ComS 229 Project 2 (part 1)

The Game of Life

March 15, 2013

0 Introduction

The "Game of Life" (GoL in this document) is a very simple but interesting model of life, devised by John Conway in the 1970's. It is not really a "game", because there are no players; it is probably more accurate to call it a "simulation". For this project, you will write some C++ programs to simulate GoL.

In GoL, the terrain is a two–dimensional grid of square cells, as shown in Figure 1. In this document, cells will be indexed using x and y integer coordinates. Theoretically, the terrain is infinitely large in all directions. At any point in time, each cell in the grid may be in one of two possible states: *alive* or *dead*. Time is also discretized, and everything evolves together in "steps" or "generations". The state of a cell evolves over time according to the states of its eight *neighbors* (see Figure 2), according to the following rules.

1. If a cell is *alive* in generation i, then:
   (a) if there are fewer than two *alive* neighbors in generation i, then the cell will be *dead* in generation $i + 1$ (from loneliness).
   (b) if there are two or three *alive* neighbors in generation i, then the cell will be *alive* in generation $i+1$.
   (c) if there are more than three *alive* neighbors in generation i, then the cell will be *dead* in generation $i + 1$ (from over-population).
2. If a cell is *dead* in generation i, then:
   (a) if there are exactly three *alive* neighbors in generation i, then the cell will be *alive* in generation $i+1$ (from reproduction); otherwise, it will be *dead*.

A simple example of evolution from one generation to another is shwon in Figure 3.

0.0 Warning 0: These are 'loose' specs

This document does not attempt to *rigorously* define what your code must do. There *are* cases that are left unspecified: some on purpose, and some because I did not think of everything. Similarly, the project is quite open–ended: there are several different, reasonable designs that can work effectively; it is up to you to choose one.

0.1 Warning 1: There will be more

Design your classes carefully, with future expansion in mind. You *will* be graded on your class design, as part of the "style" portion of the project.

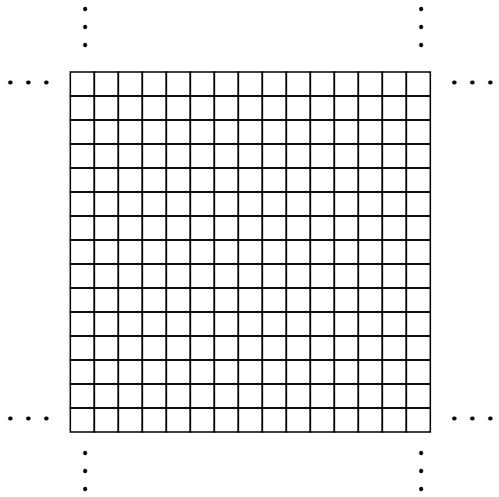

Figure 1: The terrain in The Game of Life. The grid continues forever in all directions.

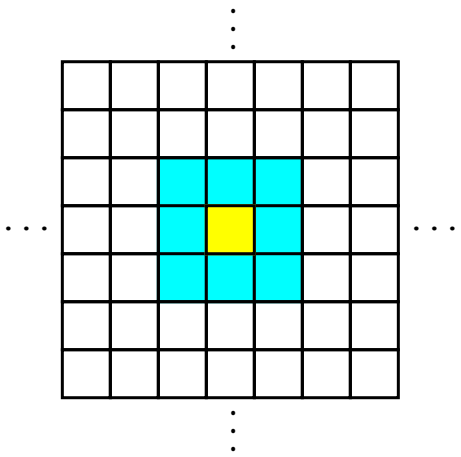

Figure 2: The eight blue cells are the neighbors of the yellow cell.

1 The programs

As a general rule, your programs should be "user friendly". That means they should print useful error messages whenever possible. Additionally, your programs should print any messages or diagnostic information to standard error, with standard output reserved for output. I do not require that your programs be absolutely "quiet", but take care that the default behavior prints at most a few messages. Finally, you may implement additional switches or features if you like.

For now, there is only one program.

1.0 `showgen`

This program reads a single `.aut` file (see Appendix A), either from a pathname passed as an argument, or from standard input if no file names are given as arguments. The input file specifies the size of the terrain

