Queen's University

Artificial Life: Craig Reynolds' Boids System and Conway's Game of Life

CISC 352

Assignment 3

Professor Sidney Givigi

March 29, 2020

Group 16

Christina Hu	[20093642]
Jie Niu	[20072214]
Wansi Liang	[20067725]
Gary Wang	[20073161]
Derek Huang	[20022672]

Purpose

To simulate the Boids System by implementing the Craig Reynolds' Boids System with 3 different rules (Cohesion, Alignment, and Separation) in Artificial Life. To simulate the Life System by implementing the Conway's Game of Life System in the field of Artificial Life.

Introduction

<u>Craig Reynolds' Boids System</u>

The Craig Reynolds' Boids System is an interesting system. Simulating the Boids System requires a visual display.

This system models coordinated animal motion such as flocks of birds and schools of fish. Rather than having an overall mechanism for controlling the flock, the system has just a few rules that each individual boid obeys. The basic flocking model consists of three simple steering behaviors which describe how an individual boid maneuvers based on the positions and velocities of its nearby flockmates.

There are 3 rules associated with the Boids System:

Rule 1 (Cohesion): Boids tend to fly towards the center of mass of neighboring boids.
Rule 2 (Separation): Boids tend to keep a small distance away from other objects and boids.
Rule 3 (Alignment): Boids tend to match velocity with nearby boids.

<u>Conway's Game of Life</u>

The Game of Life (an example of a cellular automaton) is played on an infinite two-dimensional rectangular grid of cells. Each cell can be either alive or dead. The status of each cell changes each turn of the game (also called a generation) depending on the statuses of that cell's 8 neighbors. Neighbors of a cell are cells that touch that cell, either horizontal, vertical, or diagonal from that cell.

The initial pattern is the first generation. The second generation evolves from applying the rules simultaneously to every cell on the game board, i.e. births and deaths happen simultaneously. Afterwards, the rules are iteratively applied to create future generations. For each generation of the game, a cell's status in the next generation is determined by a set of rules. These simple rules are as follows:

If the cell is alive, then it stays alive if it has either 2 or 3 live neighbors
If the cell is dead, then it springs to life only in the case that it has 3 live neighbors

Methods and Processes

<u>Craig Reynolds' Boids System (Part1)</u>

Usage

To run the `boids.py` in python GUI interface: run

```
pythonw boids.py
```

The python script `boids.py` takes in no inputs where you may just run the program itself. First, we defined variables that are commonly used in the whole program. Second, we created an initializer of a 2D vector and some mathematical algorithms for 2D vector(s). Third, we created a class called Boid that includes an initializer of the velocity and position of boids and three rules. Fourth, we built a GUI interface that immediately shows how the boids move based on the boids system rules and these boids will move continuously until we interrupt or quit the program.

We describe the functions and algorithms used in the program as follows:

1. In the first part of our program, we import random and tkinter packages. Also, we defined the width of the screen, the height of the screen, the number of boids, the radius of boids, the speed limit of boids, the acceleration of boids,

```
from tkinter import *
import random

#Variables definitions
Gwidth = 1024
Gheight = 768
Gboids = 50
GboidRadius = 3
Gspeed = 800
Gacceleration = 15
Gside = 500
GfromSide = 50
```

