Game of Life rush

Summary: In this rush, you will implement Conway's Game of Life, and attempt to make your implementation as fast as possible.

Chapter 1

Introduction

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

[Source: Wikipedia]

The Game of Life is almost trivial to implement. However, the purpose of this rush is optimization: your implementation should not only be correct, but also as fast as possible. There are many ways to think about optimization, including the following ones:

- Choice of language
- Use of compiler optimizations
- Modelling (choice of data structures to represent the game state)
- Algorithm efficiency (complexity analysis can be a useful tool)

And even advanced methods:

- Multithreading
- Cache optimization

Chapter 2

Rules of the Game of Life

The universe of the Game of Life is an infinite, two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with two or three live neighbors survives.
- Any dead cell with three live neighbors becomes a live cell.
- All other live cells die in the next generation. Similarly, all other dead cells stay dead.

[Source: Wikipedia]

Chapter 3

Instructions

- You can use the language of your choice. However, you cannot use any third-party library. Only the core language and its standard library are allowed.
- The focus of this rush is not error handling. You can assume that your program will be tested with valid input files.
- Your program should be invoked as follows: **./life initial_state iterations** (though you may choose any name for your program)
- The program should then print out the state of the simulation after computing **iterations** steps from the given **initial_state**.
- The format for the printout should be identical to the one of the **initial_state** file.

3.1 File format

initial_state is a text file representing the initial state of the automaton. It is in the following format:

- One or more lines
- Every line is of the same length (at least one character) and ends in a newline character
- Lines contain only the two following characters: . (to represent a dead cell) or **x** (to represent a live cell)

For instance:

```
% cat state
.................................
.................................
......x.................x.........
......x.................x.........
......x...............xxxx.........
.................................
...................xxxxx.........
.........x.......................
.........xxx.....................
..........x......................
...............xxxxx.............
.................................
.................................
%
```

3.2 Boundary condition

Although the game was described as an "infinite" grid of cells, your implementation should be restricted to a grid of the size given by the **initial_state** file. When counting neighbors, you must assume that every cell outside these boundaries is dead, and always stays dead.

In particular, corner cells have only three neighbors, and side cells have 5, instead of the normal 8. The transition rules are unchanged.

3.3 Optimizations

Once your program is correct, you will attempt to make it faster by applying optimizations of your choice. You **MUST** submit at least 2 functional versions of your program, in order to demonstrate a speed gain during the evaluation.

They can be either separate programs, compile-time flags, or runtime settings, depending on your language and preference.

Chapter 4

Bonuses

- Make a graphical interface for the game, that shows all the intermediary states before **iterations** steps. You may add any controls you wish to the program, such as pause, continue, skip to the N-th next step, and live editing of the game world. (For this bonus, you can use an external library or even make a separate tool).
- Implement a truly infinite game world.
- Implement a way to choose different transition rules. For instance, you can get inspiration from the condensed notation "B3/S23", which describes the Game of Life as "birth from 3 live neighbors, survival from 2 or 3 live neighbors". Using this notation, other interesting cellular automata include Highlife (B36/S23), Replicator (B1357/S1357) and Seeds (B2/S).

Chapter 5

Evaluation

As usual, the evaluator will clone your git repository, and should be able to run your project themselves. Since you have the choice of language, make sure that your project can run on the school iMacs, and to include *complete, written instructions* on how to install the requirements. The evaluator will check that you used only the core language and its standard library.

If you write your project in C, you are not required to follow the Norm. However, regardless of the language, you are still expected to provide source code that compiles and/or runs without errors for valid inputs.

You may use the highest level of optimizations your compiler provides. For instance, using **`gcc`** or **`clang`**, you can use **`-flto`**.

During the evaluation, you should not only demonstrate that your program is correct, but also explain all the different optimizations you have applied, why, and which were most significant in terms of speed gains. These versions will be run and measured with the evaluator.