DREXEL UNIVERSITY
Electrical and
Computer Engineering
College of Engineering

ECEC 301
Programming for Engineers

Programming Assignment 1: Conway's Game of Life

Professor: James A. Shackleford

Section 062

Yonatan Carver
yac25@drexel.edu

Due Date: October 10, 2017

Yonatan Carver 10.10.2017
ECEC 301 – 062 Project #1

Introduction

The goal of this programming assignment is to implement Conway's Game of Life simulation using Python 2.7. The Game of Life is a cellular automaton, or simulation, that models the life and death of cells in a world. A cellular automaton is a mathematical model with specific rules that dictate whether a cell in a simulation lives or dies.

Theory

Live cells are represented by black pixels whereas dead cells are represented by white pixels. As time progresses, the cells change states (i.e. they live or they die). The current states of each cell, along with the number of neighbors each cell has determines its future state. A cell's neighbors are the eight cells surrounding it. All cells in the world follow four simple rules to determine what its future state will be:

1) If the current cell is alive and has less than two neighbors, the cell dies (under population)
2) If the current cell is alive and has two or three living neighbors, the cell lives (stable population)
3) If the current cell is alive and has more than three living neighbors, the cell dies (over population)
4) If the current cell is dead and has exactly three neighbors, the cell becomes a live cell (reproduction)

In the case of this experiment, the user can choose to start the simulation with four types of patterns. The starting pattern determines how cells will react in the world. The four types of patterns are: stills, oscillators, spaceships, and generators. Still patterns do not move and remain in their initial state forever. Oscillators are objects that oscillate back and forth between specific states - these objects will neither die nor reproduce. Spaceships are patterns that translate across the world but do not reproduce. Generators are objects that can reproduce.

Implementation

Task 1

The first function, `generate_world(opts)`, generates the world in which the simulation occurs while accepting a single command-line argument. The goal of Task 1 is to populate the world with either live cells (black) or dead cells (white). If the `opts` argument contains a world type of `random` (line 5), a random 10% of the world will be living cells (line 6). Likewise, if the user types in a world type of `empty` or does not enter a world type (line 7), the world will not be populated with any living cells (line 8).

```
1  ## TASK 1 ##########################################################################################
2
3  world = []
4
5  if opts.world_type == 'random'
6      world = [[(random.choice([1,0], p = [0.1,0.9])) for x in range(opts.cols)] for y in range(opts.rows)]
7  elif opts.world_type == 'empty'
8      world = [[0 for x in range(opts.cols)] for y in range(opts.rows)]
9
10  ###################################################################################################
```

Task 2

The second function, `blit(world, sprite, x, y)`, inserts an object from the file patterns.py into the world that was created in Task 1. The process of copying a smaller block of pixel data into a larger block of pixel data is called blitting (block image transfer). This function requires four inputs: `world`, `sprite`, `x`, and `y`. The `world` argument brings in the world that was created in Task 1 (whether it be random or empty). The `sprite` argument brings in a sprite object from the patterns.py file – this file contains various still, oscillating, spaceship, and generator patterns. The arguments `x` and `y` are the coordinates at which the top left corner of the sprite is placed into the world. Line 4 of Task 2 "sweeps" the rows in the world, getting each row number and value, while line 5 gets each individual pixel in each row – which is a specific pixel in the world. Line 6 steps through the world, assigning each corresponding pixel in the sprite to a pixel coordinate in the world.

```
1  ## TASK 2 ##########################################################################################
2
3
4  for row_index, row in enumerate(sprite):
5      for col_index, item in enumerate(row):
6            world[row_index+y][col_index+x] = item
7
8  ####################################################################################################
```

Task 3

The third function, `update_frame(frame_num, opts, world, img)`, contains the code that runs the simulation that is the Game of Life. There are four main rules that the Game of Life follows: under population, over population, reproduction, and stable population (outlined in the Theory section of this report). Each of these four rules depend on the number of neighbors each pixel has and whether that pixel is dead or alive.

The work that is done in the `update_frame` function is done in a copy of the world (called `new_world` in this program). Once everything finishes updating in `new_world`, it is “pushed” into frame so the user can visualize the updates to the world.

To determine the new state of each pixel, the code sweeps each pixel’s eight surrounding neighbors (lines 3 – 10). Depending on the number of neighbors and the state of the current pixel, the future pixel state is either dead or alive.

After sweeping and getting the number of a pixel’s neighbors, there are two options. If the current cell we are inspecting is alive, lines 16 – 25 are executed. If the current cell is dead, lines 28 – 32 are executed. Lines 15 – 32 are the four basic rules translated into the Python programming language. Line 34 updates the cell in the world.

```
1  ## TASK 3 ##########################################################################################
2
3  for row_index, row in enumerate(world):
4        for column_index, item in enumerate(row):
5            live_neighbors = 0 #placeholder
6            for x_location in [-1, 0, 1]:
7                for y_location in [-1, 0, 1]:
8                    x_loc = (column_index + x_location) % opts.cols
9                    y_loc = (row_index + y_location) % opts.rows
10                    if world[y_loc][x_loc] == 1:
11                        live_neighbors += 1
12
13            future_state = 0
14
15            if item == 1:
16            # center cell is alive
17                live_neighbors -= 1
18                if live_neighbors < 2:
19                    future_state = 0
20                elif live_neighbors > 3:
21                    future_state = 0
22                elif live_neighbors == 2:
23                    future_state = 1
24                elif live_neighbors == 3:
25                    future_state = 1
26
27         # dead rules
28            else:
29                 if live_neighbors == 3:
30                     future_state = 1
31                 else:
32                     future_state = 0
33
34            new_world[row_index][column_index] = future_state
35
36  ###################################################################################################
```