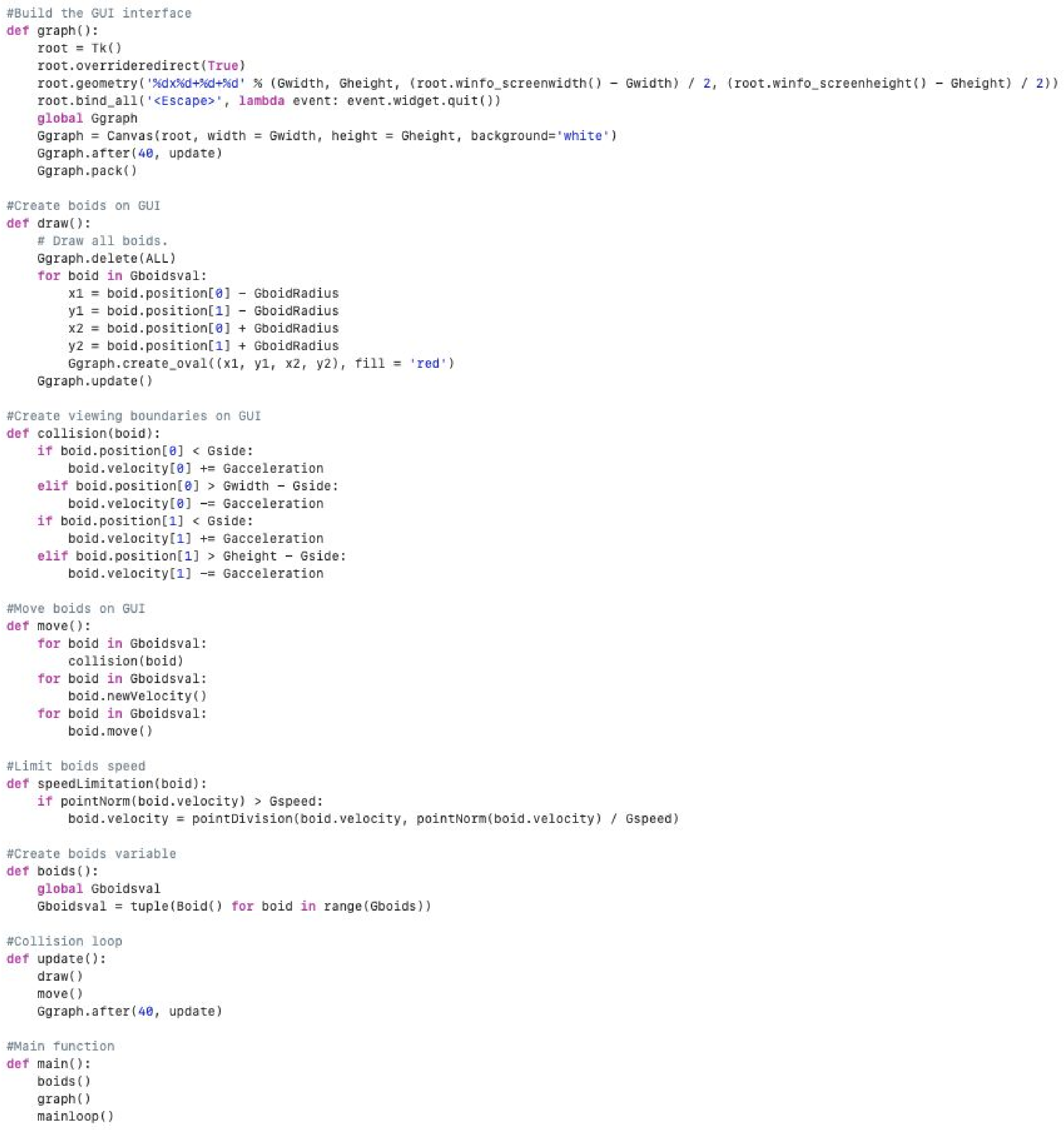
2. In the second part of our program, we created an initializer of a 2D vector and some mathematical algorithms for 2D vectors. Including, addition of two 2D vectors, subtraction of two 2D vectors, division of a 2D vector with a number, and the norm of a 2D vector. These algorithms return a 2D vector as a result, except pointNorm(input) which returns a number instead.

```
#Initialize a 2D vector
def twoDInitializer(input1, input2):
    return [float(input1), float(input2)]

#2D array addition
def pointAddition(input1, input2):
    return [float(input1[0] + input2[0]), float(input1[1] + input2[1])]

#2D array subtraction
def pointSubtraction(input1, input2):
    return [float(input1[0] - input2[0]), float(input1[1] - input2[1])]

#2D array division
def pointDivision(input1, input2):
    return [float(input1[0] / input2), float(input1[1] / input2)]

#Norm of a 2D array
def pointNorm(input):
    return (input[0] ** 2 + input[1] ** 2) ** 0.5
```

