting the live neighbors, the code checks the current state of the cell (gameState[i][j]) to determine whether it should be alive or dead in the next generation. If the cell is currently alive and has fewer than two or more than three live neighbors, it will die in the next generation (its state will be set to false). If the cell is currently dead and has exactly three live neighbors, it will become alive in the next generation (its state will be set to true).

At the end of this code segment a new thread is created and in the following line it is started. This will lead into the next section which covers the run() method.

For the next part, I needed to expand the run() method of the program. The while(true) statement and delay were pre-configured in the template. I then edited the code beneath this. First I added an if statement around the entire block to check if the Boolean playing was true or false. This is required for the start button functionality. Later I will discuss how the start button is able to toggle playing states.

If playing is true, the code creates a new 2D boolean array called newState with dimensions of 40x40. This array will be used to store the new state of the game after applying the rules.

The for loops are used to iterate over each cell in the current gameState array (also 40x40). For each cell, the code computes the number of live neighbors surrounding it. This can be seen in Fig. 1.4 below:



Fig. 1.4 Updated Run() method.

The nested for loops inside of the for (int i = 0; i < 40; i++) loop iterate over each cell in a 3x3 grid centered at the current cell. The if statement checks that the indices are within the bounds of the gameState array and that the current cell is not included in the count of neighbors. If the cell is alive, the count of liveNeighbors is incremented.

After computing the number of live neighbors, the code applies the rules of Conway's Game of Life. If the current cell is alive and has fewer than two or more than three live neighbors, it dies in the new state. If the current cell is dead and has exactly three live neighbors, it comes to life in the new state.

Lastly, the newState array is copied into the gameState array to update the state of the game.

I mention below in section 1.2 of my struggles with Jbuttons and how I re-read the question to realise that I needed rectangles. I then focused on creating the rectangles at the top of the page. I created the following methods: drawButtons and getButtonText as well as updating the paint method. The updated paint method can be seen below in Fig. 1.5:



Fig. 1.5 updated paint method()

In the paint method I now implement a for loop to create the four green rectangles, then add labels to the buttons and the rest of the code redraws the game objects as before. The methods for drawButtons and getButtonText are below in Fig. 1.6:



Fig. 1.6 drawButtons and getButtonText

The first button I attempted to get working was the random button. This functionality required the import of java.util.Random; and I needed to create a new random object inside the switch statement inside the mouseListener. I added the code for this directly into the switch statement contained within the mouseListener program. The code for this is below:

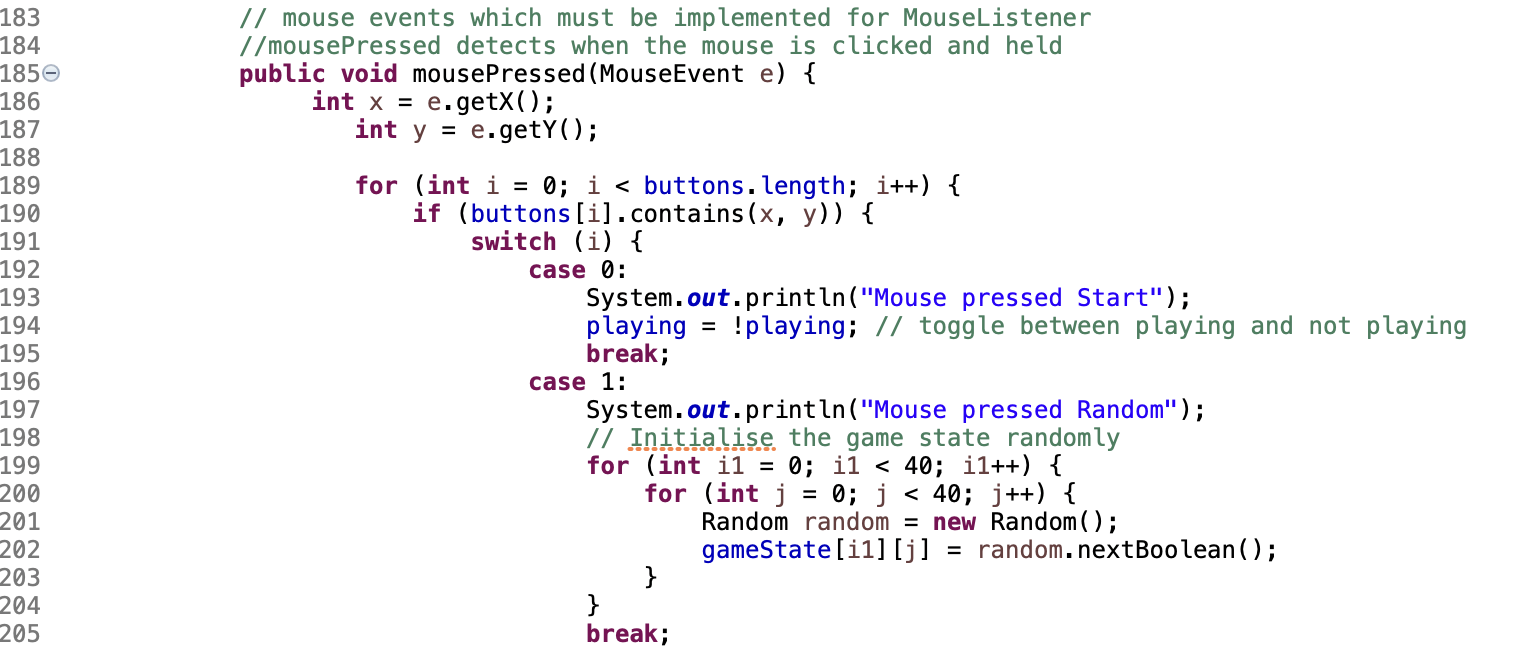


Fig. 1.7 Start and Random button functionality.

The start button just has a simple one line playing - ! playing. This is due to the previous effort of wrapping the majority of the run() method with an if statement to check if playing is true. Case 1 is for the random button. In this code, The two nested loops iterate over the entire game board, setting each cell to a randomly generated boolean value.

The Random class is used to generate random boolean values. The nextBoolean() method of this class returns a randomly generated boolean value that is equally likely to be true or false. The Random object is created once for each cell and is used to generate the boolean value for that cell. The output of this can be seen below in Fig. 1.8:



Fig. 1.8 Result of using Random button.

The image above in Fig. 1.8 is produced as a still image. When the user then presses the start button, the animation begins and works as expected. The user can then press the start button again to stop the animation. I found it was best to stop the animation before pressing the save button and save the image to a file.

Once the user clicks the save button, the code within the switch statement of case 3 for the save functionality is activated. Inside the method, the program checks which button was clicked using a switch statement.

When the Save button is clicked, the program creates a new FileWriter object and passes a File object named "gameState.txt" to its constructor. This creates a new file (or overwrites an existing one) named "gameState.txt" in the current directory.

The program uses two nested loops to iterate over the gameState array, which stores the current state of the game board. For each element in the array, the program writes either a "0" or a "1" to the file using the write() method of the FileWriter object. The value written depends on whether the corresponding cell in the game board is alive or dead: "1" is written for a live cell, and "0" is written for a dead cell. I deliberately choose to write a 0 or a 1 to the file as this was mentioned in the notes for this week’s assignment. I could have written a true or false here also. The screenshot for the example can be seen below in Fig. 1.9:

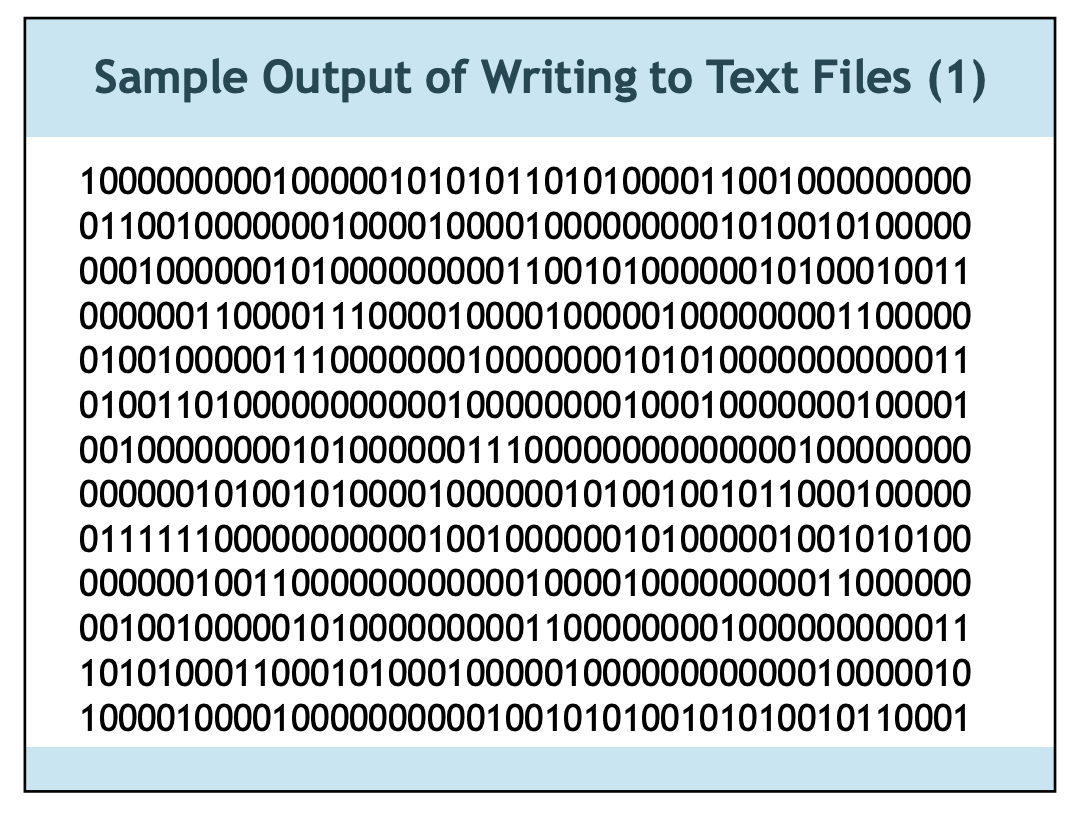


Fig. 1.9 Example of output required for save file.

