

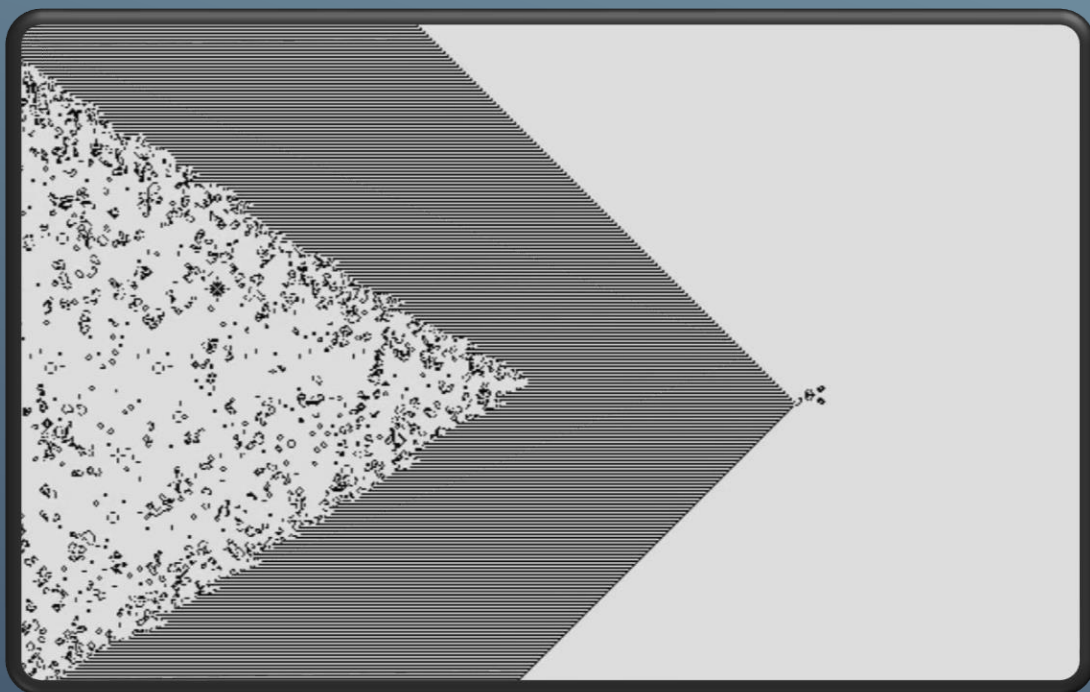
**Prepared by:**

Ehab Mohammed Qaid  
Al Ademi (202170330)

**Supervised by:**

**Prof:** Marwan Noman

**Eng:** Jihad Al-Zeyadi



CONWAY'S GAME OF LIFE

## **ABSTRACT**

This project was aimed at creating an application which showcased the famed Conway's Game of Life. This was decided since The Zero-Player game had iterations of system analysis concepts and proved to be a vital example of essential system specifications. Those specifications were reversibility, totality, and unpredictability. There was a need to showcase the types of GOL system types as well. Some were either in steady state, oscillating, dead or entirely random. The way to show this to the user was to allow them to enter whichever 2 dimensional 64x64 shape and to see the Game of Life run itself. An additional feature was added to allow the user to see a random input play itself.

# Table of Contents

<b>Chapter 1 Background.....</b>	<b>4</b>
<b>Chapter 2 Introduction.....</b>	<b>4</b>
<b>Chapter 3 Objective .....</b>	<b>8</b>
<b>Chapter 4 Methodology .....</b>	<b>8</b>
<b>4.1 FindNeighbor .....</b>	<b>9</b>
<b>4.2 Maker.....</b>	<b>12</b>
<b>4.3 Random Button.....</b>	<b>15</b>
<b>4.4 Custom Button &amp; Run Button .....</b>	<b>17</b>
<b>Chapter 5 Discussion.....</b>	<b>20</b>
<b>Chapter 6 Conclusion.....</b>	<b>21</b>
<b>Chapter 7 References .....</b>	<b>Error! Bookmark not defined.</b>

# List of Figures and Tables

Figure 1 The Hammerhead	Figure 2 The Cloverleaf	5
Figure 3 The "World Generator"		5
Figure 4 Stable System 1		6
Figure 5 Stable System 2		6
Figure 6 Oscillating System		7
Figure 7 Random Structure		7
Figure 8 Dying Structures		8
Figure 9 GUI look		9
Figure 10 initializing the world		10
Figure 11 Populating the 2D Grid		10
Figure 12 Beginning of While Loop		11
Figure 13 Continuation of the While Loop		12
Figure 14 maker.m		13
Figure 15 The Blank Custom Graph		14
Figure 16 User Input Initial State		15
Figure 17 Random Callback		15
Figure 18 Randomize 1		16
Figure 19 Randomize 2		16
Figure 20 Randomize 3 (Once it Stabilizes)		17
Figure 21 Custom Callback		18
Figure 22 Blank Structure		18
Figure 23 User Input		19
Figure 24 Run Callback		19
Figure 25 Running the User-generated Input		20

## Chapter 1 Background

The Game of Life is a cellular automaton zero-player game created by mathematician John Horton Conway in 1970. Game of Life, also known simply as Life, proved to be an important demonstration in the theory of cellular automata, and in computer science in general. The epigenetic theory of Conway's game was a further expansion of John Von Neumann's work on cellular. It gets its namesake from Conway himself. He experimented with numerous configurations on a two-dimensional grid and eventually came up with a set of guidelines he called the Game of Life that struck a good balance between extinction and endless cellular proliferation.

Conway's Game of Life is a two-dimensional grid of cell-based cellular automata. One of two states—ON or OFF—applies to each cell. It merely need the initial state as an input. At each interval, it refreshes and modifies the output in accordance with the guidelines below:

1. Every living cell dies if it has only one other living neighbor.
2. Any active cell that has at least two more active neighbors also moves on to the next group.
3. Any cell that has three or more neighbors who are still alive dies.
4. In the next set, any dead cell with precisely three alive neighbors comes to life.

## Chapter 2 Introduction

Game of Life's implementation throughout the world of Computer Science and Signal study is vital. Its importance extends into other sciences such as biology and physics. It brings out a distinct analogy on almost all ecological or digital systems. In essence, it is a system that only relies on the initial state as input, in which even the simplest inputs can create the most expansive and complex results. This can be seen in the seemingly random and yet organized and tessellating patterns that emerge as seen in **Figures 1,2 and 3**.