3. In the third part of our program, we created a class named Boid which initialized the velocity and position of boids, included three rules of the Craig Reynolds' Boids System -- cohesion, separation, alignment, and updated the velocity and position of boids.

```python
#Boid Initialization and rules
class Boid:

    #Initialize boid velocity and position
    def __init__(self):
        self.velocity = twoDInitializer(0, 0)
        self.position = twoDInitializer(*self.randNum())

    #Random boid position
    def randNum(self):
        if random.choice([True, False]):
            yCoordinate = random.randint(1, Gheight)
            if random.choice([True, False]):
                xCoordinate = -GfromSide
            else:
                xCoordinate = Gwidth + GfromSide
        else:
            xCoordinate = random.randint(1, Gwidth)
            if random.choice([True, False]):
                yCoordinate = -GfromSide
            else:
                yCoordinate = Gheight + GfromSide
        return xCoordinate, yCoordinate

    #Boid rule No.1 -- cohesion
    def rule1(self):
        vector = twoDInitializer(0, 0)
        for boid in Gboidsval:
            if boid is not self:
                vector = pointAddition(vector, boid.position)
        vector = pointDivision(vector, len(Gboidsval) - 1)
        return pointDivision(pointSubtraction(vector, self.position), 7.5)

    #Boid rule No.2 -- separation
    def rule2(self):
        vector = twoDInitializer(0, 0)
        for boid in Gboidsval:
            if boid is not self:
                if pointNorm(pointSubtraction(self.position, boid.position)) < 25:
                    vector = pointSubtraction(vector, pointSubtraction(boid.position, self.position))
        return vector

    #Boid rule No.3 -- alignment
    def rule3(self):
        vector = twoDInitializer(0, 0)
        for boid in Gboidsval:
            if boid is not self:
                vector = pointAddition(vector, boid.velocity)
        vector = pointDivision(vector, len(Gboidsval) - 1)
        return pointDivision(pointSubtraction(vector, self.velocity), 2)

    #Update boid velocity
    def newVelocity(self):
        vector1 = self.rule1()
        vector2 = self.rule2()
        vector12 = pointAddition(vector1, vector2)
        vector3 = self.rule2()
        self.temp = pointAddition(vector12, vector3)

    #Move boids to new position
    def move(self):
        self.velocity = pointAddition(self.velocity, self.temp)
        speedLimitation(self)
        self.position = pointAddition(self.position, pointDivision(self.velocity, 100))
```

4. In the fourth part of our program, we created a GUI interface, boids on the GUI, viewing boundaries of boids on the GUI, boids movement on the GUI, and the speed limitation of boids.