The screenshot Fig. 1.10 below shows the gameState.txt file created in the working directory and Fig. 1.11 shows the contents of the file:

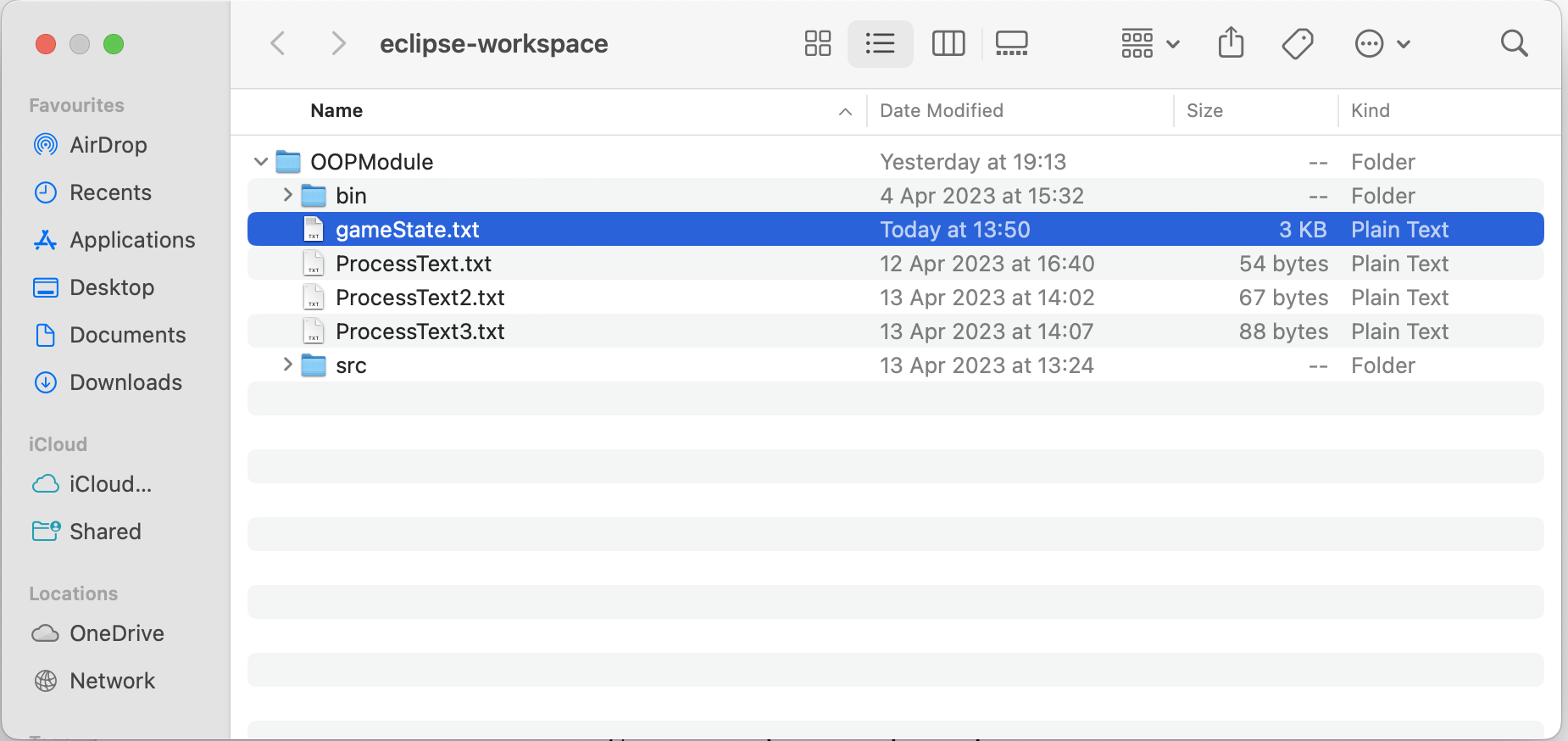


Fig. 1.10 File created in working directory

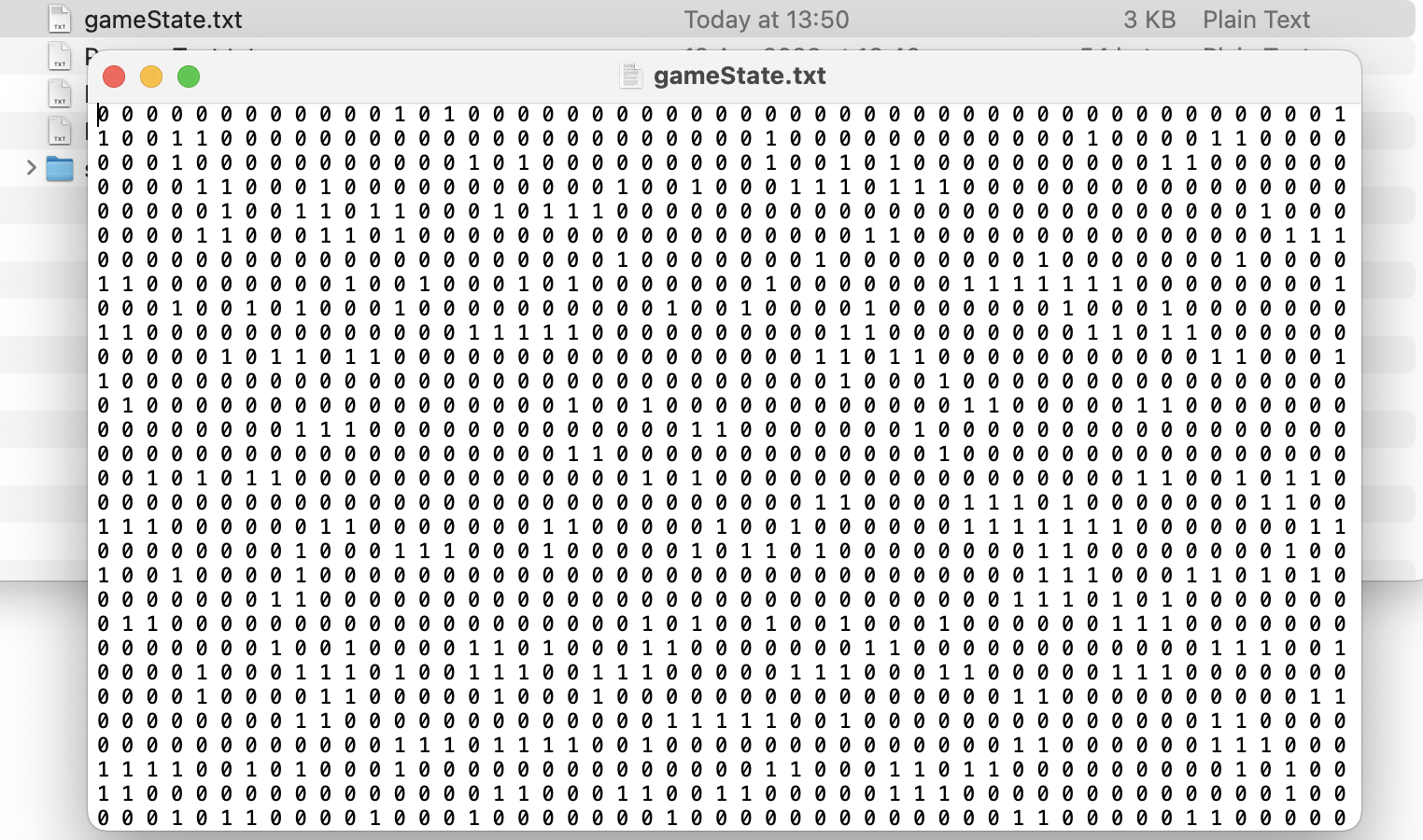


Fig.1.11 Contents of created gameState.txt file

The last button to discuss is the load button. I needed to complete the save functionality before attempting to load the file. I created a separate method loadGame and called it within the switch statement for load. I found this best as the mousePressed method was becoming quite large. This can be seen in the screenshot below in Fig. 1.12:

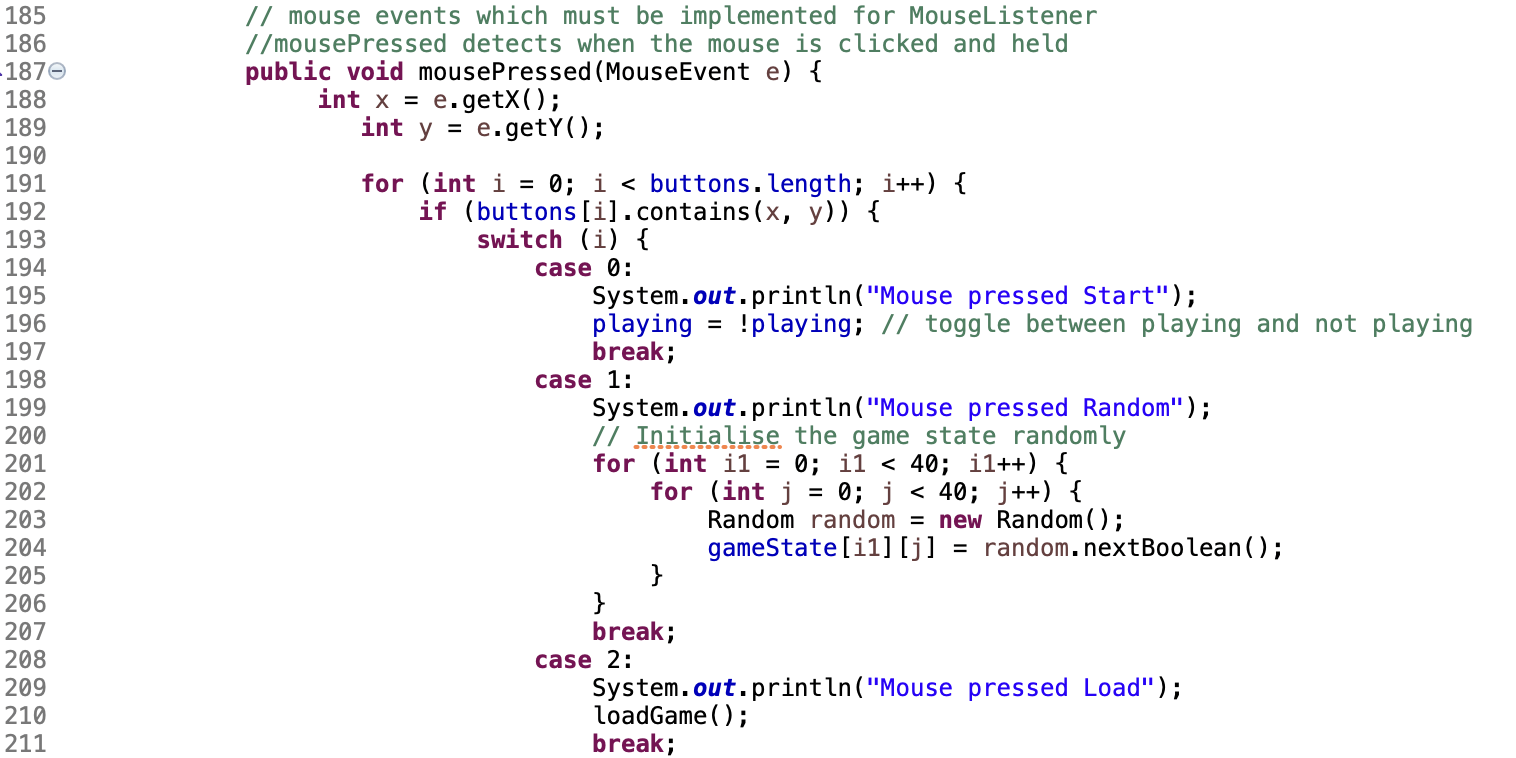


Fig. 1.12 LoadGame() call from MousePressed method.

The loadGame method essentially takes the values of the gameState.txt file by reading the file line by line, split each line into individual values using the space separator, and convert each value to a boolean. We loop through all these values and store them in the new boolean array newGameState. I then set gameState equal to newGameState and by doing this I overwrite the previous values of gameState with the new values from the file. Creating the second boolean array simplifies the code and removes the need to clear the gameState array and then load the new values into it. The loadGame method can be seen in the screenshot in Fig. 1.13below:

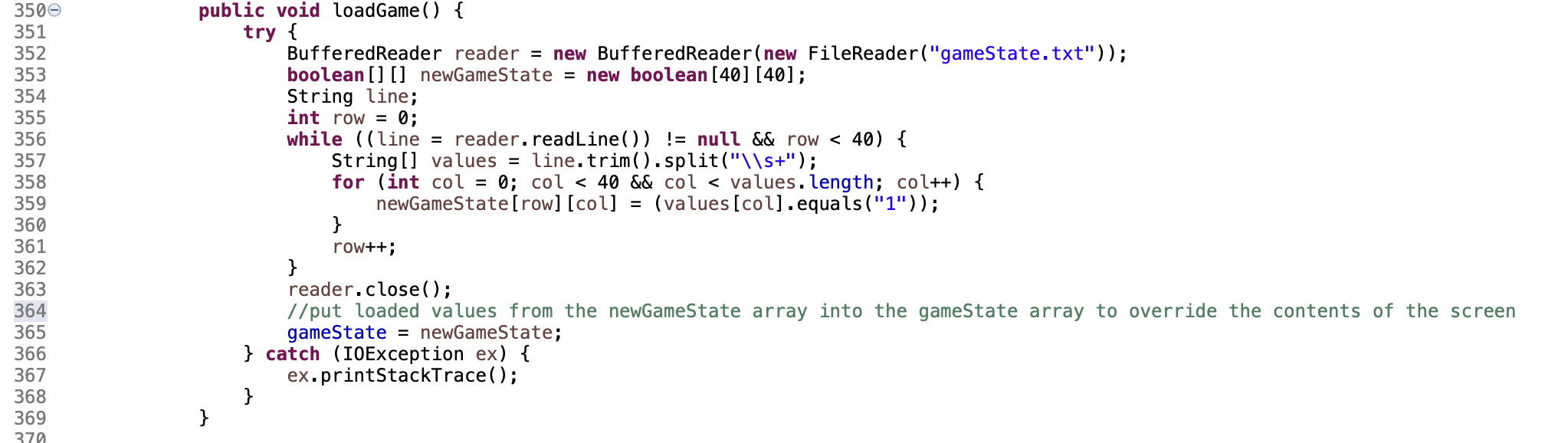


Fig. 1.13 loadGame()

## References:

conwaylife.com. (n.d.). *Conway’s Game of Life*. [online] Available at: <https://conwaylife.com/>. [Accessed 24 Apr. 2023]

Cellular Automata. (2023). *What is a cellular automaton (CA) and what is it used for?* [online] Available at: <https://www.techtarget.com/searchenterprisedesktop/definition/cellular-automaton#:~:text=A%20cellular%20automaton%20(CA)%20is>. [Accessed 24 Apr. 2023]

‌Oracle.com. (2019). *Painting in AWT and Swing*. [online] Available at: <https://www.oracle.com/java/technologies/painting.html>. [Accessed 25 Apr. 2023]

‌www.digitalocean.com. (2023). *How To Write Conditional Statements in Java | DigitalOcean*. [online] Available at: <https://www.digitalocean.com/community/tutorials/how-to-write-conditional-statements-in-java> [Accessed 25 Apr. 2023].

‌www.tutorialspoint.com. (2023). *Java - Files and I/O - Tutorialspoint*. [online] Available at: <https://www.tutorialspoint.com/java/java_files_io.htm>. [Accessed 27 Apr. 2023]

www.classes.cs.uchicago.edu.(2023). *HappyCoding.Generated.IfStatements*. [online] Available at: <https://www.classes.cs.uchicago.edu/archive/2021/spring/11111-1/happycoding/p5js/if-statements.html#:~:text=An%20if%20statement%20checks%20a>. [Accessed 27 Apr. 2023]

Java2Blog. (2022). *How to Write Array to File in Java - Java2Blog*. [online] Available at: <https://java2blog.com/write-array-to-file-java/> [Accessed 29 Apr. 2023].

‌www.tutorialspoint.com. (2023). *How to store the contents of arrays in a file using Java*. [online] Available at: <https://www.tutorialspoint.com/How-to-store-the-contents-of-arrays-in-a-file-using-Java> [Accessed 29 Apr. 2023].

www.java2s.com. (2023). *Use FileWriter to write an array of strings to a file. : FileWriter «File «Java Tutorial*. [online] Available at: <http://www.java2s.com/Tutorial/Java/0180__File/UseFileWritertowriteanarrayofstringstoafile.htm> [Accessed 29 Apr. 2023].

‌

‌

‌

## ‌1.2 Issues Encountered & Solutions

By far the most difficult part of the program I found was creating the buttons for the interaction with the user. Originally, I misread the program and attempted to implement JButtons. I went down this path for quite some time and created separate JPanels for the game and the buttons and added these to the JFrame. I found I was ultimately struggling to figure out how to avoid painting over the Jbuttons. Once I re-read the question and saw that I had to create “rectangles as buttons” I was able to make progress. The rectangles were painted on the screen and look permanent. This can be seen below:



Fig. 1.14 Updated Paint method()

I also was able to modify the mousePressed() below to detect if the mouse clicked within the rectangle. I would later add more functionality but this was a big win at the time. In the screenshot below the switch statements all have a system.out.println call to output to the console when a button is clicked on. I utilised these statements when testing to see if the mouse click was registering and kept these in as the program progressed.