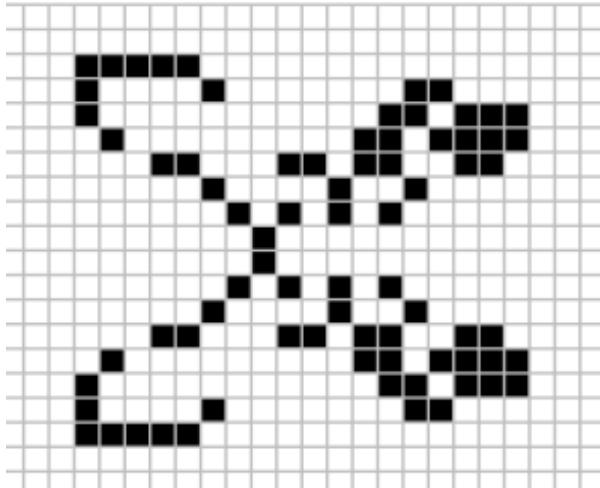


Figure 1 The Hammerhead

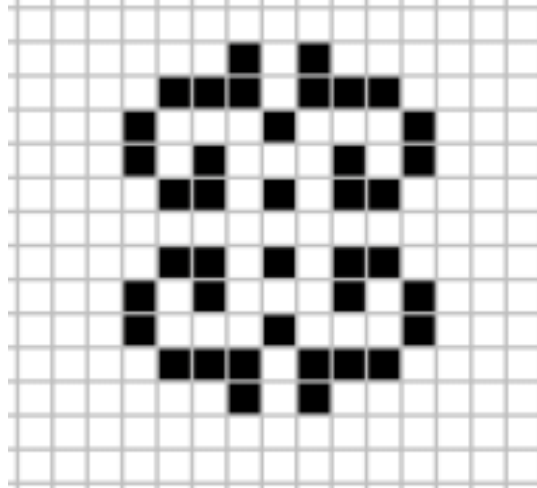


Figure 2 The Cloverleaf

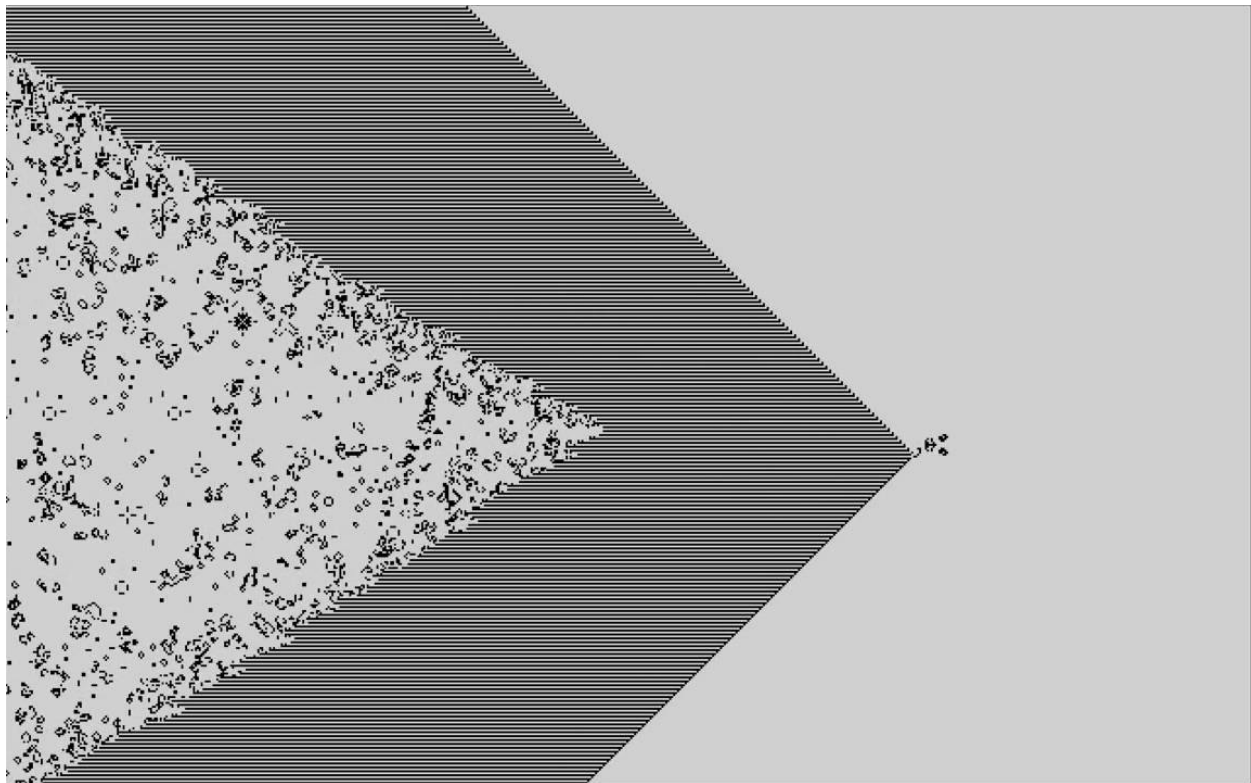


Figure 3 The "World Generator"

The Game of Life is a **reversible, totalistic unpredictable** automaton. The reasons for that is specified below:

- It is **Reversible** because in it, for every current configuration of the cellular automaton, there is exactly one past configuration.
- It is **Totalistic** because the value of a cell at time  $t$  depends only on the sum of the values of the cells in its vicinity at time  $t-1$ . Each cell in a totalistic cellular automaton is represented by a number, often an integer value selected from a finite set. The cellular automaton is correctly referred to as outer totalistic if the state of the cell at time  $t$  depends on both its own state and the sum of its neighbors' states at time  $t-1$ .
- It is **Unpredictable** because there is no algorithm that can predict if a later pattern will ever arise given an initial pattern and a later pattern.

The Game of Life can end in any sort of way. Certain patterns can point to differing states. They can divide into systems or sub-systems that include one of these four:

1. All initial patterns transform into a homogeneous, stable state. The initial pattern loses all unpredictability as seen in **Figures 4 and 5**.

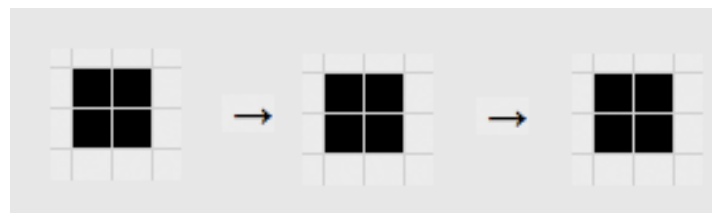


Figure 4 Stable System 1

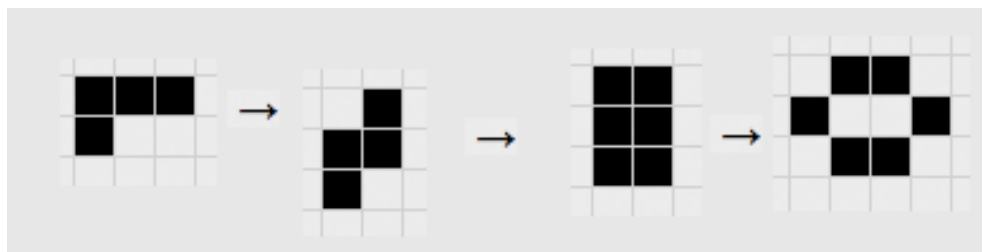
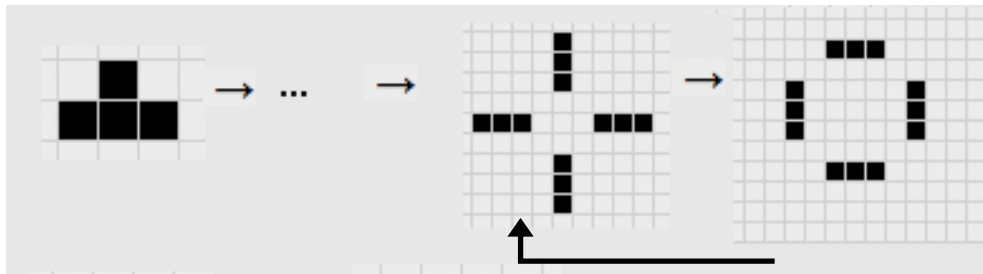