Yonatan Carver
ECEC 301 – 062

10.10.2017
Project #1

Testing Results + Screenshots

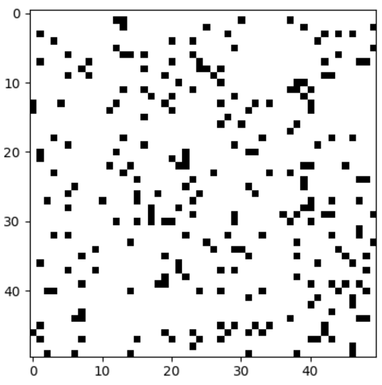

Figure 1: Task 1 - A world populated with 10% living cells
"python gameoflife.py -w random"

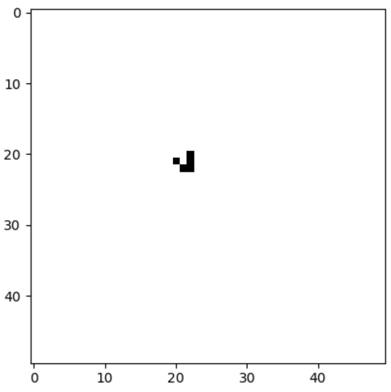

Figure 2: Task 2 - A glider placed at (20, 20)

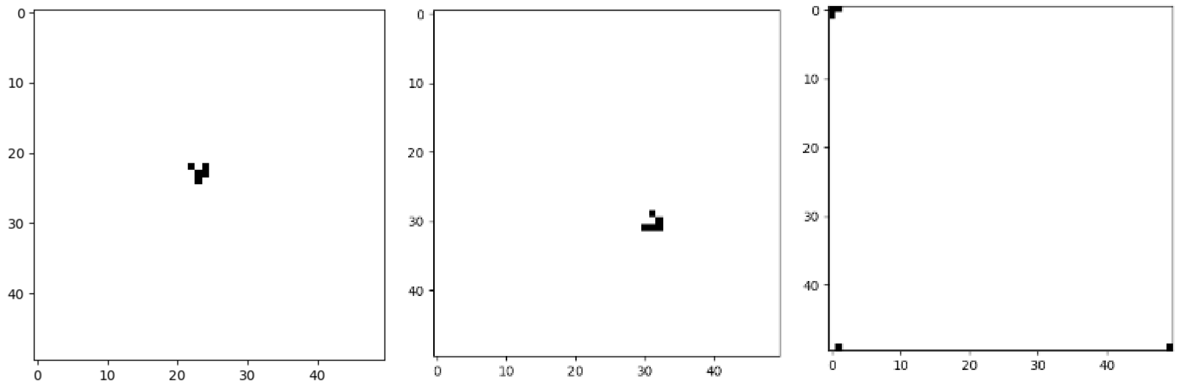

Figure 3: Task 3 - The glider pattern is being updated in the world
"python gameoflife.py"
Note: Objects wrap around the world as seen in image 3

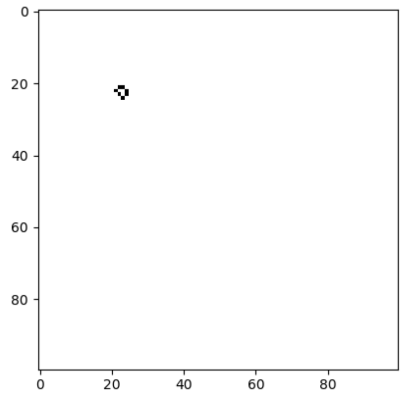

Figure 4: Loaf (still pattern) placed at (20,20)
in a world of 100 x 100
"python gameoflife.py -r 100 -c 100"

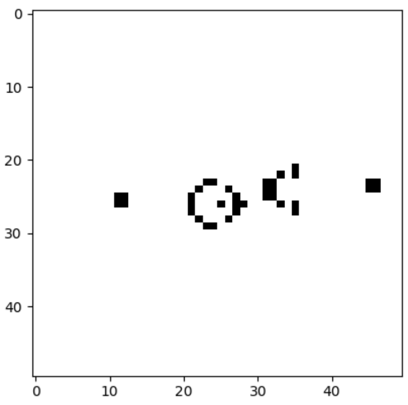

Figure 5: Gosper gun (generator) placed at (10,20)

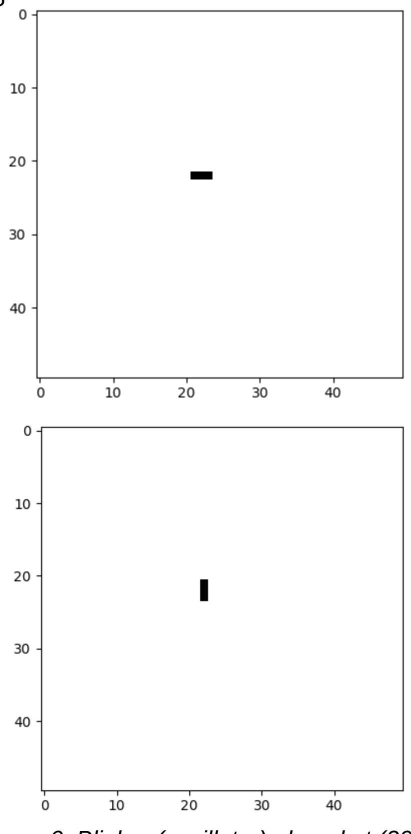

Figure 6: Blinker (oscillator) placed at (20,20)