CS116 - Intro to Programming, C++
Summer 1997

Project 1, due Thursday 8/14/97

The Game of Life

Introduction
The Game of Life was invented by the mathematician John Conway. It is one of a growing class of what are called “simulation games”, games that resemble real-life processes. The simulation takes place on a two dimensional board which we will call the universe. The universe is divided up into squares, or cells, just like a chess board. To begin with, each cell is either “alive” or “dead”. From then on, the universe evolves in discrete time steps, or generations, during which cells die and come to life according to the rules of the game.

Rules of the Game
The rules of the game specify how the universe evolves from one generation to the next. First let’s confine our universe to have a finite size. We note that each interior cell has eight neighboring cells (four adjacent orthogonally, four adjacent diagonally). The boundary cells have fewer neighbors, and the exact number of neighbors of a boundary cell depends on whether it lies in the corner or not.

What about the rules of the game? These rules are such as to make the behavior of the population both interesting and unpredictable. The rules are:

- **Survivals.** If a cell is alive, and has two or three live neighbors, then it survives for the next generation.

- **Deaths.** Every cell with four or more live neighbors dies from overpopulation. Every cell with one live neighbor or none dies from isolation.

- **Births.** Every dead cell adjacent to exactly three live neighbors - no more, no fewer - comes to life in the next generation.

It is important to understand that all births and deaths occur simultaneously. Together they constitute a single generation in the complete “life history” of the initial configuration of the universe.

You will find the population constantly undergoing unusual change. In a few cases the population eventually dies out, although this may not happen after a
great many generations. But most starting patterns either reach stable figures
-Conway calls them “still lifes”- that cannot change or patterns that oscillate
forever. Patterns with no initial symmetry tend to become symmetrical. Once
this happens the symmetry cannot be lost, although it may increase in richness.
As a simple example, try to find out how the following pattern evolves (black
cells are alive, white ones dead).

How to Get A Copy of the Code
Make a copy of the assignment folder Life for yourself. You can get a copy from
MacLab Resources : Courses : Summer 1997 : CS 116 : Life You can compile and run this project, although all you will see is a black graphics
window (all cell’s are dead).

Organization of the Code
In this project, you are going to implement the Game of Life and add some
extensions. Much of the code is provided for you, so let’s go over the basic
skeleton. There are three main classes:

• The class CLifeApp. This class implements the Macintosh application
which runs the simulation. You don’t need to do anything with this class.

• The class CUniverse. This class implements our universe. The most
important members of this class are:
  
  – mCellArray: This member variable is the array of cells which inhabit
    the universe.
  – initializePopulation: This is the function which is called at the
    very beginning of the simulation. For now this function initializes all
    cells to be dead.
  – StepSelf: This member function gets called periodically, and simu-
    lates the evolution from one generation to the next.

• The class CCell. This class implements the cells. The most important
members of this class are:
- mPreviousState and mCurrentState: These two member variables are of the enumerated type ECellState and record the states of the cell.

- mNeighbor[8]: This member variable is an array of eight CCell pointers. Each element of this array points to one of the eight neighbors of the cell (or is set to NULL if there is no neighbor in the respective direction).

- startingNewGeneration: This member function is called at the beginning of every generation.

- countNeighbors: This member function takes an argument of type ECellState and returns the number of those neighbors of the cell which are in this state. For example, we can use this function to count the number of dead neighbors of the cell.

- stepSelf: This member function is responsible for updating a cell’s state from one generation to the next. It gets called once per generation.

- drawSelf: This member function is responsible for drawing the cell onto the graphics window. It gets called once per generation, right after stepSelf.

Note that the above is only a bare bones summary of the program. Your first task is to read the source code and understand how everything works. Here is a list of the files and what they contain.

- CLifeApp.h and CLifeApp.cpp: You don’t need to do anything with these files, although you are welcome to have a look.

- main.cpp: You don’t need to change this file either. All it does is fire up the application.

- prandom.h and prandom.cpp: These files implement a random number generator. Thanks to these files, you have at your disposal the functions prandom() and puniform(). Read a description of how to use these in prandom.h. Note: We could have used the standard C++ library’s random number generator, but the one implemented in here is statistically more accurate (i.e. looks more “random”). You need not modify these two files.

- CUniverse.h: This file contains the class declarations for CUniverse and CCell, the necessary enumerated types (e.g. EDirection and ECellState), prototypes for a few utility functions, and the definition of a few colors used in our simulation. Study and understand the relevant parts of this file.

- CUniverse.cpp: This file contains implementations of some of the functions from CUniverse.h. Study this file well. You do not need to modify this file for the first part of the project.
• **StudentCode.cpp**: This is where you get started. This file contains the definitions of those member functions of CUniverse and CCell which were not provided in CUniverse.cpp. Currently these functions are mainly stubs, which means that they contain either no code or some minimal code just to ensure proper compilation. Your job is to write proper bodies for these functions.

### Part I - Implementing the Basic Rules

[ 35pts: c20e3i0t2 ]

In this part you will implement the rules of the Game of Life, and start out the simulation with a random configuration. You will need to do the following:

(a) Define `countNeighbors` to return the number of neighbors whose states equal `neighborState`. For boundary cells, non-existent neighbors are treated as dead neighbors.

(b) Update a cell's state according to the rules of the game. This update should happen in `stepSelf`.

(c) Start out the simulation with a random configuration. For example you might decide that in the initial generation, every cell starts out as being alive with probability $\frac{1}{7}$ and dead with probability $\frac{6}{7}$.

Note: The condition `((prandom() % 4) == 0)` is `true with probability $\frac{1}{4}$ (Why?)`.

(d) Figure out why you need the function `startNewGeneration`, and use it.

### Part II - Aging

[ 20pts: c11e2s5t2 ]

(a) Extend the enumerated type `ECellType` to include the cases `EJustBorn` and `EJustDied` and modify the cell's states accordingly during the course of the simulation. Then define new colors for these new states, and modify CCell::drawSelf() to draw the proper color onto the graphics window.

(b) Extend the CCell class to have a new member variable `mAge` of type `long`. This new member variable should be `private` (Why?). Modify the constructor to initialize it, and extend CCell's interface to include one new member function `getAge` which does the appropriate thing.

(c) In CCell::drawSelf, use the function `InterpolateColor` to color the cell's square with a shading that corresponds to its age (decide what exactly this should mean, and implement it).
Part III - Killer Parasites

[ 35pts: c20e3s10t2 ]

(a) At any given time, a cell has a low chance of getting infested with the “killer parasite”. When this happens, the cell’s state changes to \texttt{KillerParasite}. In the next generation, the parasite moves to a random neighboring cell (or dies if it steps out of the boundary), leaving its former host dead. If the new cell to which the parasite has moved is alive, the parasite infests that cell in the same way. If the new cell is dead, the parasite goes to sleep until the cell comes to life, then immediately infests the cell and keeps going. At any given time, the parasite may disappear as mysteriously as it appeared.

(b) Each killer parasite should only have a limited life span to begin with (i.e. a certain number of generations). Killing a cell prolongs the life span of the parasite, whereas sleeping on a dead cell does not. Try to find a reasonable life span as to make your simulation unpredictable and interesting.

Feel free to elaborate as much as you like on these problems.

Extra Credit

[ 22pts: c10e5s5t2 ]

Create a sanctuary to keep out parasites. This should be a rectangular region in the universe where parasites are not allowed to enter. Color code the sanctuary. For instance you might color each cell in the sanctuary with a slightly different shade of its respective non-sanctuary color.