Figure 5 Stable System 2

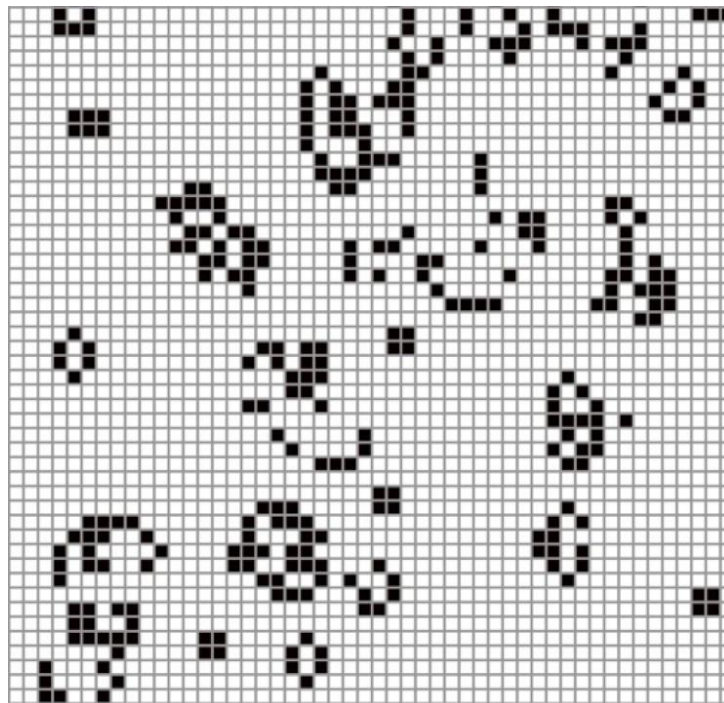
2. All basic patterns quickly transform into oscillating or stable structures. The initial pattern's unpredictability may filter out some of it, but some of it still

remains. Local modifications to the initial pattern typically stay local as seen in **Figure 6**.



*Figure 6 Oscillating System*

3. Every starting pattern develops in a chaotic or pseudo-random way. The ambient noise swiftly dismantles any sturdy structures that emerge. Local modifications to the basic pattern frequently continue to spread forever as seen in **Figure 7**.



*Figure 7 Random Structure*

4. All initial patterns develop into structures that interact in intricate and fascinating ways, resulting in the production of short-lived local structures as seen in **Figure 8**.



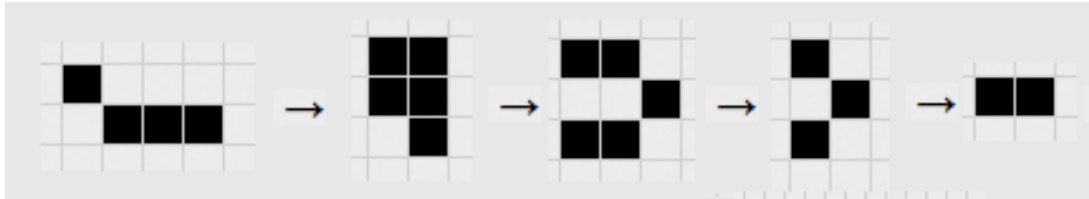


Figure 8 Dying Structures

## Chapter 3 Objective

Going forward with this project, several goals needed to be considered. One of the most self-evident was to provide a succinct application for concepts and ideas related to Signals & Systems. These concepts can be summarized in:

- Underline the unique characteristics of Conway's Game of life
- Showcasing a comprehensive application of Signals and Systems study.
- Utilize Matlab code and GUI to create a viable Game of Life application.

## Chapter 4 Methodology

For this project, Matlab R2018 was used since the requirements of this project suit its implementation quite handily. It was decided to implement this project as a GUI application which creates the 2 by 2 grid in the form of a plot figure. First we create the general GUI Which we will be using in this project.

Creating the general GUI, this simple outline was designed:

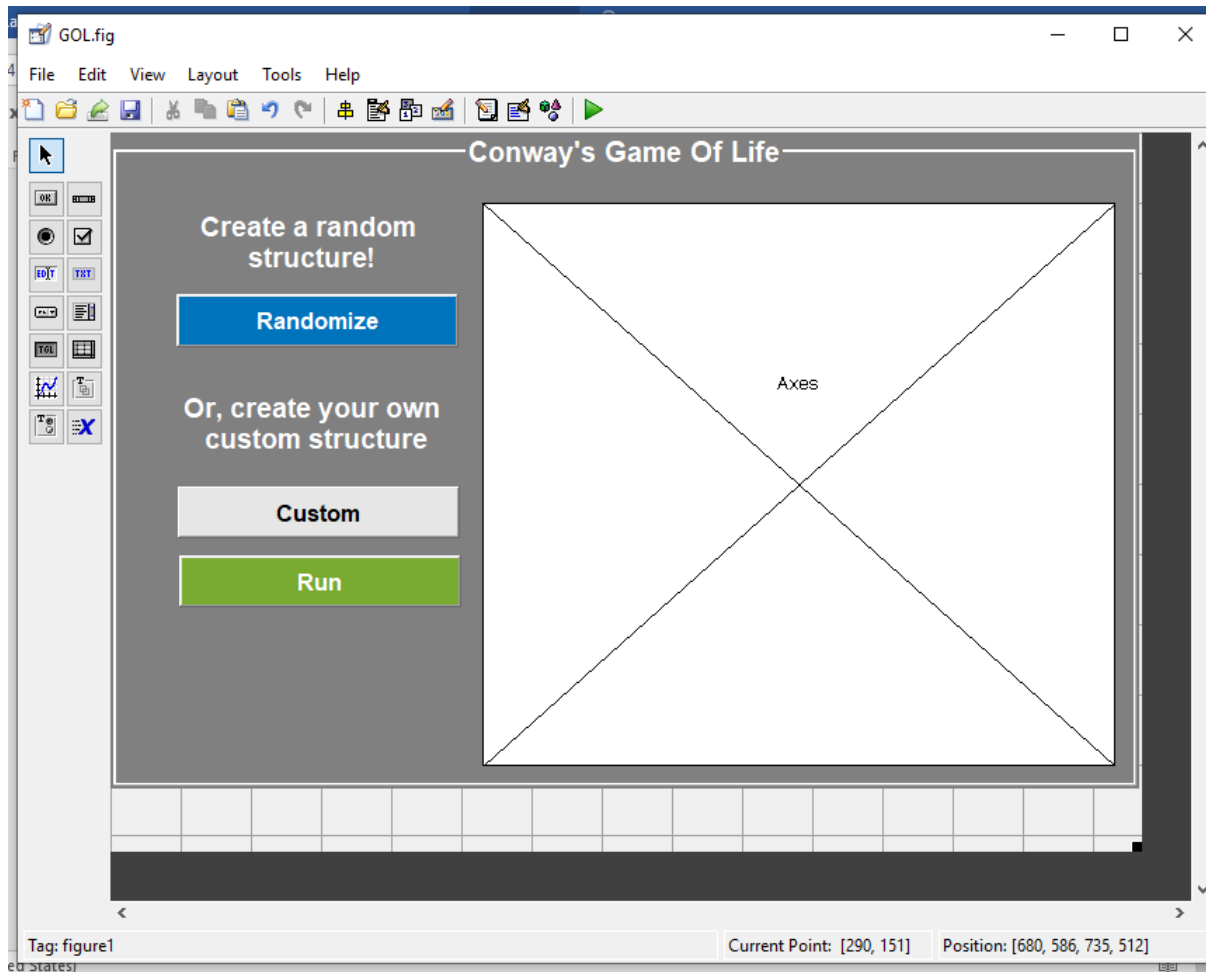


Figure 9 GUI look

Before we could work on the callbacks of each button, we will need to create a matlab function to run the iterations of Conway's Game of Life.

