Conway's Game of Life

Nathan Schomer

Drexel Ecec301, Spring 2016

Background

Conway's Game of Life is a simulation performed within a two-dimensional grid. The grid contains cells, each of which is assigned a value of 1 or 0. A value of 1 is considered "alive" while a value of 0 is considered "dead". As the simulation steps, each cell's next after the next step is dependent on the values of it's 8 surrounding neighbors. The rules dictating the state of the cell after the next step are:

1. If a live cell has less than two live neighbors, it "dies" from underpopulation.
2. If a live cell has greater than three live neighbors, it "dies" from overpopulation.
3. If a dead cell has exactly three living neighbors, it becomes alive in the next frame.
4. If a live cell has between two and three living neighbors, it remains unchanged.

Note- Cell positions wrap-around the grid edges. For example, in a grid of size 10x10, a cell at position (0,0) is considered a neighbor of (10,0), (0,10), (10,10) among others.

Starting Point

A Python-based framework for the simulation was provided. This framework excluded 3 portions:

1. World Creation- A world of specified dimension must be created using either random values or empty (0) values for each cell.
2. Blitter- This function copies a shape, or "sprite", into the world at a specified location.
3. Update Rule- This function is the engine of the simulation. It reads through the current world and generates a frame buffer with updated values for each cell depending on the update rules listed above.

Source Overview

generate_world(opts)- Generates 2d pixel buffer with dimensions specified in arguments otps.cols and opts.rows. The contents of this buffer is determined by the value of opts.world_type which is either random or empty. If the world is random, it will have approximately 10% live cells.

update_frame(frame_num, opts, world, img)- This function acts as a callback function for the simulation. During callback, update_frame() will create a copy of the current world, update it with new values per the rules listed above, and then update the visible grid with the new values.

blit(world, sprite, x, y)- The Blit() function will copy the contents of sprite into world with the topmost left corner of the sprite located at world position (x, y).

run_simulation(opts, world)- This function generates a plot using the world generated in the generate_world() function.

report_options(opts)- This function prints information to tell the user what command line options are available for use.

get_commandline_options()- This function retrieves and parses command line options provided by the user.

main()- This function creates the world, gets command line options, blits in a sprite and then runs the simulation.

Task 1 Approach & Solution

For task 1, a world of a specified type needed to be created with specified dimensions. In order to create this world, this function checks the type of world which will either be 'random' or 'empty'.

If the world is 'random', a temporary row of length opts.cols is created. The value of each position with the row is determined with the "random" function. This function returns a float value between 0 and 1. In order to give each cell a 10% chance of being initialized alive, the random function is called and if the result is less than or equal to 0.1, the cell is set to 1. The temporary row is then appended to the world. This process is repeated until opts.rows number of rows have been appended to the world.

If the world is of type 'empty', a 2-dimensional grid of dimensions opts.cols by opts.rows is created using nested list comprehensions. This task is contained within the generate_world() function.

```
## TASK 1 ##########################################################
#
#

if opts.world_type == 'random':
    for col in range(opts.cols):
        #create temporary row before it's written to the world
        tmp_row = []
        #create requested number of rows
        for row in range(opts.rows):
            #each cell has a 10% probability of being alive
            if random.random() <= 0.1:
                tmp_row.append(1)
            else:
                tmp_row.append(0)
        #append the temporary row to the world
        world.append(tmp_row)

elif opts.world_type == 'empty':
    #create empty world of requested dimensions
    world = [[0 for i in range(opts.cols)] for j in range(opts.rows)]

#
#
###################################################################
```

Task 2 Approach & Solution

For task 2, the blit() function had to be updated to copy a sprite into the current world at a specified position. This function iterates through the sprite using enumerate(). Enumerate() iterates through the sprite but also provides the current index within the sprite. Within each row in the sprite, the function iterates through each value in the row and then writes the value to the world at the calculated indexes.

The target indexes are calculated by adding the current index within the sprite to the start index specified in the function arguments and modding the result by the length of the row or column to compensate for potential wrap-around. For example, if the target index was row_idx + x = 10 (zero-indexed) but the world only contained 10 rows, this function would place the value in row 0.

```
## TASK 2 ###########################################################
#
#

#iterate through each row in world
for row_idx, sprite_row in enumerate(sprite):
    #iterate through each column in current row
    for col_idx, curr_val in enumerate(sprite_row):
        #find target indexes in world, and compensate for wrap-around
        target_row = (row_idx + x) % len(world)
        target_col = (col_idx + y) % len(world[0])
        #assign value of current position in sprite to target position in world
        world[target_row][target_col] = curr_val

#
#
####################################################################
```

Task 3 Approach & Solution

This task was contained within the update_frame() function. The update_frame function creates a frame buffer containing a grid with values updated per the rules of the simulation.