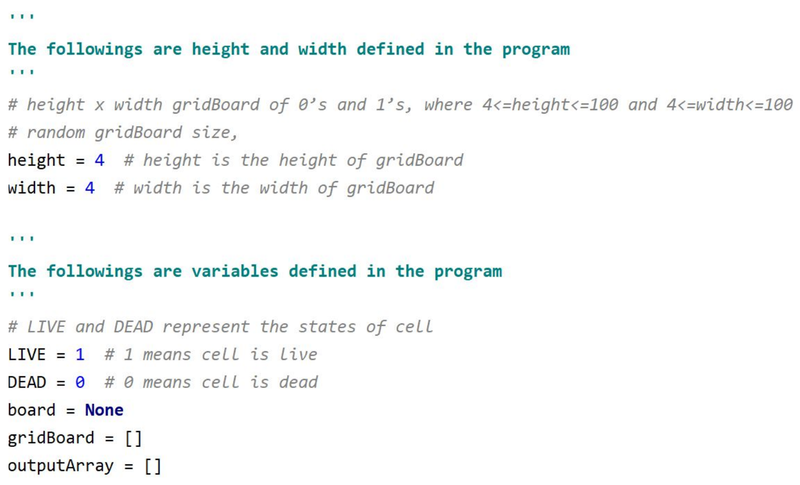
```python
#Build the GUI interface
def graph():
    root = Tk()
    root.overrideredirect(True)
    root.geometry('%dx%d+%d+%d' % (Gwidth, Gheight, (root.winfo_screenwidth() - Gwidth) / 2, (root.winfo_screenheight() - Gheight) / 2))
    root.bind_all('<Escape>', lambda event: event.widget.quit())
    global Ggraph
    Ggraph = Canvas(root, width = Gwidth, height = Gheight, background='white')
    Ggraph.after(40, update)
    Ggraph.pack()

#Create boids on GUI
def draw():
    # Draw all boids.
    Ggraph.delete(ALL)
    for boid in Gboidsval:
        x1 = boid.position[0] - GboidRadius
        y1 = boid.position[1] - GboidRadius
        x2 = boid.position[0] + GboidRadius
        y2 = boid.position[1] + GboidRadius
        Ggraph.create_oval((x1, y1, x2, y2), fill = 'red')
    Ggraph.update()

#Create viewing boundaries on GUI
def collision(boid):
    if boid.position[0] < Gside:
        boid.velocity[0] += Gacceleration
    elif boid.position[0] > Gwidth - Gside:
        boid.velocity[0] -= Gacceleration
    if boid.position[1] < Gside:
        boid.velocity[1] += Gacceleration
    elif boid.position[1] > Gheight - Gside:
        boid.velocity[1] -= Gacceleration

#Move boids on GUI
def move():
    for boid in Gboidsval:
        collision(boid)
    for boid in Gboidsval:
        boid.newVelocity()
    for boid in Gboidsval:
        boid.move()

#Limit boids speed
def speedLimitation(boid):
    if pointNorm(boid.velocity) > Gspeed:
        boid.velocity = pointDivision(boid.velocity, pointNorm(boid.velocity) / Gspeed)

#Create boids variable
def boids():
    global Gboidsval
    Gboidsval = tuple(Boid() for boid in range(Gboids))

#Collision loop
def update():
    draw()
    move()
    Ggraph.after(40, update)

#Main function
def main():
    boids()
    graph()
    mainloop()
```

<u>Conway's Game of Life (Part2)</u>

The python script `life.py` takes input from *"inLife.txt"* and output to "*outLife.txt*".

To run the Life program, make sure that *inLife.txt* file is in the same folder. This text file contains multiple lines. The input configuration is a m x n grid of 0's and 1's, where 4<=m<=100 and 4<=n<=100. The input will consist of an initial condition,, along with the number of generations to simulate. There will only be one input configuration per file.

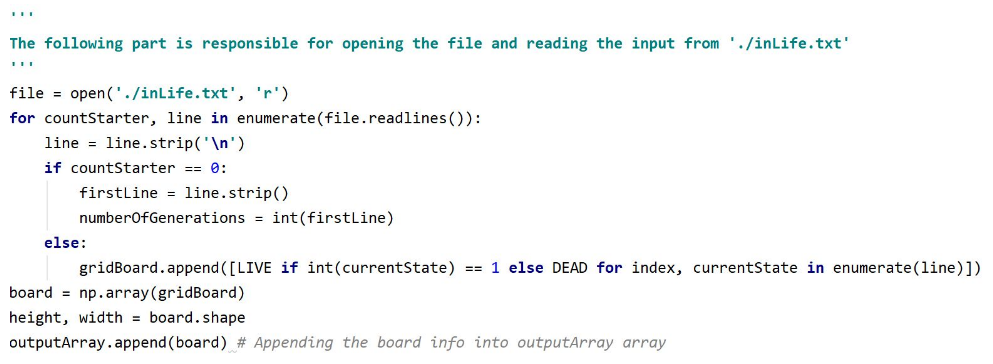
1. In the first part of our program, we defined the following *height* and *width* in the program to model an m x n grid, *board* and *gridBoard*. As well, the following are variables defined in the program in order to represent states of the cell where 1 means cell is alive and 0 means cell is dead. Moreover, the *outputArray* will be used to store the final output content.



```
'''
The followings are height and width defined in the program
'''
# height x width gridBoard of 0's and 1's, where 4<=height<=100 and 4<=width<=100
# random gridBoard size,
height = 4  # height is the height of gridBoard
width = 4  # width is the width of gridBoard

'''
The followings are variables defined in the program
'''
# LIVE and DEAD represent the states of cell
LIVE = 1  # 1 means cell is live
DEAD = 0  # 0 means cell is dead
board = None
gridBoard = []
outputArray = []
```

