ComS 229 Project 2 — Final

The Game of Life and other Cellular Automata

November 1, 2013

0 Introduction

This is a "change" document. In other words, rather than writing another entire spec, this document focuses on what should be changed relative to the prototype phase of the project.

1 Cellular Automata

A *cellular automaton* is a regular grid of cells, in any number of dimensions. At any point in time, each cell in the grid may be in one of a finite number of states. The state of a cell evolves over time according to the states of its neigbors, according to various rules. The dimensionality and shape of the grid, the number of states for each cell, and the rules for changing states define the *type* of the automaton. For example, Conway's Game of Life is one type of cellular automaton.

Cellular automata[1] are examples of models of computation. Each type of automaton can be viewed as a type of computer, while the initial configuration can be viewed as a *program* for the computer. Theoreticians are interested in studying the limits of what may be computed with various types of computers.

For the final version of this project, you will extend your programs to handle some other types of cellular automata. You will always deal with finite, two–dimensional, rectangular grids.

1.1 Wire World

In Wire World, each cell may be in one of *four* states:

0. Empty,
1. Electron head,
2. Electron tail,
3. Wire.

The rules for changing states are as follows.

State at generation n	State at generation $n+1$
Empty	Empty
Electron head	Electron tail
Electron tail	Wire
Wire	Electron head, if exactly one or two neighbors are electron heads; Wire, otherwise.

The *neighbors* of a cell are the same as Game of Life — the eight adjacent cells. See `http://en.wikipedia.org/wiki/Wireworld`.

[1] "Automata" is the plural form of "automaton".

1.2 Elementary Cellular Automata

An elementary cellular automaton is one–dimensional, but the rules below have been modified for two dimensions. Each cell is either in state 0 or 1. The rules for changing states are as follows. If a cell is in state 1, then it remains in state 1. Otherwise, the cell's new state is determined by looking at the pattern of cells above. Specifically, for cell (x, y), look at the pattern of cells at $(x-1, y+1)$, $(x, y+1)$, and $(x+1, y+1)$. The new state of the cell is then determined by the *rules* for the automaton, which are specified in the *Wolfram code* for the rules. In the Wolfram code, the eight possible patterns are written in order 111, 110, ..., 001, 000. Underneath, a 0 or 1 is written. The Wolfram code is the number whose binary representation corresponds to those 8 bits.

For example, consider the rule table below.

Above pattern	111	110	101	100	011	010	001	000
New state	0	0	0	1	1	1	1	0

According to this table, if the current state of cell (x, y) is 0, and the states of cells $(x-1, y+1), (x, y+1), (x+1, y+1)$ are $0, 1, 1$, then the next state of cell (x, y) is 1. Taking the bits 00011110 and converting to decimal, we get the Wolfram code 30; this is known simply as Rule 30.

See `http://en.wikipedia.org/wiki/elementary_cellular_automaton`.

2 The programs

2.0 `life`

Modify this program to also handle the cellular automata described in Section 1 of this document. The input file format is described in Appendix A of this document.

2.1 `life_gui`

- Modify this program to also handle the cellular automata described in Section 1 of this document.
- Modify this program so that the grid display has horizontal and vertical scrollers.
- Modify this program so that, if the `-c` switch is specified, there is a dialog box that controls the simulation. You should implement the following in the controller dialog box:

  1. The ability to advance to the next generation, i.e., compute and display it.
  2. The ability to continuously run the simulation, i.e., compute the next generation, display it, delay for some time, and repeat.
  3. The ability to set the duration of the delay between generations.
  4. The ability to pause the simulation.
  5. The ability to restart the simulation back to generation 0.
  6. An indication of the number of the current generation.
  7. The ability to set the size of the grid squares.
  8. The ability to quit the simulation.

  An example interface is shown in Figure 1. Note that you are allowed to design your own interface, as long as it supports the above features.

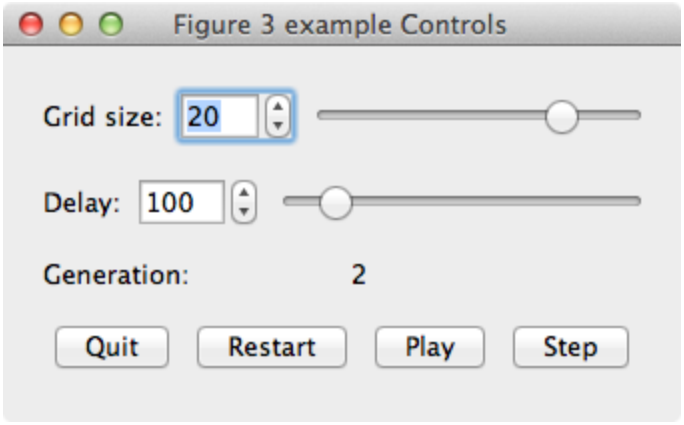

Figure 1: An example control dialog box

3 Limits

Your terrain will be bounded in both the x and y directions, and will be a finite rectangle. You may assume that the cells outside of this rectangle remain in the default state of the automata, regardless of what happens inside the rectangle.