We will also need to create a function to allow us to make custom structures.

## 4.1 FindNeighbor

FindNeighbor.m has the purpose of executing the game itself. This is done since it takes a 64x64 array of 0s and 1s. the exact composition of those zeros and ones is determined by a variable called "random".

```

1      % Game of life
2 -    tic
3 -    global starting_world random
4 -    domain_size = [64 64]; % row,column
5
6      % draw in real time?
7
8 -    draw_world = 1;
9
10     % number of world replicates
11 -    n_world_replica = 1;
12
13 -    gen=zeros(n_world_replica,1);
14
15     % Prepare figures
16     %f1 = figure('Position',[10 150 500 500]);
17     %f2 = figure('Position',[800 250 500 500]);

```

Figure 10 initializing the world

The “tic” is met with a corresponding “toc” at the end of the block which calculates the estimated time needed for the program to run. The draw\_world and n\_world replica variables are to make sure that the world is drawn in real time and to manage the quantity of replicates respectively. Gen is simply nothing more than the initial all zero array of the starting world.

Next, we populate the world with 1s and 0s. as seen in **Figure 11**

```

13 -    gen=zeros(n_world_replica,1);
14
15 -    for(n=1:n_world_replica)
16 -        % Initial status
17 -        if random == 1
18 -            world = randi([0 1], domain_size(2), domain_size(1));
19 -        else
20 -            world = starting_world;
21 -        end
22 -        old_world = world;
23
24         % clear borders
25 -        world(:,1) = 0;
26 -        world(1,:) = 0;
27 -        world(:,end) = 0;
28 -        world(end,:) = 0;
29
30 -        if draw_world==1
31 -            %figure(f1);
32 -            pcolor(world);
33 -            colormap gray
34 -            axis square
35 -            title(sprintf('Replica %f of %f', n, n_world_replica))
36 -        end
37

```

Figure 11 Populating the 2D Grid

This part interacts with our global variable “random”. If ‘random’ is set at a value of 1, then the world grid will be filled with a random 1 and 0 array, if else, It will be given the value of ‘starting\_world’ specified in another function.

We add the variable ‘old\_world’ to compare the code before and after the change. This is to create an exit point once the the new state is equal to the old state. We then make sure the edges stay at 0 and decide the style of our graph.

We now enter our main infinite loop. With a while loop as seen in **Figure 12**

```

38 % main cycle
39 gen(n)=1;
40 while(1)
41
42
43     new_world = world;
44     if ~mod(gen,2) % even generation
45         if old_world == new_world
46             break
47         end
48         old_world = world;
49     end
50     for row=2:size(world, 1)-1
51         for column=2:size(world, 2)-1
52             if world(row,column)==1 % LIVE CELL
53                 %neighborhood status
54                 neighborhood = sum(sum(world(row-1:row+1,column-1:column+1)))-1;
55                 if neighborhood < 2 % death by underpopulation
56                     new_world(row,column)=0;
57                 end
58                 if neighborhood >= 2 && neighborhood <= 3 % survive
59                     new_world(row,column)=1;
60                 end
61                 if neighborhood > 3 % death by overpopulation
62                     new_world(row,column)=0;
63                 end

```

Figure 12 Beginning of While Loop

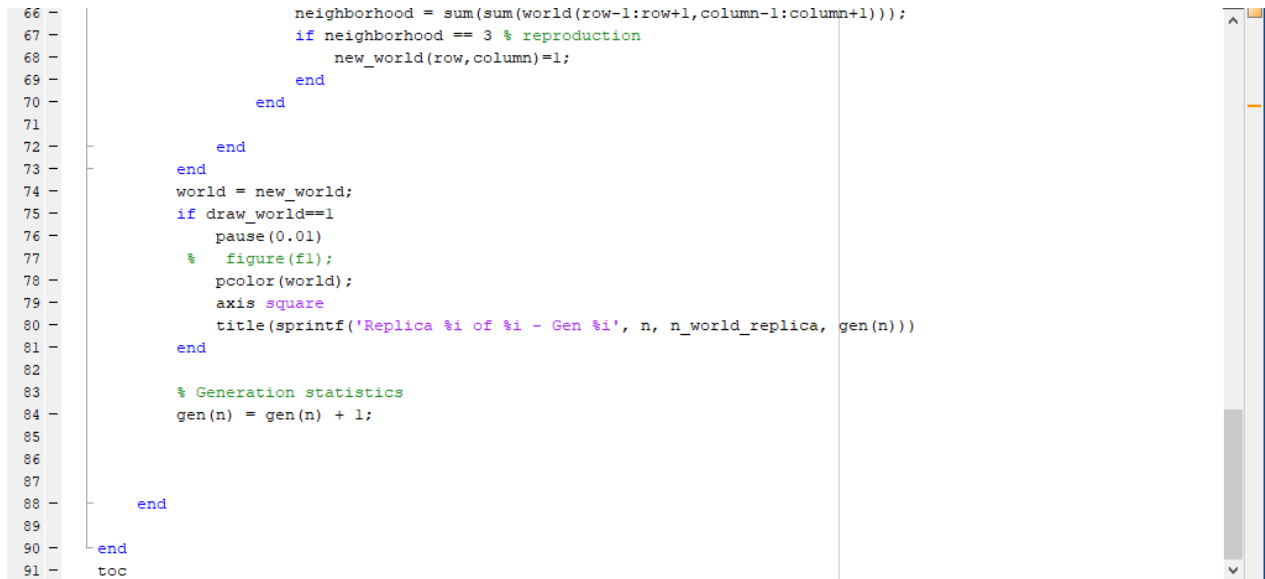
We first have to specify the redundant states in which the break action must be executed. Namely, those are the stable and oscillating systems seen in **Figures 5** and **6**.

We then specify the conditions for the game specified earlier in the report:

1. Every living cell dies if it has only one other living neighbor.
2. Any active cell that has at least two more active neighbors also moves on to the next group.
3. Any cell that has three or more neighbors who are still alive dies.
4. In the next set, any dead cell with precisely three alive neighbors comes to life.

The way that this is done is by creating an 3x3 array around our cell called “neighborhood”. Here, “neighborhood” is the sum of all the array’s parts minus the middle cell itself. We utilize the “drawworld” variable here to dictate the speed of the process. This all transpires as gen(n) is incremented at each loop.

