CS 2337 – Extra Credit 4 – Conway's Life System

Life is a Bit!

Dr. Doug DeGroot's C++ Classes

Due: Sunday, November 19, 2023, by midnight

Key concepts: cellular automata, the onset of chaos, artificial life, the emergence of order from chaos, the butterfly effect, theory of computation

How to submit: Upload your entire project (code, output files, any notes, etc.) to the eLearning site. The code must compile on either Codeblocks or MS Visual Studio 2015 (or later). Include your name in the code filename, like so:

EC4-CS2337-Jane-Doe.cpp

Due to a USB hardware failure, I've temporarily lost my codes & assignment files for both the Mandelbrot and Wolfram's One-Dimensional Cellular Automata system. I'll upload them as soon as I can recover them.

Maximum Number of Points: 20

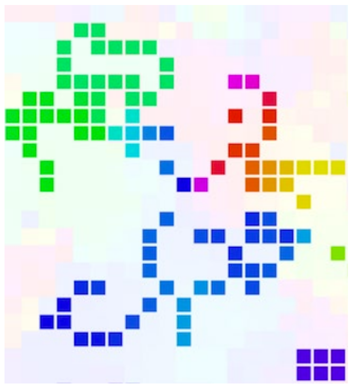
Objective:

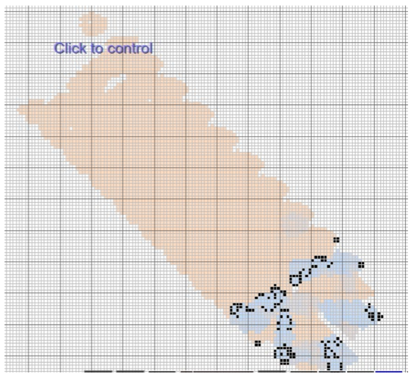
Write a program that implements the Game of Life cellular automata system invented by John Conway.

1. Create two game grids of size at least 50x50. These grid cells can be either Boolean or integer – probably start with Boolean values. In the following, I'll refer to these two grids as gridOne and gridTwo.
2. Start by initializing gridOne. Allow the user to specify three different ways of initializing the grid:
    - by specifying an input pattern file to be read or
    - by randomly initializing some percentage of the grid's cells.
    - by entering cell coordinates manually

   One or more sample life pattern files are included in the homework folder on eLearning. One is named **lifePattern-Glider.txt**. It specifies one way to set up a simple glider (but there are other ways). Use this pattern file to start. Once you get the simulation working correctly, you can download other files from the web and use them for your simulations as well. (See the information below on **Input.)**
3. After initializing gridOne, print it to the console and then pause the program so the user can view the game world to ensure that the pattern was read correctly and that the grid was correctly initialized. Resume when the user enters Y, y, ENTER, Yeehah, or whatever.
4. Using gridOne, evolve the grid (calculate the conditions for all cells for the *next* generation of the game) and store the results in gridTwo. Then use gridTwo to evolve the system into gridOne for the next generation of the simulation. Flip back and forth from gridOne to gridTwo to gridOne and so on, using one grid to calculate the other.
5. Print the next generation grid at each turn. In other words, if you're evolving gridOne, store the next state of the world into gridTwo; and if you're evolving gridTwo, store the next state in gridOne. Flip back and forth this way and print out the appropriate grid at each step. **NOTE:** Be sure to clear the screen before printing out a grid, otherwise you won't see the evolution in action.
6. Once the simulation begins, you might want to pause the game to inspect the state of the cells. So, add a pause feature just as we did in the Snake Game.

7. Make sure you are handling your edge cells properly. There are two approaches you can take. You could wrap the grid from side to side and top to bottom. Alternatively, when a live cell reaches the edge of a grid, don't wrap the world around; just let edge cells die a lonely death if they want to. Consider neighboring but non-existent edge cells as 0's. or 1's, your choice
8. When printing a grid on the user's screen, use '.' or '-' or a space for dead cells. For live cells, use 'O' or 'X'.
9. **Experimenting:** Once things are working well, try using larger and larger grid sizes (or even smaller, if you want). Also, try finding other Life Patterns on the web and running them with your program. And *be sure* to use some random initializations at some points and play around with different percentages of initial live cells. You should also test your program with the other pattern files I've uploaded to eLearning.
10. **Stopping and Printing:** At some point, you will likely be tempted to pause your game, print it out to a PPM file, and quit. I'd like you to submit three or four of your game evolutions. Please make them impressive.
11. **Multicolored Output:**
12. Instead of using a Boolean grid of cells, perhaps use an integer grid so you can use multiple colors. Now, when you evolve a grid from one time to another, don't just use 0 and 1 values, but use integer counts for the cells. When a cell dies, set it's next value (in the opposite grid to zero). But if a cell lives from one generation to the next, increment the cell count. This cell count will record the number of generations the call has been alive. Use different colors for different life spans. You might end up with something similar to the following examples (but the first one is meaningless, and the right one is nearly useless! 😊):





Resources:

1. Check out the Life Lexicon at http://www.conwaylife.com/ref/lexicon/lex.htm for numerous pattern files.
2. See Wikipedia's articles on John Conway and Conway's Life System (Game).
3. You can find a bunch of material on this site: https://conwaylife.com/
4. Play the Game of Life: https://playgameoflife.com/
5. And there are many, many more great web sites you can find.
6. I highly recommend the book called ***Chaos: Making a New Science*** by James Gleick. (This is not the same "new kind of science" as Wolfram's "new kind of science.")

