Anthony Burwinkel
CS5001 Spring 2022
Final Project Report

Project Description:

For my final project, I created an implementation of Conway's Game of Life using Python's pygame module. In Conway's game, a grid of "cells" is created. At the beginning of the game, each cell in the grid is marked as either alive or dead. After this, each cell's next state is calculated based on some rules, a new grid is made based on these rules and the current grid's state, then the new grid replaces the old one. The game of life is watching this happen over and over again. The following rules govern life and death in the game:

- If a cell is alive:
  - If it has less than 2 living neighbors, it dies (of loneliness)
  - If it has 2 or 3 living neighbors, it continues living
  - If it has 4 or more living neighbors, it dies (of overpopulation)
- If a cell is dead:
  - If it has exactly 3 living neighbors:
    - It comes back to life
  - Otherwise:
    - It stays dead

n	n	n
n	c	n
n	n	n

In the diagram, the cell (labeled 'c') in the middle, has 8 neighbors, each labeled 'n'. This is the case for every cell in the grid. They all are each either living or dead, and they each have 8 neighbors that are either living or dead. The game can go on forever, depending on the initial configuration and the size of the grid.

I implemented Conway's game by creating a 2d array (list of lists) of boolean values (1s and 0s). I wrote sets of functions that would look at each position in this array of boolean values, check the cell's 8 neighbors, and evaluate what the cell should be in the next round of the game. The state of this game board was then rendered as a set of rectangles to the screen by a custom display module I wrote using pygame's draw module.

The rectangle objects were all stored in a parallel array to the game board, so my implementation used an associative array as the main data structure. I used loops through ranges to write functions that would take the state of the game board's array and transfer it to the rectangles array as it was drawn to the screen. I also used the stack data structure that we built in class to implement a go_back_one_round function that could take us as far back in the game of life as we want. Each time a new game state is calculated from the old one, the game logic module pushes the current game board's state onto the stack before creating the new one and then rendering it through the display module.

Alterations from Planning:

Originally I planned to implement the cells in the grid as Cell type objects with an "alive" attribute and a "get neighbors" function. When I began designing the logic of the game, I realized how much simpler it would be to simply track the state of the game with an array of boolean values. Calculating neighbors would be simplified to summing the values of the neighbors in the array (which would be 1 if alive or 0 if dead).

Designing the game board this way made designing the game display straightforward. If I just made an array the same size as the game board of pygame rectangle objects, and wrote a function that would render each rectangle as one color if the corresponding place in the game board was a 1 and another color if the space was a 0, I could have one array mirror the other. So the booleans array plays the game, and the rectangles array displays each step as it is played. This is the associative array structure that makes the whole application work.

I included a few features unmentioned in the planning as well. These occurred to me as I was writing the application and were easy enough to implement or of sufficient interest or both that I decided to include them. The step forward and back buttons were not in the original plan, but make the game much more interesting in my opinion. The step counter in the upper right was challenging to set up but I think it was worth it, as it makes it possible to see how many iterations away from the start we are.

My application also makes use of the modulo trick we used to create a circular buffer for our queue class. In this case, I use the modulo operator in the calculate neighbors function so that the game board will wrap to the other side. This means that every edge of the board thinks that it is the opposite edge's neighbor.

Reflection:

I learned quite a lot in this project. Designing the game engine was a really fun exercise, and I'm happy with the simplicity of the associative array structure I ended up using to implement the game. Using the stack was surprisingly simple and showed me the power of general use data structures. Once you have one designed, you can use it all over the place.

The other large challenge of the project was refactoring. I wrote the pygame portion all in one file at first, as one ~300 line mess of objects, event loops, and configurations. The game logic was always separated into its own file, but writing it as an object was a challenge, as well as writing the pygame display updating functions as object methods. By abstracting away the game logic and the display methods and configuration, I was able to strip the main function of the application to a very minimal inner and outer event loop. There are no function definitions in main, only function calls to other modules. What really worked out about refactoring is that I organized all the display rectangles into dictionaries, so that the rendering function draws from those dictionaries to render everything. This means adding more rectangles and settings will be easy, since an organized structure is already defined.

If I could do the project over, I would have taken more care to design a display that would dynamically adjust to the screen, or user configuration. As it is, the game runs with hard coded pixel settings. I would also have made more options for grid settings, like cell size. I may still implement more changes. I am pleased with the level of modularity that the application has now. It's easy to add rectangles and colors to the config object. I think it's easy to change and maintain the code.

Acknowledgements:

Pygame documentation:

https://www.pygame.org/docs/ref/rect.html

- Useful for figuring out how to render rectangles in different colors to the display

https://www.pygame.org/docs/ref/font.html

- Needed this to write text to the buttons

Game of Life Research:

https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

- Description of conway's game and rules

https://en.wikipedia.org/wiki/Oscillator_(cellular_automaton)

- Finding repeating patterns to test out