These operations can be seen in **Figure 13**.



```

66 - neighborhood = sum(sum(world(row-1:row+1,column-1:column+1)));
67 - if neighborhood == 3 % reproduction
68 -     new_world(row,column)=1;
69 - end
70 - end
71 -
72 - end
73 - end
74 - world = new_world;
75 - if draw_world==1
76 -     pause(0.01)
77 -     % figure(f1);
78 -     pcolor(world);
79 -     axis square
80 -     title(sprintf('Replica %i of %i - Gen %i', n, n_world_replica, gen(n)))
81 - end
82 -
83 - % Generation statistics
84 - gen(n) = gen(n) + 1;
85 -
86 -
87 -
88 - end
89 -
90 - end
91 - toc

```

Figure 13 Continuation of the While Loop

As seen in the figure above, this marks the end of the main cycle, and the aforementioned “toc” at the bottom to estimate the time taken to run the program

## 4.2 Maker

The role of maker.m is to create a grid according to the user’s choice. The way to do that is by allowing the user to click on any box to turn it on. If the user clicks on it again, the box will turn off..

The function “ginput(x,y)” was used here to allow us to pick and choose points using the mouse cursor. In **Figure 14** you will see that the array value [x,y] is given a value using the “ginput function”. If [x,y] is equal to zero, then it will equal one and vice versa.

```

1  % Click outside of the matrix to complete
2  global starting_world
3  starting_world = zeros(64, 64);
4  starting_world(starting_world==0) = 1;
5
6  %figure(fl);
7  pcolor(starting_world);
8  colormap gray
9  while (1)
10     [x,y] = ginput(1);
11     if (x>size(starting_world,1) || y>size(starting_world,1) || (x<0 || y<0))%condition for clicking out of the matrix
12         break;
13     end
14     if starting_world(floor(y),floor(x)) == 0
15         starting_world(floor(y),floor(x)) = 1;
16     else
17         starting_world(floor(y),floor(x)) = 0;
18     end
19     pcolor(starting_world);
20 end
21 starting_world(starting_world==1) = 2;
22 starting_world(starting_world==0) = 1;
23 starting_world(starting_world==2) = 0;
24 clearvars -except starting_world ;

```

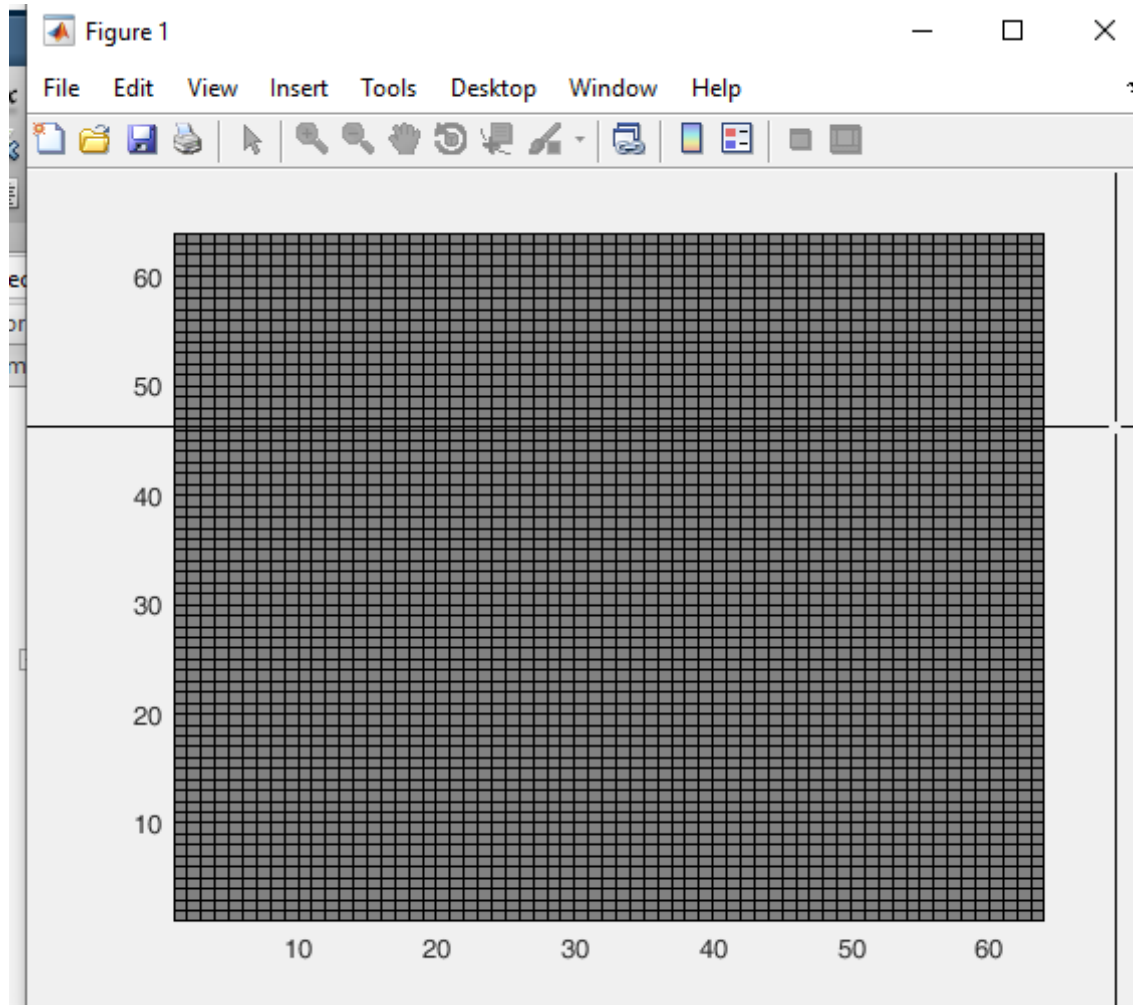
Figure 14 maker.m

Notice that “starting\_world” is a global variable since it is accessed in other buttons and files in the program. Similar to “gen” from the previous section, “starting\_world” is initialized with an all-zeros array.

The “ginput()” variable gives the box it clicks on a 1 value. This applies in a condition that if the variable has a 0 value. Otherwise, the box will always be zero. This means that if the user clicks an ON box, they will be able to turn it off.

There was a need for a condition which allowed us to break out of the while loop. Here, the “starting\_world” array is 64x64. If the user clicked outside of the array (meaning any column or row higher than 64, or any column or row lower than 0) then the loop is broken. This is seen in **Figure 14** line 11.

We then initialize a figure which looks like the following.