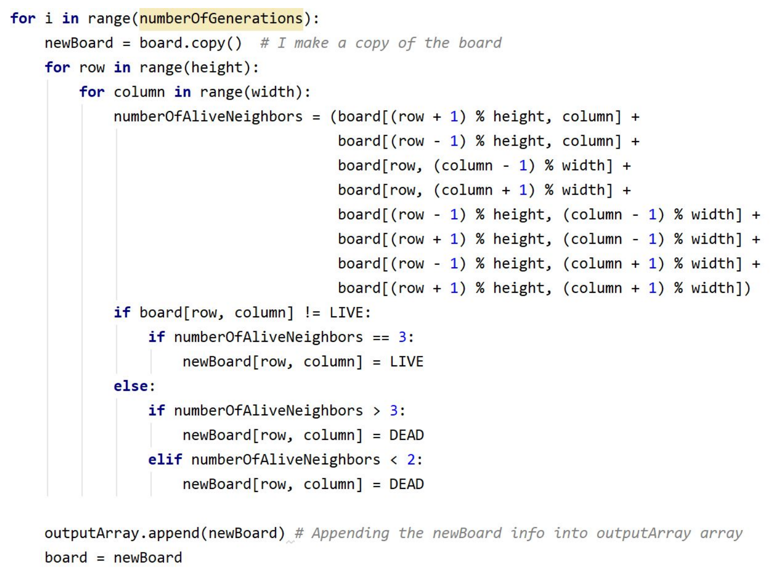
2. In the second part of our program, we defined the following part to be responsible for opening the file as well as reading the input from the '*inLife*.txt' file. This part should be fairly straightforward where it reads in input from the input file and then appends either *Live* (1) or *Dead* (0) values into *gridBoard*. We also transformed it into a numpy type of

board in order to improve efficiency. Moreover, we will also be storing by appending the board info into *outputArray*.



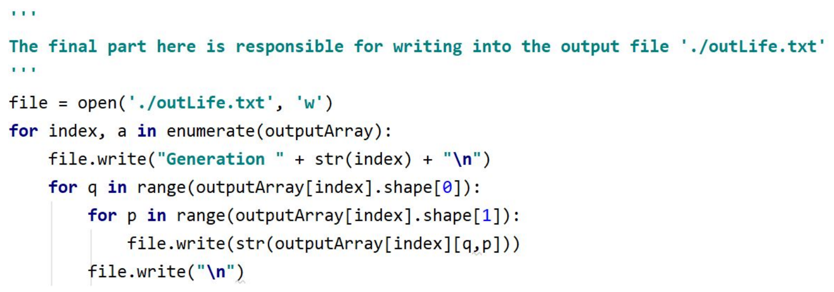
```
'''
The following part is responsible for opening the file and reading the input from './inLife.txt'
'''
file = open('./inLife.txt', 'r')
for countStarter, line in enumerate(file.readlines()):
    line = line.strip('\n')
    if countStarter == 0:
        firstLine = line.strip()
        numberOfGenerations = int(firstLine)
    else:
        gridBoard.append([LIVE if int(currentState) == 1 else DEAD for index, currentState in enumerate(line)])
board = np.array(gridBoard)
height, width = board.shape
outputArray.append(board) # Appending the board info into outputArray array
```

