Conway Technical Document

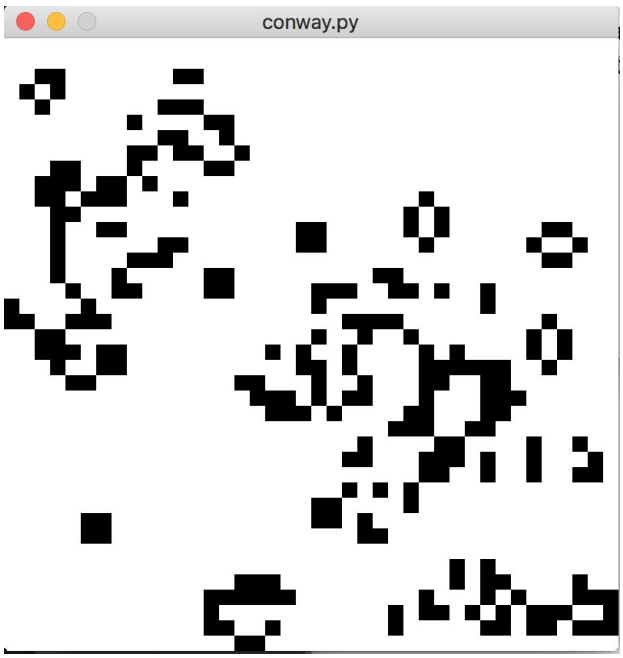
CISC 352 Assignment 4

Anna Ilina (10150979)
Jake Pittis (10148335)

Introduction

We implemented Conway's Game of Life in Python. We used the Pyglet package to create a 2D graphics display to visualize the cells. The application can be given a text file as a first argument which will dictate the game starting pattern. If no input file is specified, a random starting pattern is generated.

Because a GUI is used to output the game state, game state is not outputted to a file or to standard output. Simulation will continue until the program is exited.



A screenshot of our GUI running a random simulation.

State Structure

We decided to store our board as a two dimensional list of booleans representing whether a cell is alive.

```
def create_board(width, height):
    random.seed()
    board = []
    for y in range(0, height):
        row = []
        for x in range(0, width):
            if random.random() < DENSITY:
                row.append(True)
            else:
                row.append(False)
        board.append(row)
    return board
```

A simple double for loop for generating a new board. We set `DENSITY` to a small percentage to populate a small percentage of the board with living cells.

```
def parse_board():
    def num_to_bool(num):
        num = int(num)
        if num == 1:
            return True
        else:
            return False
    b = []
    generations = 0
    first = True
    for line in fileinput.input():
        line = line.strip()
        if first:
            generations = int(line)
            first = False
            continue
        b.append(list(map(num_to_bool, list(line))))
    return b
```

Similarly when we are provided an input file, we parse the lines to create a board. We ignore the `generations` value because our graphical display loops until closed by the user

Counting Neighbours

The core of of this algorithm involves counting the number of living neighbours around a cell.

```
OFFSETS = [[1, -1],
        [-1, 1],
        [1, 1],
        [-1, -1],
        [-1, 0],
        [0, -1],
        [1, 0],
        [0, 1]]

def count_neighbours(x, y):
    def within_bounds(coord):
        return coord[0] >= 0 and coord[0] < CHEIGHT and \
                coord[1] >= 0 and coord[1] < CWIDTH
    def is_living(coord):
        return board[coord[0]][coord[1]]
    def add_offset(coord):
        return [coord[0] + y, coord[1] + x]
    neighbours = list(map(add_offset, OFFSETS))
    neighbours = list(filter(within_bounds, neighbours))
    return len(list(filter(is_living, neighbours)))
```

For a given cell located at `x`, `y`, we generate the coordinates for the 8 neighbouring cells. We then filter the cells that are off the map and then count the number of living cells.

The Game of Life

We hold the old game state in the `board` 2D list and build the new game state in the `next_board` 2D list. We iterate over the old board and call `update_cell` on each cell and apply the rules of the Game of Life.

```
def update_cell(x, y, alive):
    n = count_neighbours(x, y)
    if alive and (n == 2 or n == 3):
        next_board[y][x] = True
    elif not alive and n == 3:
        next_board[y][x] = True
    else:
        next_board[y][x] = False
```

Rendering and GUI

GUI is handled by the Pyglet package. We iterate through the current board and render each cell at the correct location on the screen.

```
def on_draw():
    pyglet.gl.glColor4f(1, 1, 1, 1)
    draw_rectangle(0, 0, WIDTH, HEIGHT)
    pyglet.gl.glColor4f(0, 0, 0, 0)
    for y, row in enumerate(board):
        for x, cell in enumerate(row):
            if cell:
                draw_cell(x, y)
```

Conclusion

We found that rendering graphics using Python was quite slow. We had to try a number of graphics libraries before we found one that was fast enough. We were generally surprised our implementation of a graphical Conway's Game of Life was so short in terms of lines of code. (129 lines of code with liberal spaces.) Jake and Anna pair programmed program.