*Figure 15 The Blank Custom Graph*

Notice the mouse cursor pointer that pinpoints where the user clicks. After assigning a pattern by the user as seen in **Figure 17**:

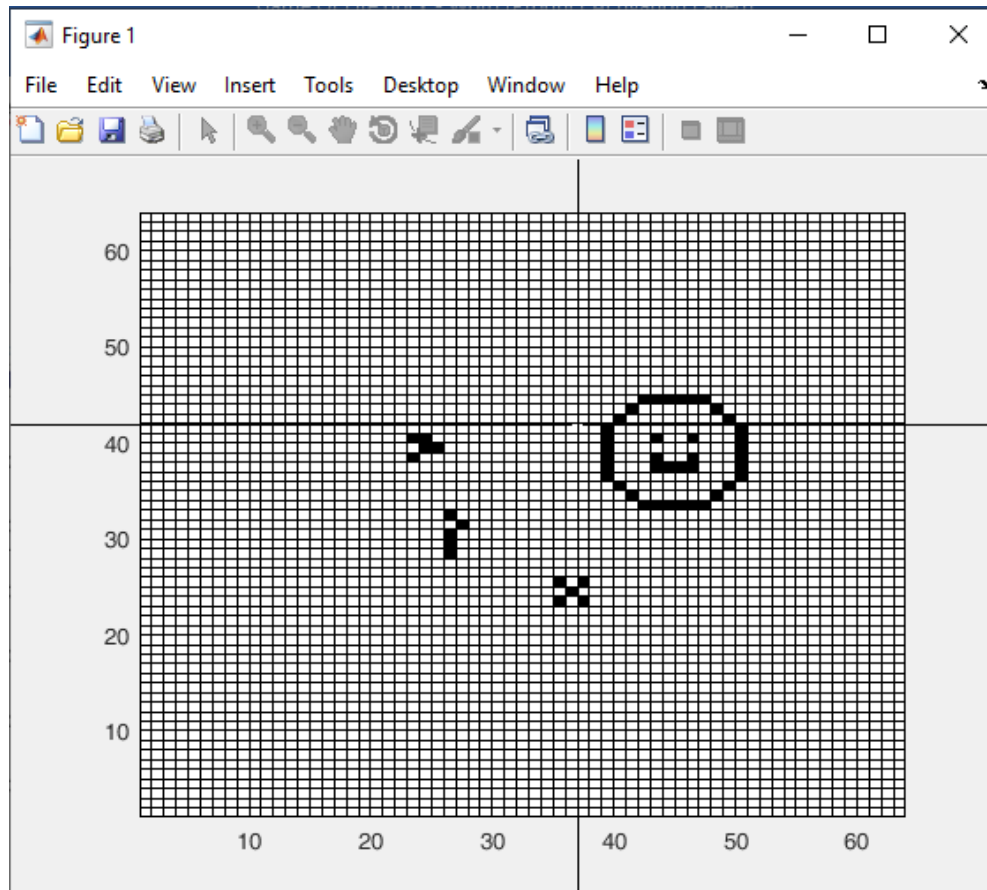


Figure 16 User Input Initial State

### 4.3 Random Button

The Random button is different from the others since it does not need another button to run. The user can click it and start the process on its own.

Here, we will access the aforementioned FindNeighbor.m and give the variable “Random” a value of 1 as seen in **Figure 17**.

```

76 % --- Executes on button press in Random.
77 function Random_Callback(hObject, eventdata, handles)
78 % hObject      handle to Random (see GCBO)
79 % eventdata    reserved - to be defined in a future version of MATLAB
80 % handles      structure with handles and user data (see GUIDATA)
81 set(handles.pause,'value',0)
82 global random
83 axes(handles.Axes);
84 random = 1; % if set to one load a random matrix
85 findNeighbor;

```

Figure 17 Random Callback

This is in order to give it the random array specified in **Figure 11** lines 17 and 18.



Clicking on it results in the random array shown in **Figures 18, 19, and 20**.