Game of Life Rules

OK, now we need to know how to evolve the world (the grid).

Each cell has at most 8 neighbors (cells on the edge of the world will have fewer, depending on your edge scheme; be let’s focus on internal cell only for now):

1	2	3
4	Cell	5
6	7	8

1. If a cell is alive and has 2 or 3 neighbors, it stays alive.
2. If a cell is alive but has fewer than 2 neighbors (0 or 1), the cell dies of loneliness.
3. If a cell is alive and has 4 or more neighbors, it dies from overcrowding.
4. If an empty cell has exactly 3 neighbors, it becomes alive.
5. All births and deaths happen simultaneously. (Thus, this is a discrete-event simulation system.)

Input:

If the user asks you to randomly initialize the grid, ask the user for a percentage of cells to turn on (e.g., 20 or 70, meaning 20% or 70%, etc.). Then go through gridOne turning cells either on or off with that percentage. (Use a random number generator to determine which.)

On the other hand, if the user asks you to initialize the grid by reading from a file, ask the user for the name of the file. Check for the file’s existence by attempting to open it for reading. Assume the file is a plain text file that contains a Life grid pattern. The file will be in plain ASCII format. There can be both comment lines and pattern lines in the file.

Comment Lines: Comment lines will start with the ‘#’ symbol. Read these comment lines and output them to the user. Then skip right past them, as they are only informative comment lines.

Pattern Lines: Pattern lines will contain a series of ‘.’ and ‘O’ characters. A ‘.’ character indicates that the corresponding cell is dead while a ‘O’ character indicates that the cell is alive. Read each pattern line and set the cells in the corresponding row of gridOne accordingly.

Example Life Pattern File: Here is a sample of what a Life Pattern file looks like when coded as a plain ASCII text file:

```
#Name: Gosper glider gun
#Author: Bill Gosper
#The first known gun and the first known finite pattern with unbounded growth.
#www.conwaylife.com/wiki/index.php?title=Gosper_glider_gun
#9x36 – size of this pattern
#
........................O
......................O.O
............OO......OO............OO
...........O...O....OO............OO
OO........O.....O...OO
OO........O...O.OO....O.O
..........O.....O.......O
...........O...O
............OO
```

Start loading the pattern into your gridOne at starting location something like 10,10 (i.e., don't start loading the pattern near the edges of your grid.) Make the 10, 10 location variable, not hardwired. The reason we load the pattern into an offset location (e.g., 10,10) is so the pattern won't be affected by loading it starting at the top edge of the world. We need to allow room for the pattern to grow and move around, if it wants to.

Try this:

1. While the simulation is running, try to predict what's going to happen. Is the automata going to completely die out and leave an empty world behind? Will it reach a stable state where there are still live cells but nothing is happening (there are many stable organisms). Will it reach a cyclic state where it repeats itself every *n* generations (like the blinker or even the 5-cell glider)? Or perhaps it will continue to change but never really increase in size or complexity. Can you guess from looking at it? Could you predict this with another computer analysis program, perhaps a neural-network program or some sort of predictive data analytics program?

Annotations for the code:

1. The main function can be at either the beginning or the end of the program. I don't care which.
2. Add comments at the top of your program file to include your name, the name of the program, and any relevant notes on how your design works when executed. Add a change log in your comments that list the major changes you make to your logic – nothing too terribly detailed, but a list of breadcrumbs that will remind you and others of what you've done as your program becomes more sophisticated and/or nearly complete.
3. Point out (in the comments at the top of your program) any special features or techniques you added using a comment saying something like "// Special Features."

4. Comment your code effectively, as we discussed in class. Use descriptive variable names everywhere so that the code becomes as self-documenting as possible. Use additional commentary only to improve readability and comprehensibility by other people.
5. Use very good program decomposition techniques (no large routines/functions). Break your routines down into subproblems and implement each separately. Try to have only one major control construct in each routine when possible; if more are truly needed for clarity of the computation, feel free to use more than one.
6. You absolutely MUST use consistent indentation and coding styles throughout the program. Failure to do so will result in a loss of three points.
7. Don't make your code unnecessarily hard to read by forcing all variables to be local variables. You can use global variables when appropriate.
8. If the program does not work at all, or works incorrectly, at least 10 points will be deducted.
9. Poorly structured, poorly documented, hard-to-read code, non-self-documenting code, etc. will all result in loss of ***at least*** 5 points. Write clear, extensible, maintainable code.