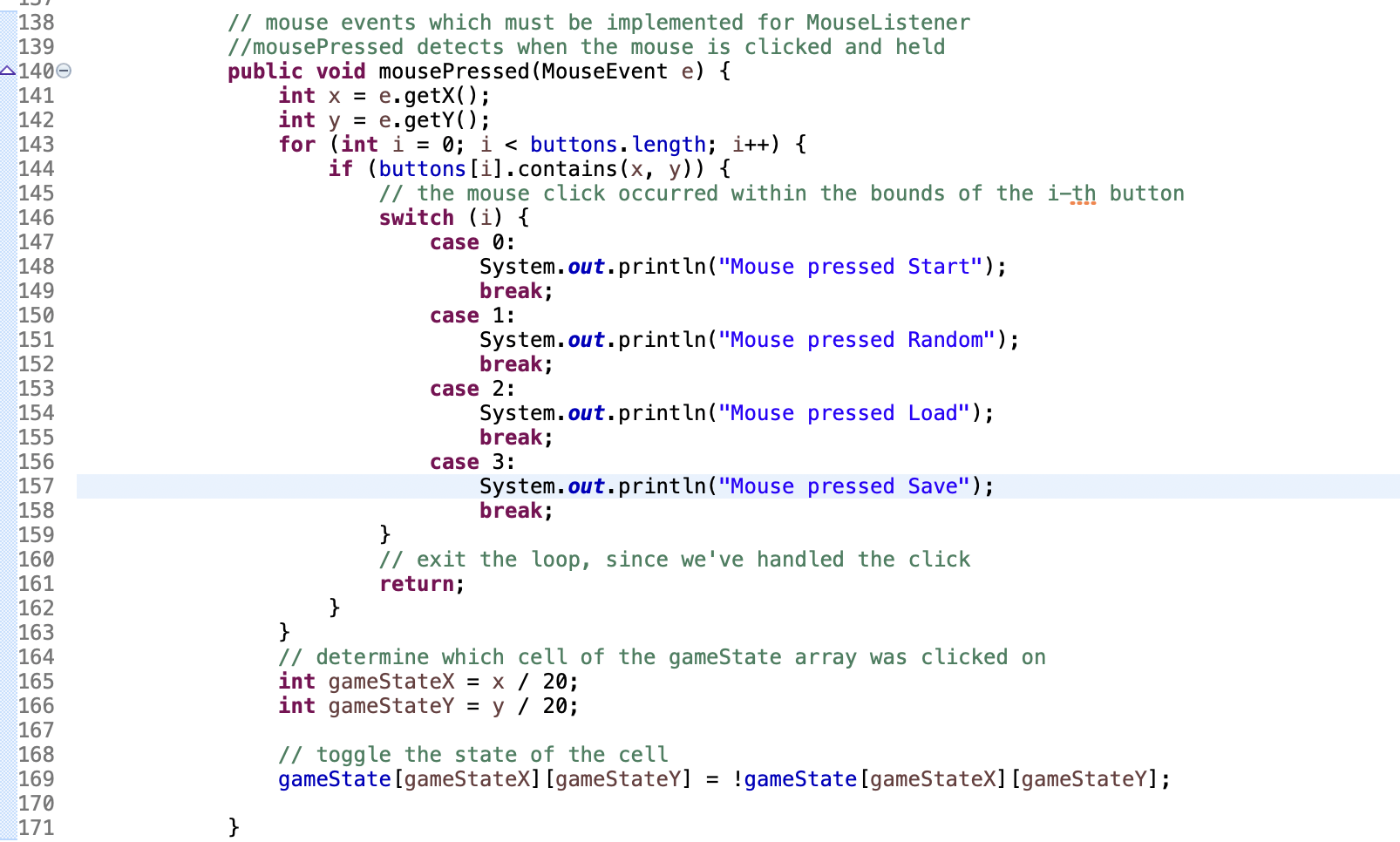


Fig. 1.15 Early mousePressed method

## 1.3 Testing

Testing for this program involved checking the following items:

* The start button causes the animation to begin and stop
* The random button changes whatever pattern is on the screen to a random set of white squares.
* The save button can create/overwrite a file called gameState.text and save the pattern on the screen to 1’s and 0’s.
* The load button can take a file called gameState.txt from the working directory, read the 1’s and 0’s, convert these to Booleans true or false and then create the correct pattern on the screen.

The test details are below:

### Test 1: Start button with user inputted pattern

The start button functionality controls the animation in this game. When the program originally loads the end user can select certain squares and change them to white. Once there pattern is selected, the end user then selects the start button to watch the animation begin. In Fig. 1.16 I have a pattern selected by the end user and Fig.1.17 and Fig. 1.18 show various evolutions of this pattern.