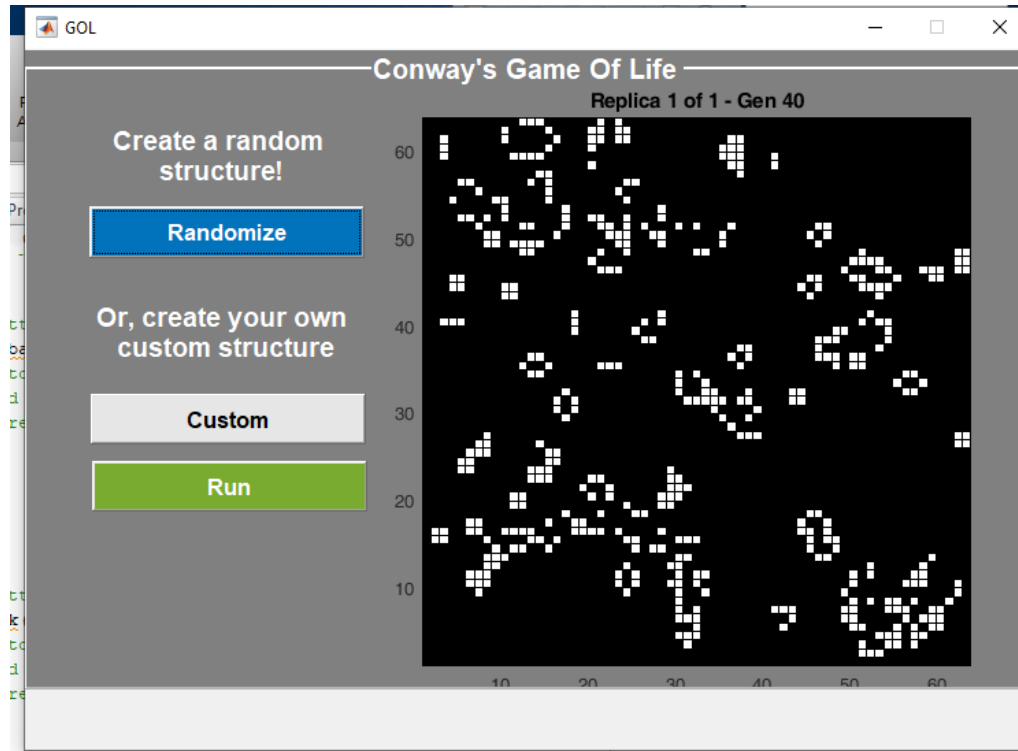


Figure 18 Randomize 1

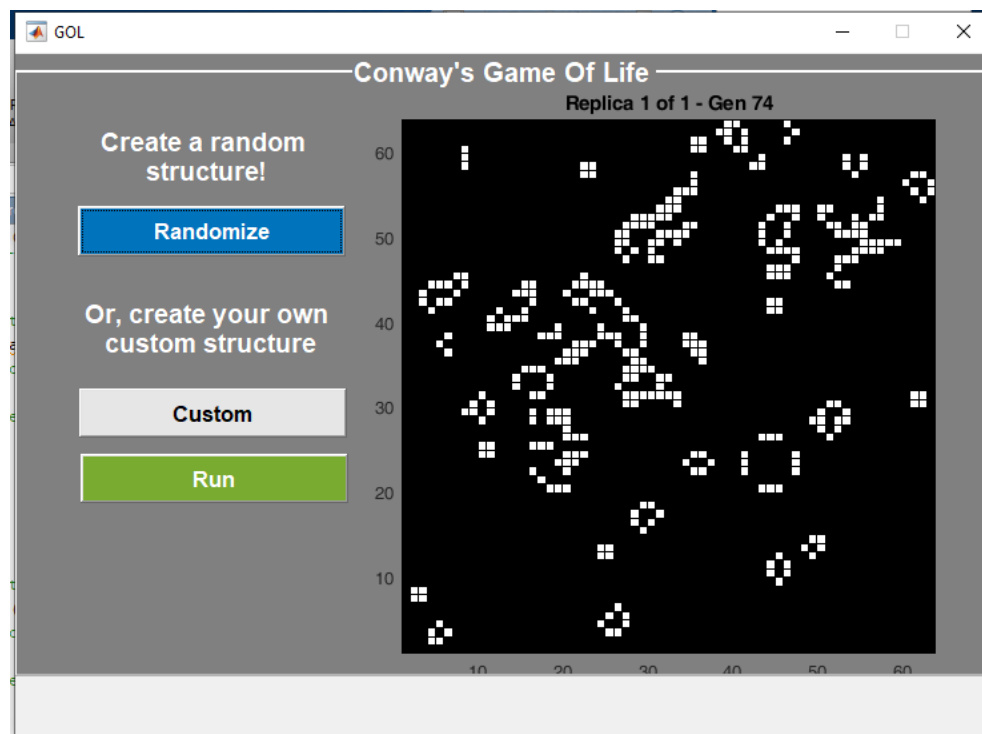


Figure 19 Randomize 2

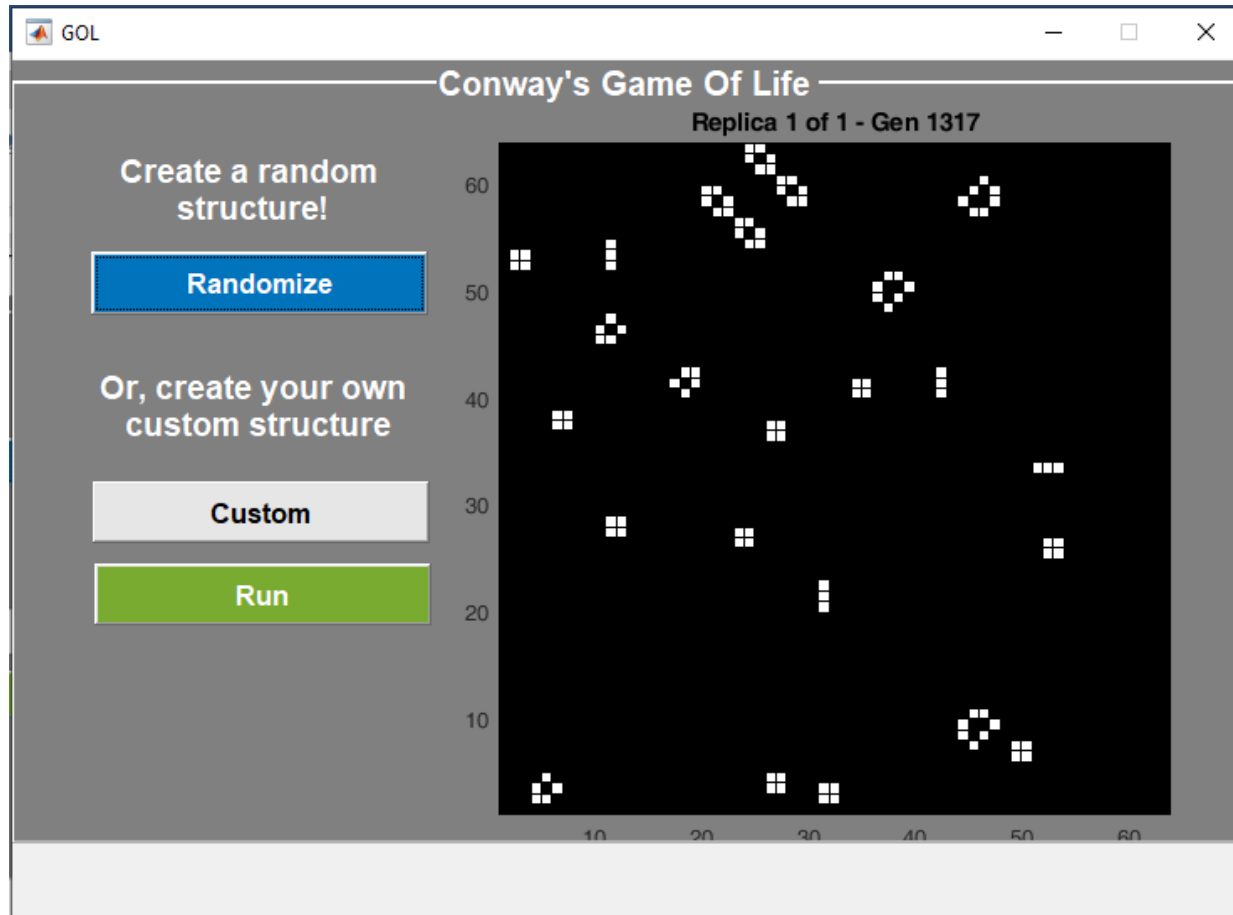


Figure 20 Randomize 3 (Once it Stabilizes)

As seen in **Figure 20**, this random input stabilized after 1317 loops (or generations) all the shapes shown above are either steady state or oscillating. Not all inputs stabilize however, as some can continue in random behavior forever.

#### 4.4 Custom Button & Run Button

The Custom button accesses the maker.m we worked on earlier. As seen in **Figure 21**

```
93 % --- Executes on button press in Custom.
94 function Custom_Callback(hObject, eventdata, handles)
95 % hObject    handle to Custom (see GCBO)
96 % eventdata  reserved - to be defined in a future version of MATLAB
97 % handles    structure with handles and user data (see GUIDATA)
98 - axes(handles.Axes);
99 - maker;
100 |
```

Figure 21 Custom Callback

That gives us a blank graph ready to be controlled by the user. As seen in **Figure 22**.

