Megan Fanning
CS162

Conway's Game of Life

Design

I started my design by creating my algorithm to determine when cells would live or die and from there determined which other functions would be needed to support the algorithm.

The grid will be displayed on a two nested vectors. There will be a current board and a future state board, the births and deaths will be recorded in the future board. Once all the births and deaths have been tallied then the future board and current board pointers will be swapped. Then the current board will be printed to the screen by the printBoard function, which will check the cell's boolean value and if true print a “8” to the board.

The cell life cycle will be managed by a function which checks all adjacent cells to determine the status of the cells. This will be done using nested for loops which check a three by three grid, with the target cell in the center. If the target cell is alive it should not be counted as a adjacent live cell.

I considered a few options to allow an infinite grid, stitching the top and the bottom of the grid together, trying to identify when a group of cells had left the 'live' grid etc. However I found the simplest solution was to create a set of ghost cells beyond the visible grid, and then kill cells once they were 3 past the visible grid.

Features List:
-print board (displays the visible section of the board)
-lifecycle (kills and births the cells and maintains the edges of the grid)
-clear array function (removes all the users designs from the board)
-welcome screen (contains menu options and guides user through setting up a pattern)
-user seed function (accepts spawn point and number of rounds to display)
-stop button (allows user to quit the program)
(Note these are the features, there will be other supporting functions as necessary).

Figuring out how to seed the grid but still allow the user to move the design was a little tricky. What I decided to do was create a “base” seed which the users could shift down and to the right by adding the user input to the X and Y coordinates. I chose to limit how the user could give imput on shifting the design to ensure they would not accidentally shift something outside of the visible grid.

Making the screen speed up and slow down required some googling on my part. I ended up using a user's suggestion from this stack overflow thread: http://stackoverflow.com/questions/4184468/sleep-for-milliseconds. However td::this_thread::sleep_for(std::chrono::milliseconds(x)); isn't supported by the flip libraries, so I had to convert to usleep.

Lessons learned

One of my biggest debugging issues came from using rows and columns as my variables instead of x and y coordinates, which resulted in me swapping the coordinates and getting bad data in my print outs.

One of the best debugging methods I found was to print the number of neighbors each cell had instead of just printing a '8'. This allowed me to check for errors in my algorithm by being able to immediately see the number of neighbors.

I ended up scrapping the stop button feature because I felt making the user hit enter each lifecycle of the board was clunky. Instead, I allowed the user the choice of how many life cycles to display, and then let them choose to stop or continue when the animation is finished.

I tested each of my “ghost walls” by adding a rule to delete the edge row (or column) and then sending a glider gun in the direction of my ghost wall to see if it would collide or crash the program. Much of my testing was incremental. I would write my function and then test it several ways before adding another function.

When asking the user how many spaces they wanted to shift the design, if they reply with something other than a number, it will create an infinite loop. However, if they reply with an invalid number, it will properly cycle. I was hoping to do more input validation, but since it wasn't included in the requirements I opted to implement it last, so a number of my functions would break if the user put in bad input. As such, much of my testing was testing expected cases.

I tested each of the possible designs the user could select, as well as speeding up and slowing down the speed of the graphics. For the glider, I placed it on the board on several locations to confirm it would exit the screen completely. The most interesting test I found was where when I would place multiple seeds at the same time, such as when I placed a glider gun and an oscillator on the screen which resulted in the glider gun 'exploding' into a number of oscillators some of which were half on the visible screen, half off, that continued to blink, which I found very satisfying as confirmation that my algorithm was creating the illusion of a window to an infinite grid.

I used oscillators to test my speed since they have a consistent speed and pattern. That way, I could go through more cycles of the oscillation in a shorter time frame, which helped me confirm they would continue to oscillate.