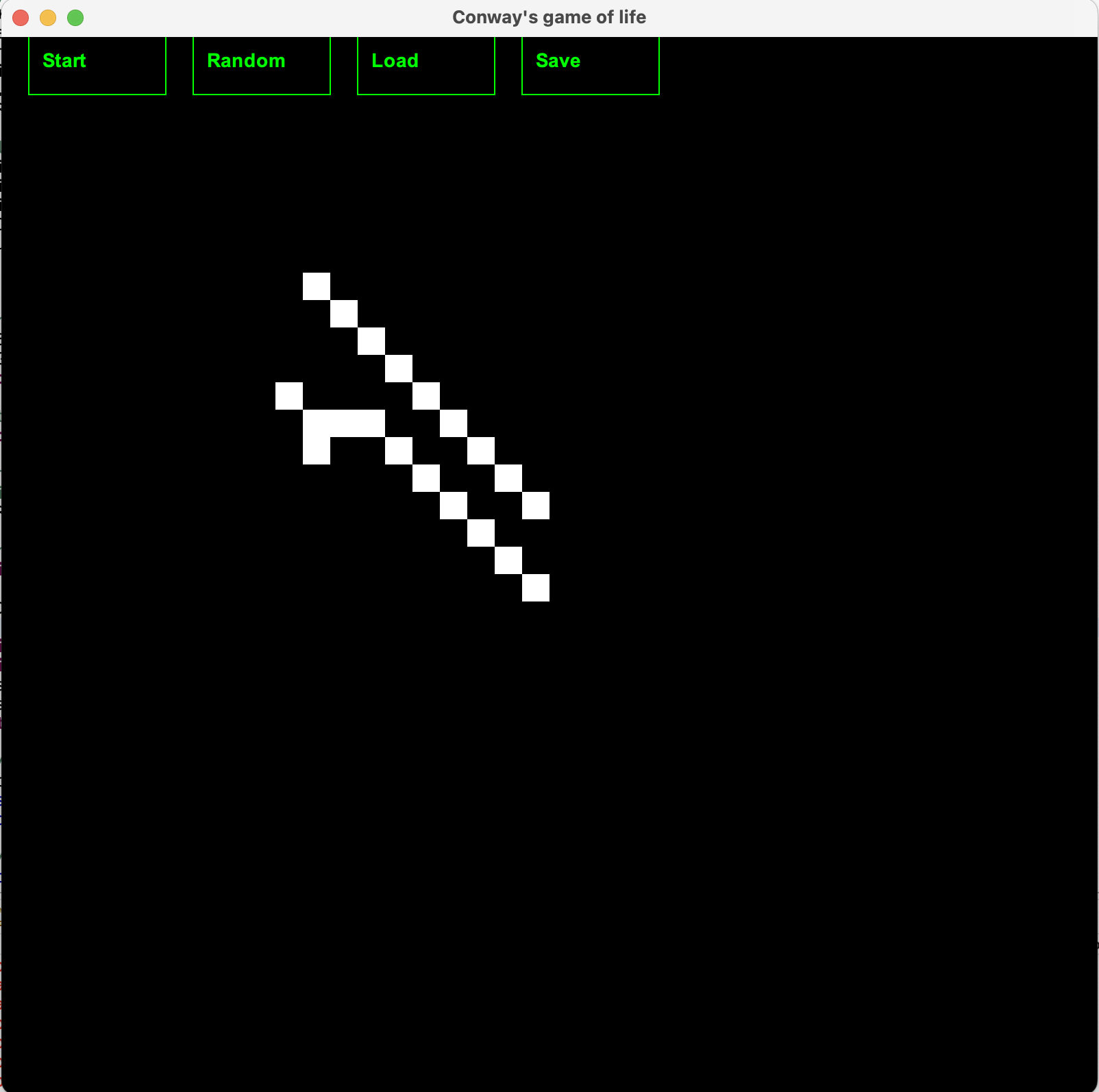


Fig. 1.16 starting pattern

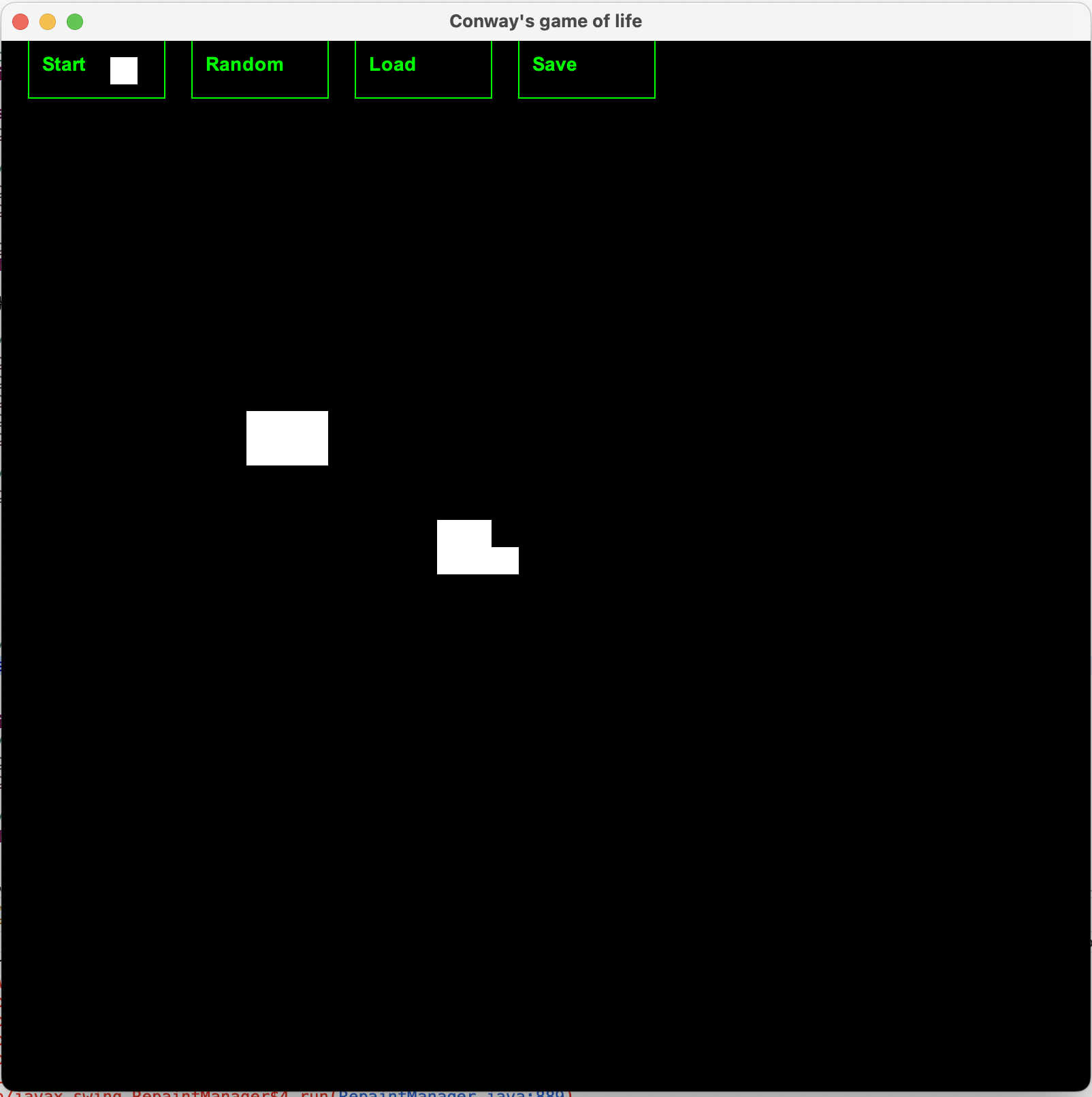


Fig. 1.17 pattern evolution 1

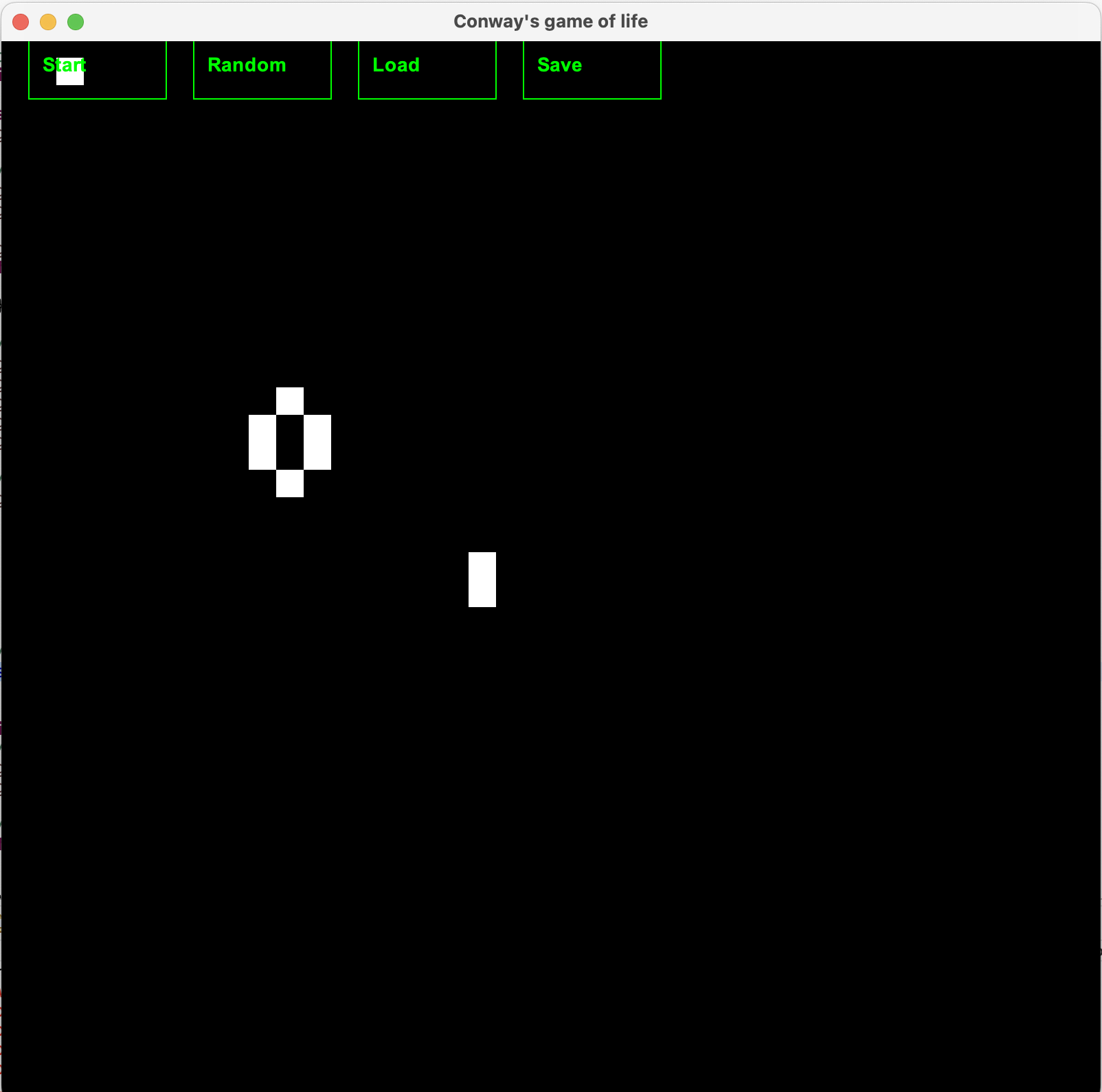


Fig. 1.18 pattern evolution 2

### Test 2: Start and random button

The start button can also be used with the random button. First the end user can select the random button to produce a random pattern on the screen. This can be seen below in Fig. 1.19:

### 

Fig. 1.19 Random pattern

Once the pattern is created the end user can press start to watch the pattern evolve. Some of the evolutions can be seen below in Fig. 1.20 and Fig. 1.21:

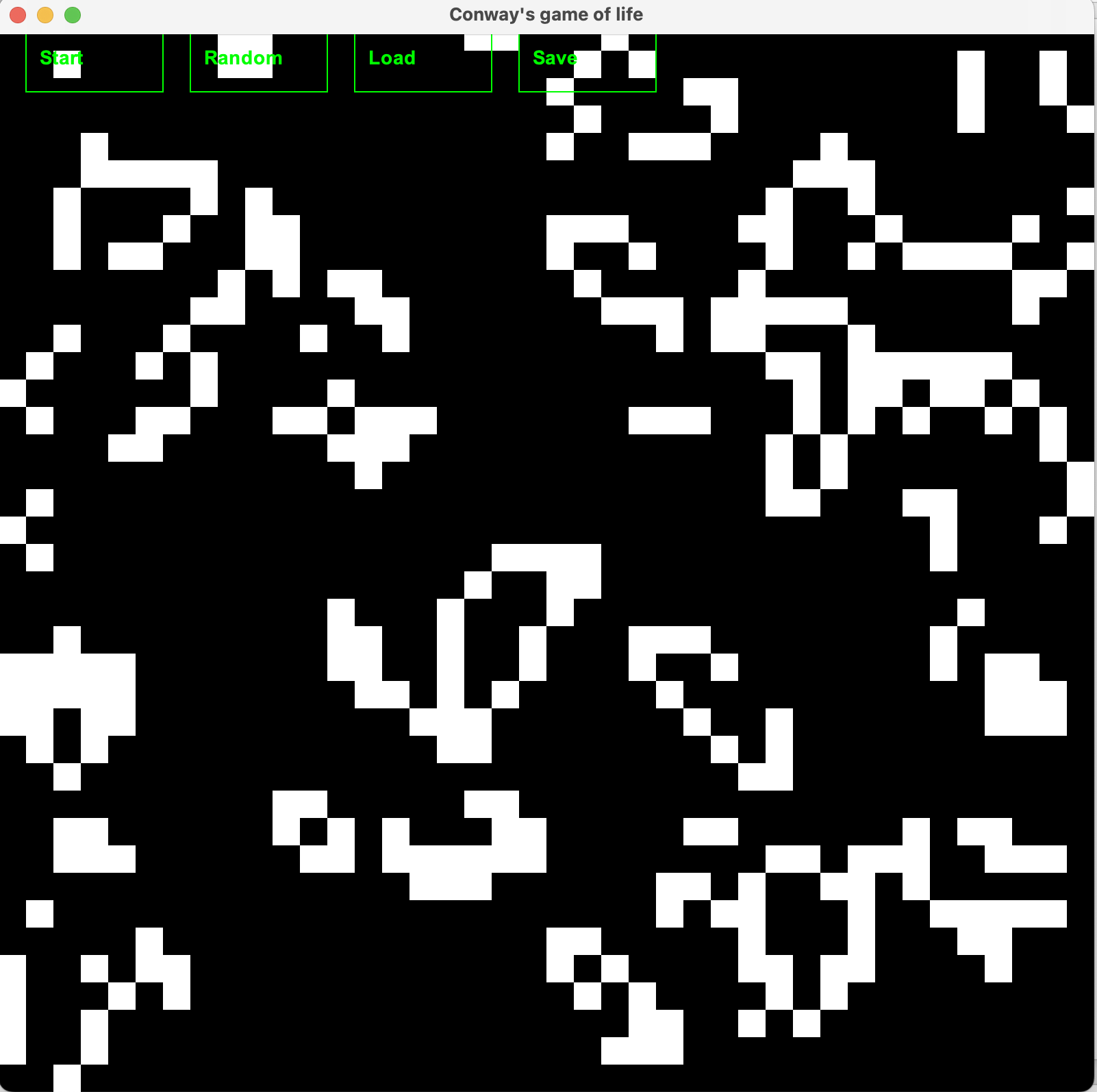


Fig. 1.20 Evolution 1

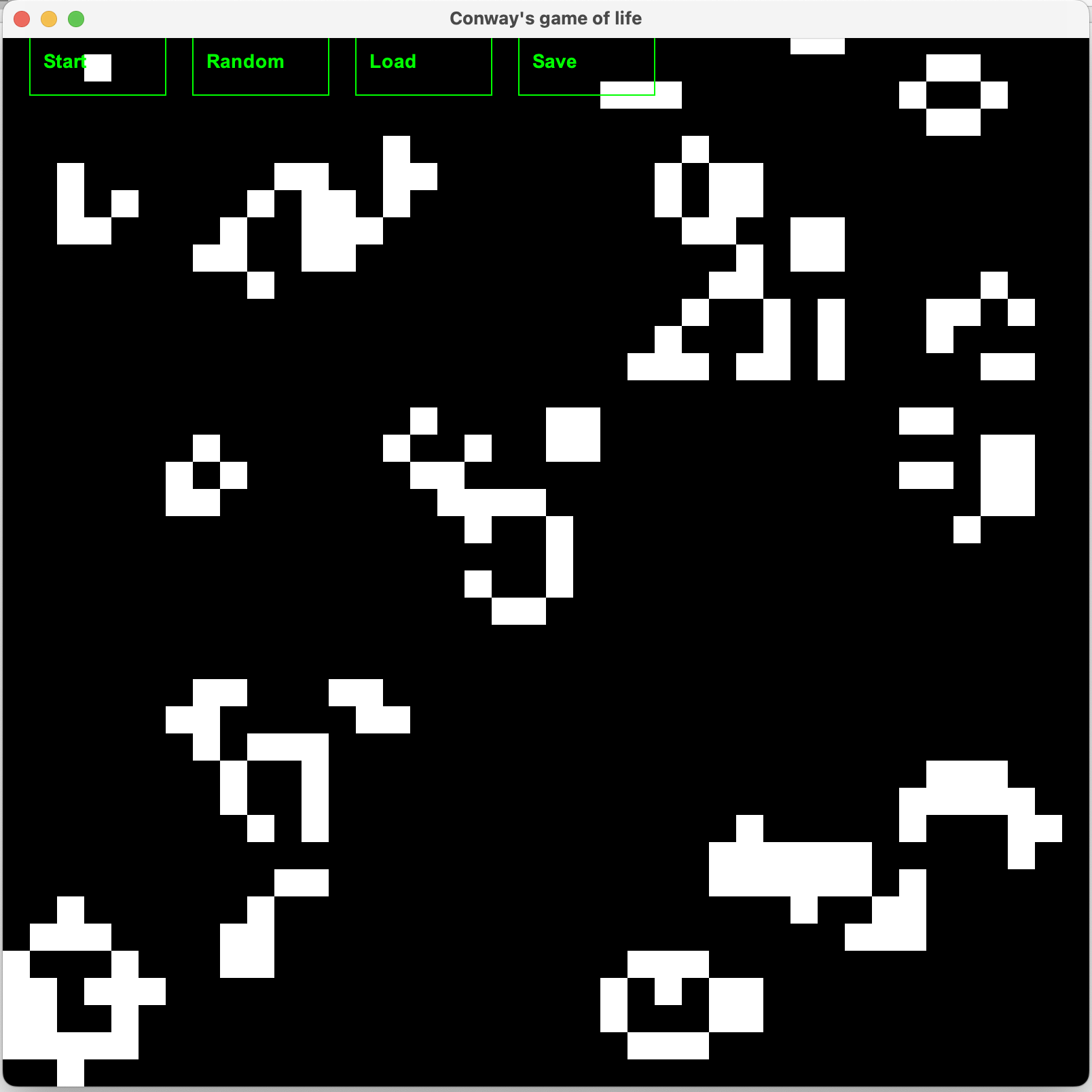


Fig. 1.21 Evolution 2

### Test 3. Save and load button

Next, I clicked save to take the stopped pattern in Fig. 1.21 and save it to the gameState.txt file. Then I click random to put a different pattern on the screen. The different pattern can be seen in Fig. 1.22 below:



Fig. 1.22 random pattern

I then click the load button to load the pattern from Fig. 1.21 to the screen. The outcome is below in Fig. 1.23:

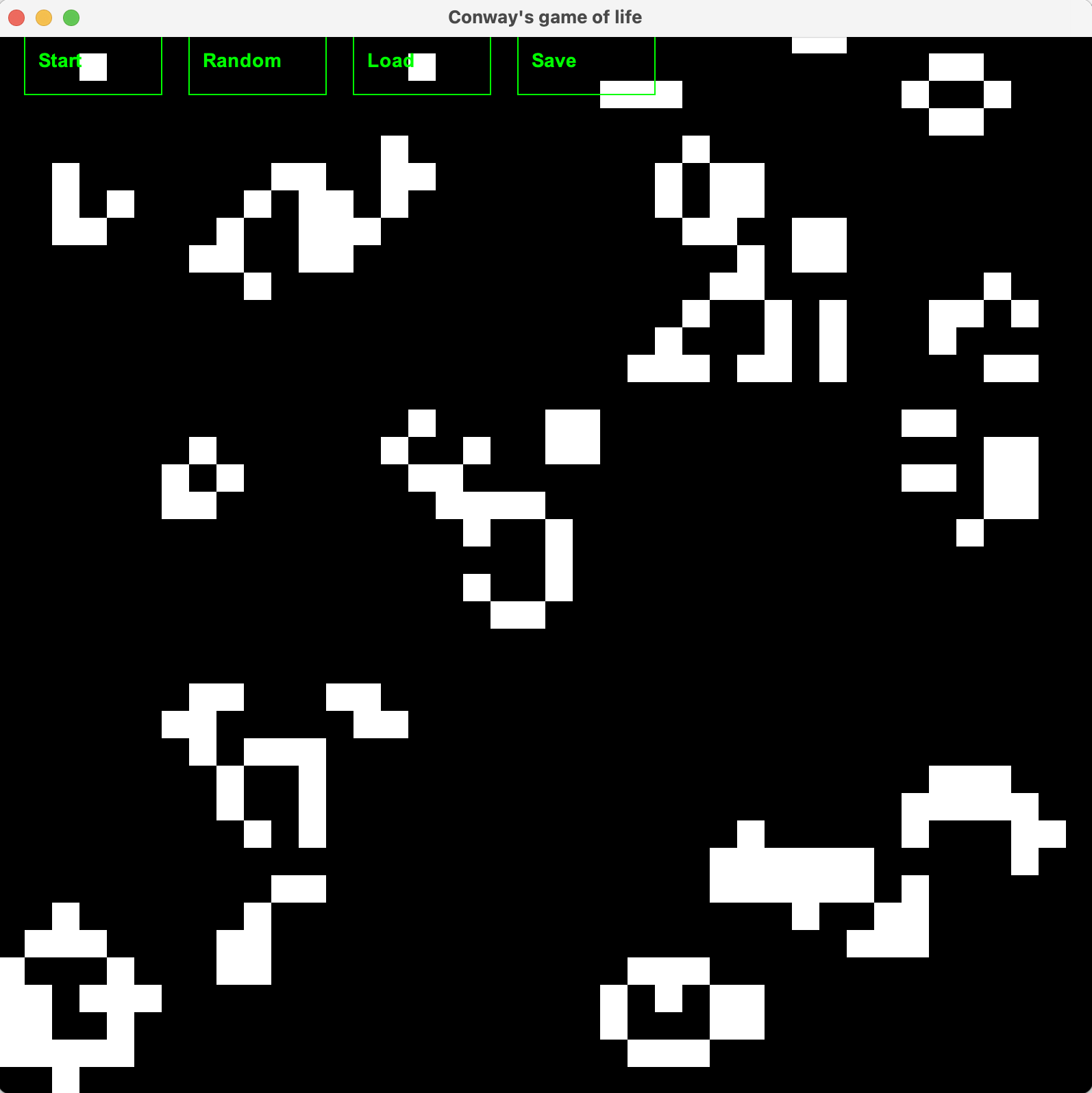


Fig. 1.23 loaded pattern.