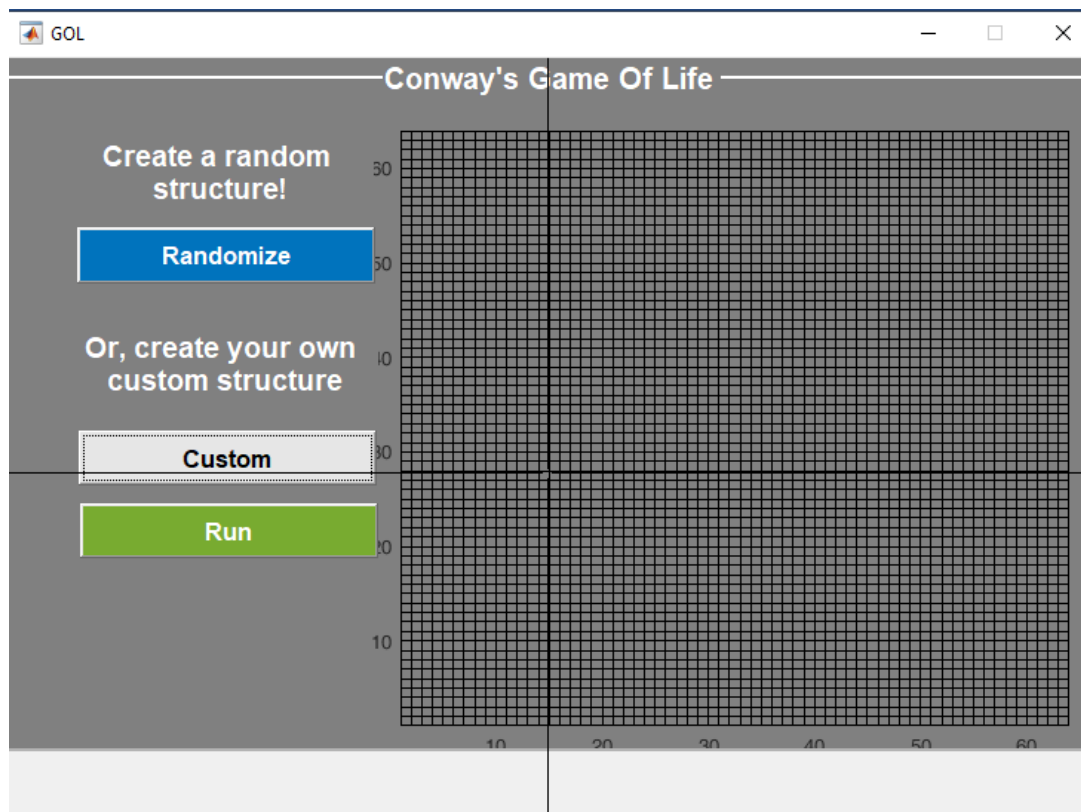


Figure 22 Blank Structure

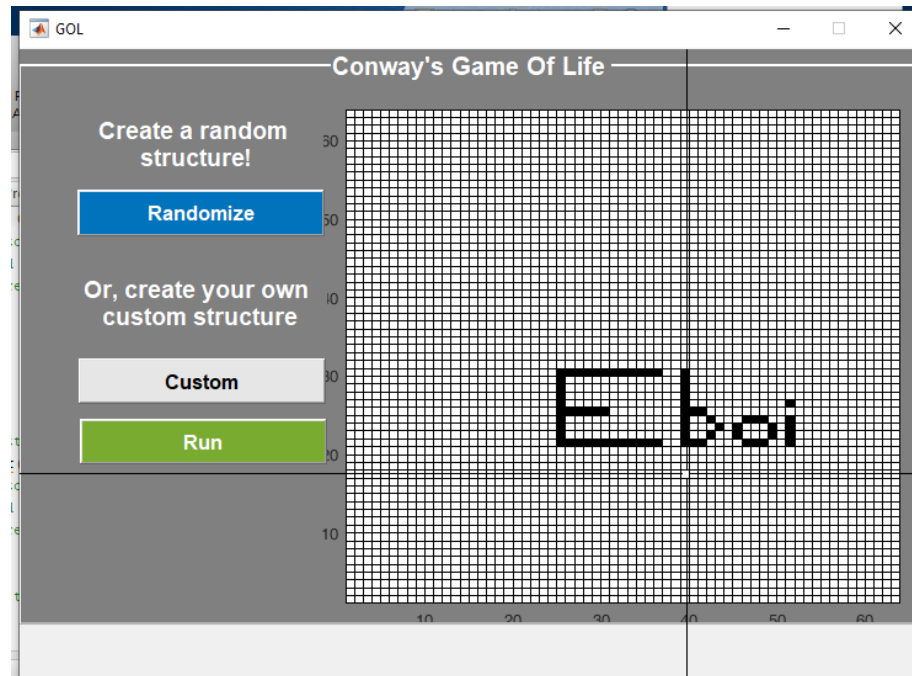


Figure 23 User Input

The Run Callback is similar to the Randomize Callback with the only exception being the initial value.

```

103
104 % --- Executes on button press in Run.
105 function Run_Callback(hObject, eventdata, handles)
106 % hObject    handle to Run (see GCBO)
107 % eventdata  reserved - to be defined in a future version of MATLAB
108 % handles    structure with handles and user data (see GUIDATA)
109 global random
110 axes(handles.Axes);
111 random = 0; % if set to zero load a 2d matrix named "starting_world"
112 findNeighbor;
113
114

```

Figure 24 Run Callback

Since random is given a value other than 1, that will give us access to “starting\_world” as specified in **Figure 11** lines 19 and 20. Running the input shown in **Figure 23**, we get:

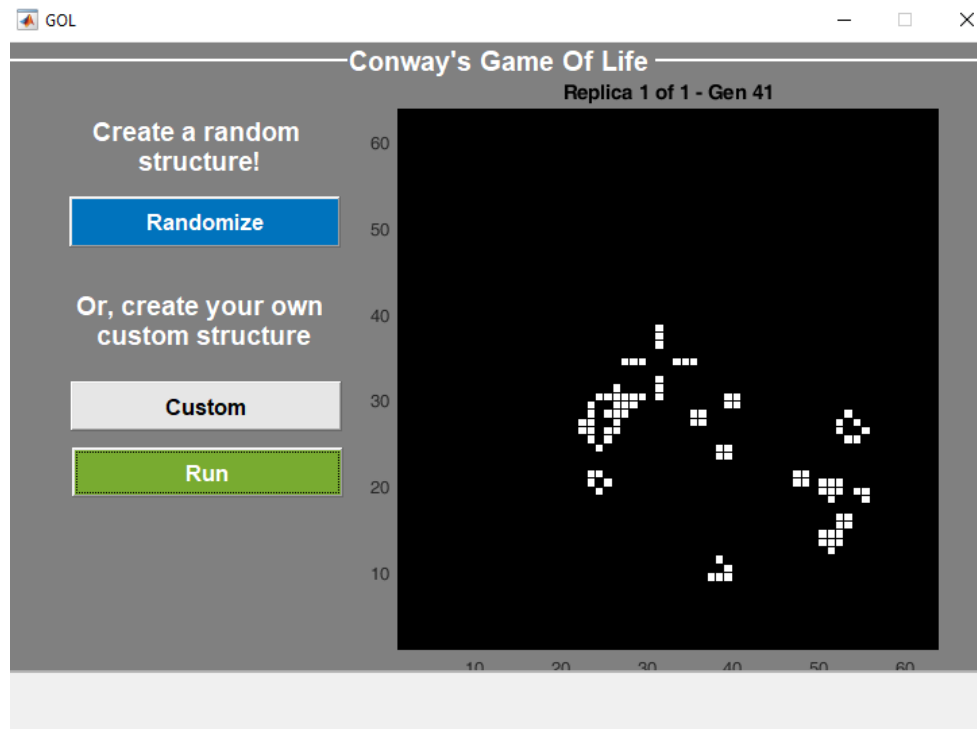


Figure 25 Running the User-generated Input

## Chapter 5 Discussion

The creation of this project posited some difficulties in terms of coding. One of the most evident is creating the conditions for the next state. Another one was finding a suitable function that allowed the user to input an initial state themselves. This gave way for new functions to be explored such as “ginput”, “tic” and “toc” and many other useful applications in Matlab. The Matlab GUI served as an effective tool in simplifying the the blocks on code into a comprehensive User Interface accessible by anyone regardless of their comprehension skills in programming.

It is crucial that Conway's Game of Life be applied to the fields of computer science and signal analysis. Its significance extends to various fields of study, including physics, biology, meteorology, and other disciplines. It highlights a clear comparison for virtually all biological or digital systems. It is essentially a system that only accepts the starting state as an input and in which even the most straightforward inputs can result in the most extensive and convoluted outcomes. This is evident in the patterns that appear to be random but are actually structured and tessellating. The Game of Life can be played by setting up an initial configuration and then watching it develop. It can imitate a universal constructor or any other Turing machine since it is Turing complete.

## **Chapter 6      Conclusion**

The goals set out for this project, were neatly fulfilled as each facet has been completed. Conway's Game of Life was featured in all of its details, and principles and ideas linked to signals and systems were presented in a clear application. This project used Matlab code and a GUI to make a workable Game of Life application and extensively demonstrated how to apply a study of signals and system.