This function works by enumerating through each row and colum of the world, calculating the index of the 8 surrounding cells and then retrieving the values of each of those cells. These 8 values are then summed into “cell_sum” which is used to determine the future state of the current cell per the rules of the simulation. The updated value of the current cell is then written to the frame buffer.

```
## TASK 3 ###############################################################
#
#

for row_idx, row in enumerate(world):
    for col_idx, col in enumerate(row):
        #index values of rows and colums surrounding current cell
        # useful for calculating indexes of surrounding cells
        # compensate for possibility of wrap-around
        last_row_idx = (row_idx - 1) % len(world)
        next_row_idx = (row_idx + 1) % len(world)
        last_col_idx = (col_idx - 1) % len(row)
        next_col_idx = (col_idx + 1) % len(row)

        #get values of all 8 surrounding cells
        top = world[last_row_idx][col_idx]
        bottom = world[next_row_idx][col_idx]
        left = world[row_idx][last_col_idx]
        right = world[row_idx][next_col_idx]
        top_left = world[last_row_idx][last_col_idx]
        bottom_left = world[next_row_idx][last_col_idx]
        top_right = world[last_row_idx][next_col_idx]
        bottom_right = world[next_row_idx][next_col_idx]

        #sum values of surrounding cells
        cell_sum = top + bottom + left + right + top_left + bottom_left + top_right + bottom_right

        curr_cell = world[row_idx][col_idx]
        #use sum of surrounding cells to determine if cell lives or dies
        if cell_sum > 3 or cell_sum < 2:
            curr_cell = 0
        elif cell_sum is 3:
            curr_cell = 1

        #write value to new_world (the frame buffer)
        new_world[row_idx][col_idx] = curr_cell
#
######################################################################
```

Test Cases

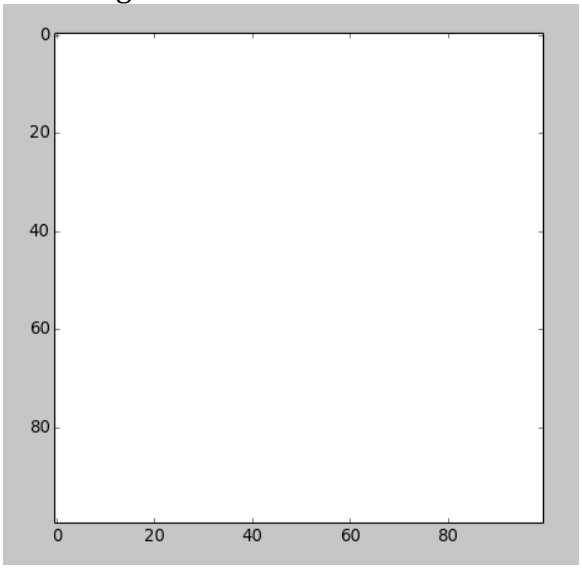
1. **Empty World of Size 100x100 (non-default dimensions)**
2. **Random World**
3. **World with Glider**

Test Case #1: Empty World

Command:

```
parallels@ubuntu:~/Ecec301/project1$ ./gameoflife.py -r 100 -c 100
Conway's Game of Life
=====================
   World Size: 100 x 100
   World Type: empty
  Frame Delay: 100 (ms)
```

Resulting Window:



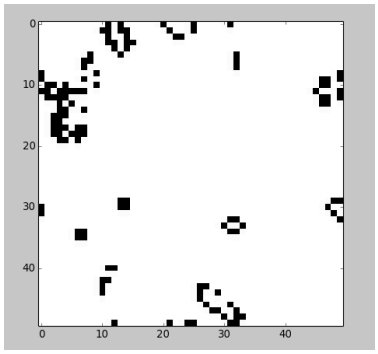
Description of results- An empty window of size 100x100 was created.

Test Case #2: Random World

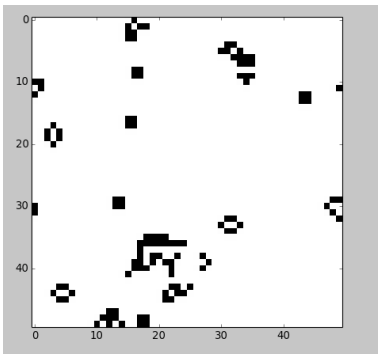
Command:

```
parallels@ubuntu:~/Ecec301/project1$ ./gameoflife.py -w random
Conway's Game of Life
=====================
    World Size: 50 x 50
    World Type: random
  Frame Delay: 100 (ms)
```

Resulting Window:



...after 5 seconds:



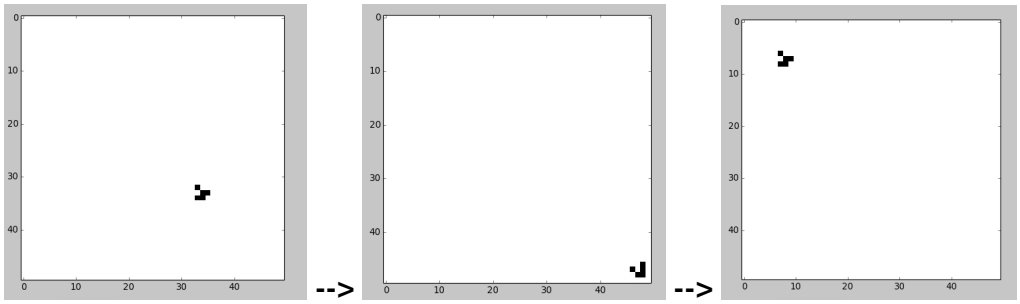
Description of Results- A random grid was generated with dimensions 50x50. The grid changed as expected per the rules of the simulation.

Test Case #3: Blit in Glider

Command:

```
parallels@ubuntu:~/Ecec301/project1$ ./gameoflife.py
Conway's Game of Life
=====================
   World Size: 50 x 50
   World Type: empty
  Frame Delay: 100 (ms)
```

Resulting Window:



Description of Results- A grid of size 50x50 was created with a glider sprite located at position 20x20. The glider moved diagonally down the screen eventually wrapping around the grid.