3. In the third part of our program, we defined the following part to be responsible for generating neighbors and producing next generations. The outermost for loop iterates over each number of generations. The status of each cell changes each turn of the game (also called a generation) depending on the statuses of that cell's 8 neighbors. In this sense, the variable *numberOfAliveNeighbors* specifies the number of alive neighbors of that cell's 8 neighbors. In our algorithm, we consider that if the cell is dead, then it springs to life only in the case that it has 3 live neighbors. On the other hand, if the cell is alive, then it stays alive if it has either 2 or 3 live neighbors. Finally, we append the *newBoard* info into *outputArray*, and update the *board* with *newBoard*'s information.



```
for i in range(numberOfGenerations):
    newBoard = board.copy()  # I make a copy of the board
    for row in range(height):
        for column in range(width):
            numberOfAliveNeighbors = (board[(row + 1) % height, column] +
                                      board[(row - 1) % height, column] +
                                      board[row, (column - 1) % width] +
                                      board[row, (column + 1) % width] +
                                      board[(row - 1) % height, (column - 1) % width] +
                                      board[(row + 1) % height, (column - 1) % width] +
                                      board[(row - 1) % height, (column + 1) % width] +
                                      board[(row + 1) % height, (column + 1) % width])
            if board[row, column] != LIVE:
                if numberOfAliveNeighbors == 3:
                    newBoard[row, column] = LIVE
            else:
                if numberOfAliveNeighbors > 3:
                    newBoard[row, column] = DEAD
                elif numberOfAliveNeighbors < 2:
                    newBoard[row, column] = DEAD

    outputArray.append(newBoard) # Appending the newBoard info into outputArray array
    board = newBoard
```

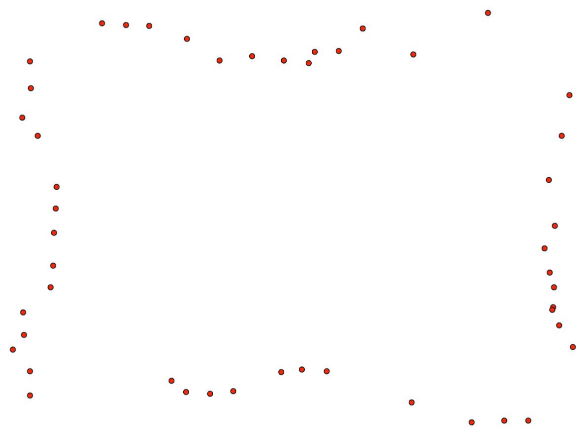
4. In the last part of our program, we defined the following part to be responsible for writing the final output content into the output file '*./outLife.txt*'. This part should be fairly straightforward where it writes into the output file by using multiple for loops iteratively by integrating the contents stored in the *ouputArray* that we previously defined. As well, when writing, the first line of the output for each generation will specify which number of generation the current grid belongs to, and this is also the reason as to why we used the outermost for loop to accomplish this.



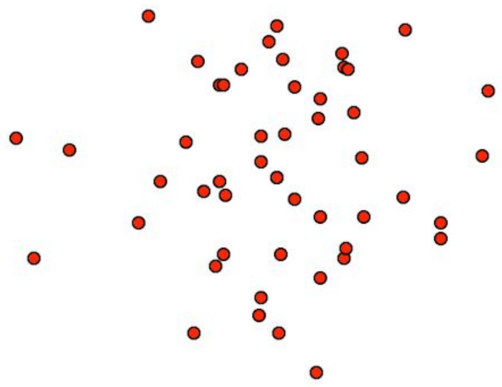
```
'''
The final part here is responsible for writing into the output file './outLife.txt'
'''
file = open('./outLife.txt', 'w')
for index, a in enumerate(outputArray):
    file.write("Generation " + str(index) + "\n")
    for q in range(outputArray[index].shape[0]):
        for p in range(outputArray[index].shape[1]):
            file.write(str(outputArray[index][q,p]))
        file.write("\n")
```