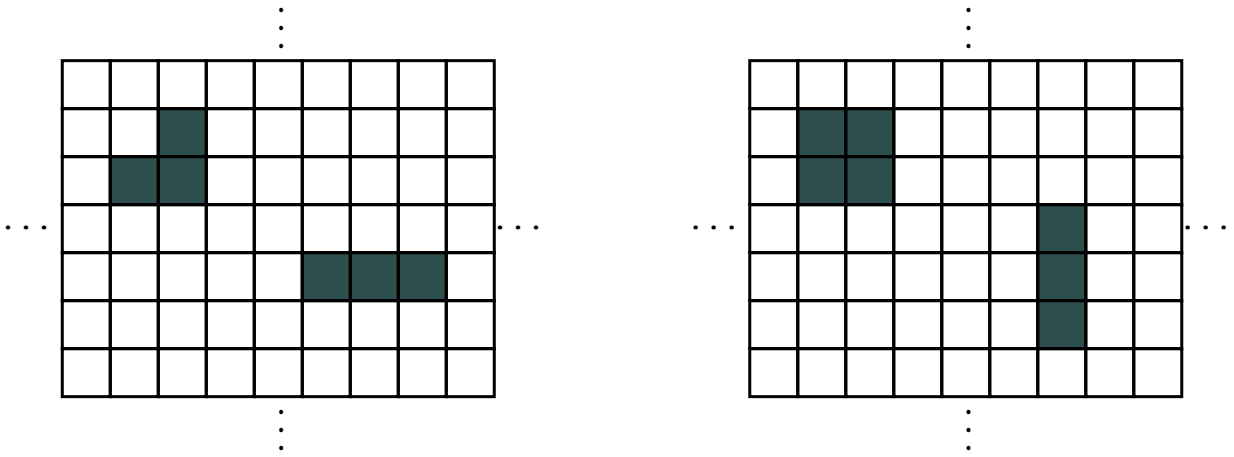

Figure 3: Example of evolution according to GoL rules. Grey squares represent *alive* cells and white ones are *dead* cells.

```
~~~~~~~~~
~11~~~~~~
~11~~~~~~
~~~~~~1~~
~~~~~~1~~
~~~~~~1~~
~~~~~~~~~
```

Figure 4: ASCII display of generation 1 in Figure 3.

and the initial configuration for generation 0. The program determines and displays the state of each cell at generation n. The program accepts the following command–line switches.

`-a`	Output should be in `.aut` format. Otherwise, the output should be an ASCII display of the terrain with a tilde character (‘`~`’) for *dead* and a 1 for *alive* (see Figure 4).
`-g` n	Specify the desired generation number. If omitted, the default should be $n = 0$.
`-h`	Display a help screen that describes the program.
`-tx` l, h	Set the x range of the terrain; overrides values specified in the `.aut` file.
`-ty` l, h	Set the y range of the terrain; overrides values specified in the `.aut` file.
`-wx` l, h	Set the x range of the output window; otherwise this defaults to the x range of the terrain.
`-wy` l, h	Set the y range of the output window; otherwise this defaults to the y range of the terrain.

2 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 *dead* state, regardless of what happens inside the rectangle.

3 A note on working together

This assignment is intended as an *individual* project. However, some amount of discussion with other students is expected and encouraged. The discussion below should help resolve the grey area of "how much collaboration is too much".

3.0 Not allowed

Basically, any activity where a student is able to bypass intended work for the project, is not allowed. For example, the following are definitely not allowed.

- Working in a group (i.e., treating this as a "group project").
- Posting or sharing code.
- Discussing solutions at a level of detail where someone is likely to duplicate your code.
- Using a snippet of code found on the Internet, that implements part of the assignment. If you have any doubt, check with the instructor first.

As a general rule, if you cannot honestly say that the code is yours (including the ideas behind it), then you should not turn it in.

3.1 Allowed

- Sharing test files (please post them on Piazza).
- Discussions to clarify assignment requirements, file formats, etc.; again, please post these on Piazza.
- High–level problem solving (but be careful — this is a slippery slope).
- Generic C++ discussions. If you have trouble with your code and can distill it down to a short, generic example, then this may be posted on Piazza for discussion.

4 Submitting your work

You do not need to turn in your solution to part 1; only the "final" version will be submitted. Eventually, you will turn in a gzipped tarball containing your source code, makefile, and a `README` file that documents your work (including your class design). The tarball should be uploaded in Blackboard.

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

A `.aut` file format

The `.aut` file format is a simple, free–form text file that contains statements of the form

```
Keyword <data>;
```

where keywords are case–sensitive strings of letters, and the `<data>` section is dependent on the keyword. Statements may include an arbitrary amount of whitespace (including newlines) between keywords and symbols. The character '`#`' means to ignore the rest of the line, and counts as a section of whitespace. A `<data>` section may consist of a compound statement, of the form `{ <stmt>; <stmt}; }`, where each `<stmt>` is another statement. However, there **will not** be nested compound statements. Except for compound statements, no `<data>` section will contain a semicolon.

You must handle the following keywords. Statements with other keywords should be ignored, by consuming the file until the end of the statement, and printing a warning message to standard error.

`Xrange` Sets the x range of the terrain. A statement is of the form `Xrange xlow xhigh;` where `xlow` and `xhigh` are integers.

`Yrange` Sets the y range of the terrain. A statement is of the form `Yrange ylow yhigh;` where `ylow` and `yhigh` are integers.

`Initial` Sets the configuration of generation 0, by specifying which cells are *alive*. This statement **must** be the last statement. This is a compound statement of the form

```
Initial { <initstmt>; <initstmt>; };
```

where each `<initstmt>` specifies the positions of some *alive* cells. Anything not specified as *alive* defaults to *dead*.

An `<initstmt>` has the form

```
Y = y : <xlist>;
```

where `<xlist>` is a comma–separated list of x values.

Other keywords will be added in later parts of the project. The file shown in Figure 5 is an example of a legal `.aut` file.

```
#
# Comments, to be ignored
#

This  bogus keyword should cause everything to be ignored
until the next semicolon, allowing for future expansion ;

Another { more evil bogus keyword;  we must also consume;
  the inner bogus statements; };

# Ugly formatting, but allowed:
Xrange
-10 10; Yrange -5 5;

Initial {
  Y = 2 :    -1;        # Sets (-1, 2) to "alive"
  Y=1:       -2, -1;    # Sets (-2, 1) and (-1, 1) to "alive"
  Y=-1:      2,3,4;     # Sets (2, -1), (3, -1), (4,-1) to "alive"
};
```

Figure 5: An example `.aut` file