The pattern appears to be the same. However, with a lot of white squares it’s difficult to know if it is exactly correct. I did another test with just ten white squares in the next test to be more conclusive.

### Test 4. Second Load test.

First, I started the program and just placed ten individual white squares on the screen as can be seen in Fig. 1.24 below:

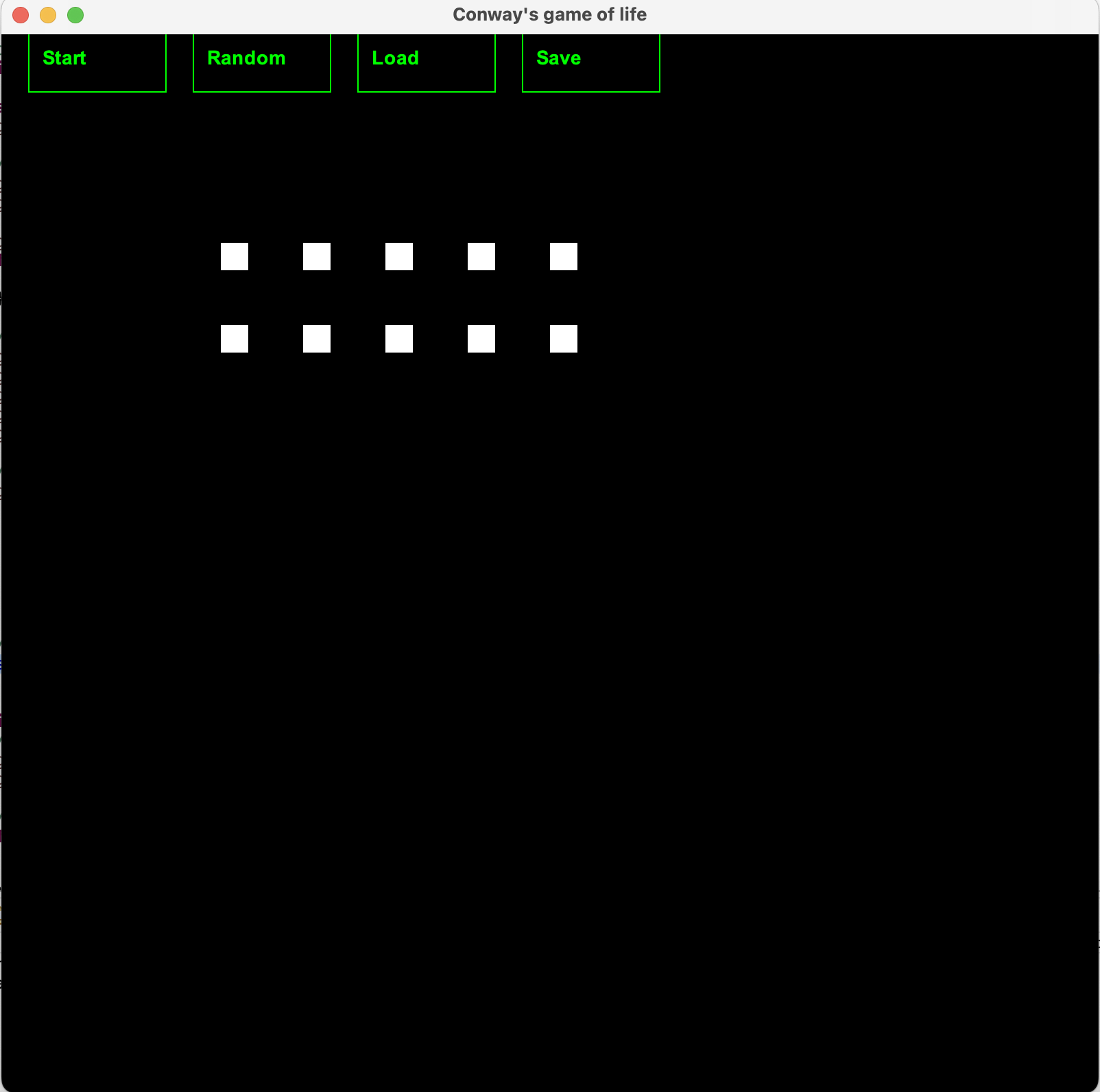


Fig. 1.24 Test 4: 10 squares

Next, I click save to write this pattern to gameState.txt. Then I click random to produce a different pattern. The random pattern can be seen below in Fig. 1.25:



Fig. 1.25 Test 4: Random pattern

Then I click the load button to bring back the contents of the saved file. This should be the ten white squares in the pattern that I set in Fig. 1.24. The result can be seen in Fig. 1.26:

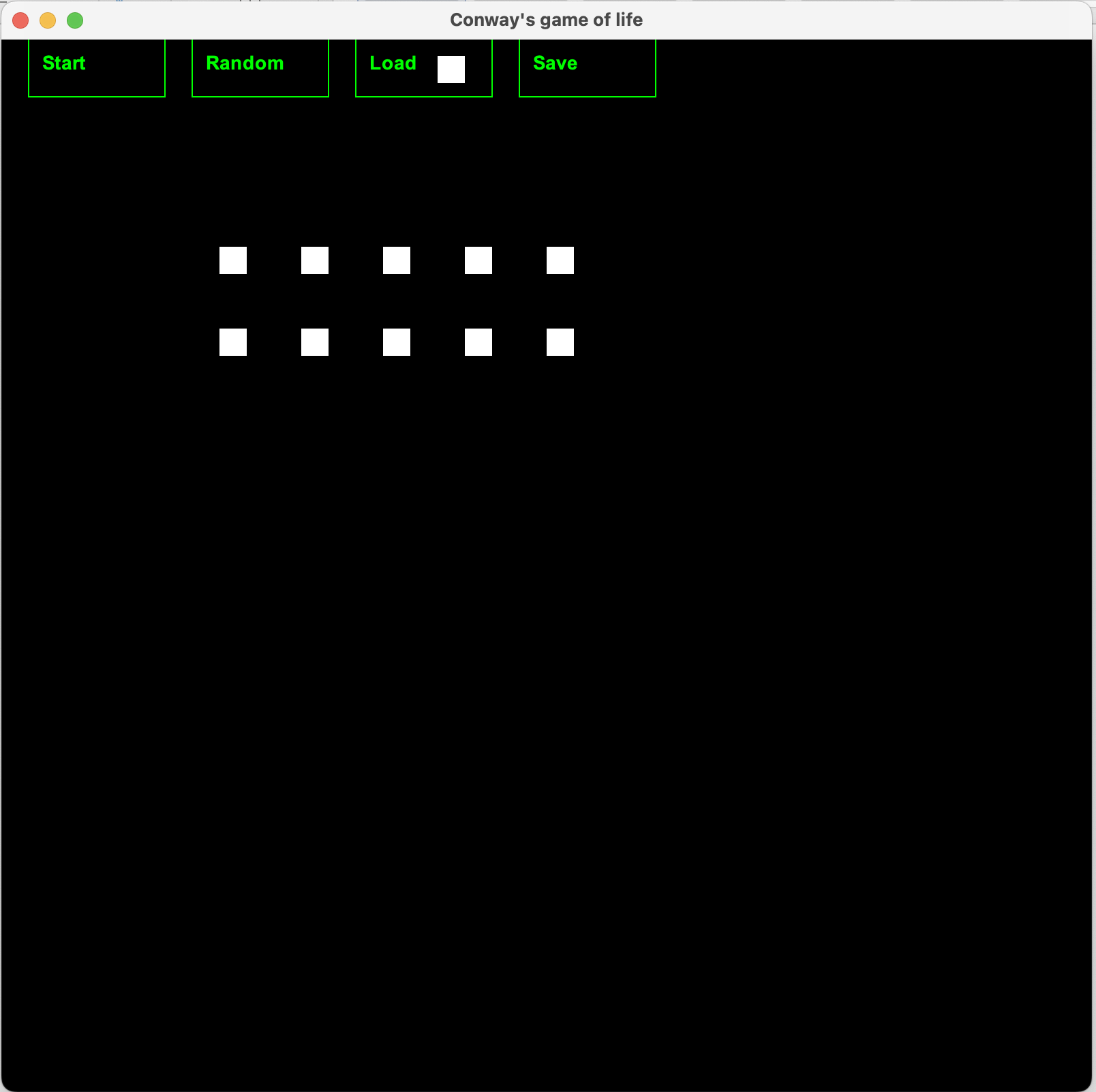


Fig. 1.26 Test 4: Load results.