Results

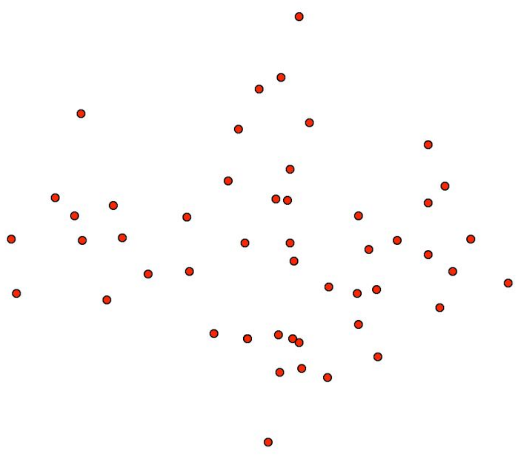
<u>Craig Reynolds' Boids System (Part1)</u>



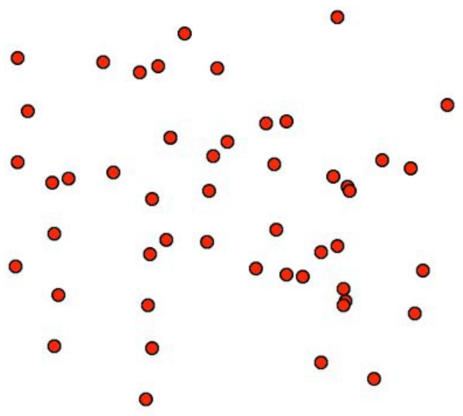
(1)



(2)



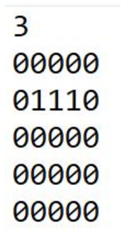
(3)



(4)

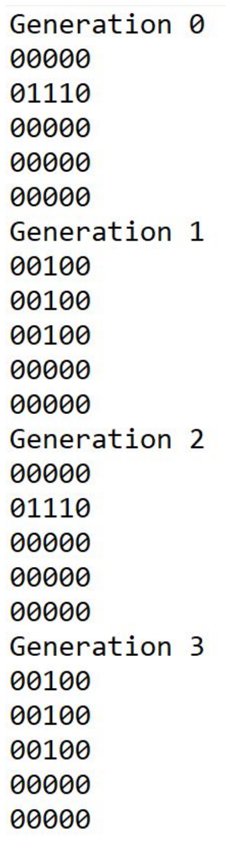
Figure 1: outputs of file boids.py

<u>Conway's Game of Life (Part2)</u>



```
3
00000
01110
00000
00000
00000
```

Figure 2: inputs of sample input file inLife.txt



```
Generation 0
00000
01110
00000
00000
00000
Generation 1
00100
00100
00100
00000
00000
Generation 2
00000
01110
00000
00000
00000
Generation 3
00100
00100
00100
00000
00000
```

Figure 3: outputs produced from sample input file inLife.txt

Conclusion

In summary, the Craig Reynolds' Boids System and Conway's Game of Life are both very interesting systems for us to explore in Artificial Life in the field of Artificial Intelligence. By simulating the Craig Reynolds' Boids System, it helps us better understand how animals such as flocks of birds and schools of fish work cooperatively with the three rules (Cohesion, Alignment, and Separation) of Artificial Life specified.

On the other hand, the Conway's Game of Life System gives us an opportunity to explore the game where neighbors of a cell as well as the next generation evolution are taken into account along with the predefined set of rules specified in the Life System, such as whether or not cells are alive or dead when there are either two or three neighbors. Overall, it's an interesting game of life system for us to explore.

References

S. Givigi. (2020). Assignment 3, Artificial Intelligence, CISC 352, Winter, 2020.