4 Submitting your work

You should turn in a gzipped tarball containing all source code, a makefile, and a `README` file that documents your work. The tarball should be uploaded in Blackboard.

Your executables will be tested on `pyrite.cs.iastate.edu`. You should **test early, and test often** on `pyrite`. We will use some scripts to check your code; this means you should not change the name of the executables.

5 Grading

The following distribution of points will be used for grading the final version.

life — prototype : 150 points
: Program `life` behaves as required for the prototype.

life_gui — prototype : 150 points
: Program `life_gui` behaves as required for the prototype.

life_gui — controls : 200 points
: The grid scrollers, and the control dialog box.

other automata : 300 points
: The points are divided out "evenly":

- `life` with Wire World support: 75 points
- `life` with Elementary support: 75 points
- `life_gui` with Wire World support: 75 points
- `life_gui` with Elementary support: 75 points

makefile : 50 points

Simply typing "`make`" should build all of your executables. Also, "`make clean`" should remove the executables and any other automatically–generated files. You may include other targets for your convenience as you choose (e.g., "`make tarball`"). Note that this `Makefile` will need to be more clever than the last project, because you will need to let Qt automatically generate a makefile for `life_gui`.

Also, note that if your code does not *build* because of compiler or linker errors (either using `make` or by hand), you will likely lose much more than 50 points. Effectively: the TAs reserve the right to not grade your project *at all* if it fails to compile or link. **Test early, and test often on `pyrite`.**

Documentation & style : 150 points

50 points for the `README` file, as in the prototype. 100 points for your actual design, and documentation (i.e., comments) of your source code.

Usability : 100 points

Style, from the end user's perspective. Are the programs easy to use? Is the GUI interface nice and responsive? Are error messages informative?

Total (final) : 1100 points

A Input file format

The input files for each automaton type are similar in format to the "life" format given for the prototype phase of this project, and are in fact a generalization of this format. Specifically, the first identifier in the file will be one of "`Life`", "`WireWorld`", or "`Elementary`". The "`Name`", "`Terrain`", and "`Window`" assignments are allowed in any type of automaton, and have the same format. The "`Chars`", "`Colors`", and "`Initial`" assignments are allowed in any type of automaton, but the names of the identifiers within the structs are changed to correspond to the state names for the automaton:

Life has states "`Dead`" (the default) and "`Alive`".

WireWorld has states "`Empty`" (the default), "`Head`", "`Tail`", and "`Wire`". See Figure 2 for an example file.

Elementary has states "`Zero`" (the default) and "`One`".

Finally, an "`Elementary`" struct allows a statement of the form "`Rule = <integer>;`" which defines the rules of the automaton according to the given Wolfram code. See Figure 3 for an example file.

```
WireWorld = {
  Name = "Diodes";
  Terrain = {
    Xrange = -12..12;          Yrange = -6..6;
  };
  Chars = {
    Empty = 32;                Head = 35;
    Tail = 43;                 Wire = 46;
  };
  Colors = {
    Empty = (64, 64, 64);      Head = (255, 64, 64);
    Tail = (255, 64, 255);     Wire = (64, 64, 255);
  };
  Initial = {
    Head =  (-9, 3), (9, -3);
    Tail =  (-10, 3), (10, -3);
    Wire =  (-8, 3), (-7, 3), (-6, 3), (-5, 3), (-4, 3),
            (-3, 3), (-2, 3), (-1, 3), (1, 3), (2, 3),
            (3, 3), (4, 3), (5, 3), (6, 3), (7, 3),
            (8, 3), (9, 3), (10, 3),
            (-1, 2), (0, 2),
            (-1, 4), (0, 4),
            (-10, -3), (-9, -3), (-8, -3), (-7, -3), (-6, -3),
            (-5, -3), (-4, -3), (-3, -3), (-2, -3), (-1, -3),
            (1, -3), (2, -3), (3, -3), (4, -3), (5, -3),
            (6, -3), (7, -3), (8, -3),
            (-1, -2), (0, -2),
            (-1, -4), (0, -4);
  };
};
```

Figure 2: An example wireworld file.

```
Elementary = {
  Name = "Rule 30";
  Rule = 30;
  Terrain = {
    Xrange = -200..200;        Yrange = -200..0;
  };
  Chars = {
    Zero = 46;                 One  = 35;
  };
  Colors = {
    Zero = (255, 255, 255);    One  = (64, 64, 64);
  };
  Initial = {
    One  = (0, 0);
  };
};
```

Figure 3: An example elementary cellular automaton.