The pattern has been loaded successfully to the screen. One issue is that when I clicked the load button, this also registered as a user changing the cell to active which actually puts 11 white squares in the JFrame. This is something I was hoping to avoid with my original thought of using JButtons.

# Appendix 1

# Assignment 1

# Source Code: **ConwaysLife.Java**

package Wk2;

//Starting Conway’s Game of Life

//The main application class (single instance)

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import java.awt.image.\*;

import java.util.Random;

import java.awt.Rectangle;

//imports for file handling

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.io.BufferedReader;

public class ConwaysLife extends JFrame implements Runnable, MouseListener {

// member data

private BufferStrategy strategy;

private Graphics offscreenBuffer;

private boolean gameState[][] = new boolean[40][40];

//game states

private boolean playing;

//creating an array called buttons to hold the buttons in

//this will be used by the mouseListener to check for a button press

private Rectangle[] buttons;

// constructor

public ConwaysLife () {

//Display the window, centered on the screen

Dimension screensize = java.awt.Toolkit.getDefaultToolkit().getScreenSize();

int x = screensize.width/2 - 400;

int y = screensize.height/2 - 400;

setBounds(x, y, 800, 800);

setVisible(true);

this.setTitle("Conway's game of life");

// initialise double-buffering

createBufferStrategy(2);

strategy = getBufferStrategy();

offscreenBuffer = strategy.getDrawGraphics();

//Create 4 buttons for Start, Random, Load and Save options

buttons = new Rectangle[4];

buttons[0] = new Rectangle(20, 20, 100, 50);

buttons[1] = new Rectangle(140, 20, 100, 50);

buttons[2] = new Rectangle(260, 20, 100, 50);

buttons[3] = new Rectangle(380, 20, 100, 50);

// register the JFrame itself to receive mouse events

addMouseListener(this);

// initialise the game state

playing = false;

for (int i = 0; i < 40; i++) {

for (int j = 0; j < 40; j++) {

int liveNeighbors = 0;

// count the live neighbors of cell [x][y][0]

for (int xx = -1; xx <= 1; xx++) {

for (int yy = -1; yy <= 1; yy++) {

if (xx != 0 || yy != 0) {

// calculate the coordinates of the neighbor cell

int nx = i + xx;

int ny = j + yy;

// check if the neighbor cell is within the board boundaries

if (nx >= 0 && nx < 40 && ny >= 0 && ny < 40) {

// check if the neighbor cell is alive

if (gameState[nx][ny]) {

liveNeighbors++;

}

}

}

}

}

// determine if cell [x][y][0] should be alive or dead

if (gameState[i][j]) {

if (liveNeighbors < 2 || liveNeighbors > 3) {

gameState[i][j] = false; // cell dies

}

} else {

if (liveNeighbors == 3) {

gameState[i][j] = true; // cell becomes alive

}

}

}

}

// create and start our animation thread

Thread t = new Thread(this);

t.start();

}

// thread's entry point

public void run() {

while (true) {

// 1: sleep for 1/5 sec

try {

Thread.sleep(200);

} catch (InterruptedException e) {

}

//if statement to control whether the game plays or not

//this links to the toggle functionality of the start button

if (playing) {

boolean newState[][] = new boolean[40][40];

// compute the new state of the board

for (int i = 0; i < 40; i++) {

for (int j = 0; j < 40; j++) {

// compute the number of live neighbors of cell (i, j)

int liveNeighbors = 0;

for (int x = i - 1; x <= i + 1; x++) {

for (int y = j - 1; y <= j + 1; y++) {

if (x >= 0 && x < 40 && y >= 0 && y < 40 && !(x == i && y == j)) {

if (gameState[x][y]) {

liveNeighbors++;

}

}

}

}

// apply the rules of the game

if (gameState[i][j]) {

if (liveNeighbors < 2 || liveNeighbors > 3) {

newState[i][j] = false;

} else {

newState[i][j] = true;

}

} else {

if (liveNeighbors == 3) {

newState[i][j] = true;

} else {

newState[i][j] = false;

}

}

}

}

// update the game state

gameState = newState;

}

// 3: force an application repaint

// initialise double-buffering

createBufferStrategy(2);

strategy = getBufferStrategy();

// create a graphics object

Graphics g = strategy.getDrawGraphics();

// clear the buffer

g.setColor(Color.black);

g.fillRect(0, 0, getWidth(), getHeight());

// draw the game state

g.setColor(Color.white);

for (int i = 0; i < 40; i++) {

for (int j = 0; j < 40; j++) {

if (gameState[i][j]) {

g.fillRect(i \* 20, j \* 20, 20, 20);

}

}

}

//draw 4 buttons required

drawButtons(g);

// dispose the graphics object

g.dispose();

// show the buffer

strategy.show();

}

}

// mouse events which must be implemented for MouseListener

//mousePressed detects when the mouse is clicked and held

public void mousePressed(MouseEvent e) {

int x = e.getX();

int y = e.getY();

for (int i = 0; i < buttons.length; i++) {

if (buttons[i].contains(x, y)) {

switch (i) {

case 0:

System.out.println("Mouse pressed Start");

playing = !playing; // toggle between playing and not playing

break;

case 1:

System.out.println("Mouse pressed Random");

// Initialise the game state randomly

for (int i1 = 0; i1 < 40; i1++) {

for (int j = 0; j < 40; j++) {

Random random = new Random();

gameState[i1][j] = random.nextBoolean();

}

}

break;

case 2:

System.out.println("Mouse pressed Load");

loadGame();

break;

case 3:

System.out.println("Mouse pressed Save");

try {

FileWriter writer = new FileWriter(new File("gameState.txt"));

for (int j = 0; j < 40; j++) {

for (int k = 0; k < 40; k++) {

writer.write(gameState[j][k] ? "1" : "0");

writer.write(" ");

}

//I put a space between each element as it was printed to the file to assist with loading the file

writer.write("\n");

}

writer.close();

} catch (IOException ex) {

ex.printStackTrace();

}

break;

}

return;

}

}

// Only toggle cells when the game is playing

if (playing) {

// determine which cell of the gameState array was clicked on

int gameStateX = x / 20;

int gameStateY = y / 20;

// toggle the state of the cell

gameState[gameStateX][gameStateY] = !gameState[gameStateX][gameStateY];

}

}

//event is triggered when the mouse button is released

public void mouseReleased(MouseEvent e) {

// determine which cell of the gameState array was clicked on

int x = e.getX()/20;

int y = e.getY()/20;

// clear the state of the cell

gameState[x][y] = false;

this.repaint();

}

//detects when mouse enters the program

public void mouseEntered(MouseEvent e) {

System.out.println("Mouse entered the program");

}

//detects when mouse exits the program

public void mouseExited(MouseEvent e) {

System.out.println("Mouse exited the program");

}

//mouse clicked detects when the mouse is clicked normally

public void mouseClicked(MouseEvent e) {

// determine which cell of the gameState array was clicked on

int x = e.getX()/20;

int y = e.getY()/20;

// select the state of the cell

gameState[x][y] = true;

this.repaint();

}

//

// application's paint method

public void paint(Graphics g) {

super.paint(g);;

g = offscreenBuffer; // draw to offscreen buffer

// clear the canvas with a big black rectangle

g.setColor(Color.BLACK);

g.fillRect(0, 0, 800, 800);

g.setColor(Color.GREEN);

for (Rectangle button : buttons) {

g.fillRect(button.x, button.y, button.width, button.height);

}

// add labels to the buttons

g.setColor(Color.WHITE);

g.drawString("Start", buttons[0].x + 30, buttons[0].y + 30);

g.drawString("Random", buttons[1].x + 30, buttons[1].y + 30);

g.drawString("Load", buttons[2].x + 30, buttons[2].y + 30);

g.drawString("Save", buttons[3].x + 30, buttons[3].y + 30);

// redraw all game objects

g.setColor(Color.WHITE);

for (int x=0;x<40;x++) {

for (int y=0;y<40;y++) {

if (gameState[x][y]) {

g.fillRect(x\*20, y\*20, 20, 20);

}

}

}

// flip the buffers

strategy.show();

}

public void drawButtons(Graphics g) {

// set the font and color for the buttons

g.setFont(new Font("Arial", Font.BOLD, 14));

g.setColor(Color.GREEN);

// draw each button

for (int i = 0; i < buttons.length; i++) {

g.drawRect(buttons[i].x, buttons[i].y, buttons[i].width, buttons[i].height);

g.drawString(getButtonText(i), buttons[i].x + 10, buttons[i].y + 30);

}

}

public String getButtonText(int index) {

switch (index) {

case 0:

return "Start";

case 1:

return "Random";

case 2:

return "Load";

case 3:

return "Save";

default:

return "";

}

}

public void loadGame() {

try {

BufferedReader reader = new BufferedReader(new FileReader("gameState.txt"));

boolean[][] newGameState = new boolean[40][40];

String line;

int row = 0;

while ((line = reader.readLine()) != null && row < 40) {

String[] values = line.trim().split("\\s+");

for (int col = 0; col < 40 && col < values.length; col++) {

newGameState[row][col] = (values[col].equals("1"));

}

row++;

}

reader.close();

//put loaded values from the newGameState array into the gameState array to override the contents of the screen

gameState = newGameState;

} catch (IOException ex) {

ex.printStackTrace();

}

}

// application entry point

public static void main(String[] args) {

ConwaysLife w = new ConwaysLife();

}

}