Conway's Game of Life

Evan Nichols

NPRG063 Winter 2020/21

What is the Game of Life?

Conway's Game of Life is a zero-player automaton that consists of a grid and its cells, which are either "alive" or "dead" (filled or not filled). John Horton Conway, the British mathematician, developed the Game of Life in 1970. This program will present the user with a simple but robust interface that presents a board for the Game of Life and various controls for the user to manipulate the state of the automaton. The user will be able to place some standard patterns, adjust the speed and size of the game, and choose from several known starting patterns that have already been discovered.

The game runs indefinitely, advancing from state to state. Each cell's status (filled/alive or empty/dead) in the next state is determined by its current status and these four rules:

- Any live cell with fewer than two live neighbors dies, as if by under-population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by overpopulation.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

... which can be consolidated into these three essential rules:

- **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 state & all other dead cells stay dead.**

These rules were developed by Conway to ensure the following conditions for the game:

- There should be no explosive growth.
- There should exist small initial patterns with chaotic, unpredictable outcomes.
- There should be potential for von Neumann universal constructors.
- The rules should be as simple as possible, whilst adhering to the above constraints.

Main Functionality

State Continuation

The game continues from state to state on its own, with no user input required aside form setting the initial state. Cells determine their status in the next state by these rules:

- 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 state & all other dead cells stay dead.

Implementation

Each instance of a cell keeps a list of its neighbors (up to 8 other cells). During each state, each cell takes a count of its active neighbors and based on this sets its "willLive" property to true or false. This repeats every state, for every cell. **Model.cs** holds the classes for the board and the cells. Each cell has two bools: "is alive in this state" and "will be alive next state", and a list of its neighbors. A board has a 2D array of cells, its dimensions (in integer variables), and the size of each cell.

Selecting Initial States

Many patterns have been documented and named since the game's creation. The user should be able to choose one of these as an initial state. Selecting one should reset the board so that the only live cells are those that are present in the pattern. Choosing an initial state should also pause the game, to give the user time to view the pattern before the game begins running. This pause also allows the user to add to the pattern if they wish.

Implementation

A drop-down list (a ComboBox) of initial state patterns is offered, and selecting one of those states fetches a string from the Patterns class. The string consists of rows of the pattern, with '+' marking live cells, and rows delimited by newline characters '\n'. This rows in the string are then pulled into an array of strings, and the board is compared with the array with cells corresponding to the '+'s in the strings having their isLive property set to true.

Manual Pattern Placement

The user should be able to select from some basic patterns and place them onto the board by clicking on it, which sets the appropriate nearby cells to alive, and also pauses the game.

Implementation

To choose a small pattern to insert into the board, a set of RadioButton tools are available. Activating one of these buttons deactivates all others, so only one pattern is ever chosen at one time. To insert the pattern into the board, the user clicks on the board. The coordinates

of the clicked point are sent to the DrawShape function, which sets the appropriate cells active, in relation to the cell with the clicked coordinates. Placing the cell activation into a **try** block, safely handles the case of the user accidentally clicking outside the bitmap.

Saving & Loading States

The user should be able to save the current state and load it at any time later.

Implementation

A ToolStrip is above the menu, and holds two buttons Save and Load. Activating Save sets a Board variable "backup" equal to the current board. Activating Load then sets the board equal to the backup. Ctrl+S and Ctrl+L will also activate Save and Load, respectively.

Grid Display

The user is able to toggle the display of grid lines, to clearly demarcate individual cells.

Implementation

Instead of drawing lines, toggling grid lines via a CheckBox item actually adjusts the size of the drawn rectangle for each cell. When grid lines are toggled on, the size of each cell drawn is slightly smaller than when lines are toggled off. This gives the effect of visual grid lines when the black board shows through the space between individual cells.

Graphical User Interface

The user is presented with the game board and a menu through which they can control the state of the game. The initial state, state duration, cell size, cell color, and presence of grid lines can be adjusted. The user should be able to disable the game from running (and also enable it). Additionally, the user is able to select a small pattern and insert it into the board, by clicking on a cell and turning on the nearby cells that correspond to the pattern.

Implementation

Two buttons "Run" and "Pause" allow the user to control whether the game continues. Pressing one deactivates the other. Cell size and the duration of each state are adjusted via two NumericUpDown items. Cell size is at minimum 1 and capped at 100. State speed is also at minimum 1 ms and is capped at 1000 ms. A ComboBox is used to change the color of cells via a drop-down list. The menu items for selecting initial states and inserting patterns manually are described above. In short, another ComboBox offers the initial state presets and a set of radio buttons allows the user to choose basic patterns to insert.

Source Code Organization

This program is a Windows Forms application developed in C# using Visual Studio 2019.

Forms

Two forms are presented to the user:

- **Form1** is the main user interface. It contains the board and all control options and interactions. The main feature is the game board, on which the cells are drawn from one state to the next. The user is able to change the pause and resume the game, change its speed, change the size of the board, toggle the drawing of grid lines, and change the filled cells' colors. The current generation is tracked and displayed.

  The user is also able to begin the game with several preset patterns, randomize or clear the board, and add some fundamental patterns to the board via mouse clicks directly on the board. The current state can be saved at any time and loaded later. Lastly, there is an *Info* which summons the only other form, which contains some text about how the game works.

- **Form_Info** is a small window that contains only a short summary of the game. It is created and shown when the user presses the *Info* button on Form1.

Classes

Form1 contains all of the controls for the user interface.

First, *timer1* advances the game and counts the generation. The game is driven forward by itself when the timer is enabled.

Then, the GUI interactions are handled via click and button press functions. The shape functions are called by the radio buttons, and they simply set the 'SHAPE' string which is used by *DrawShape()*. The functions to update the board when the size is changed, the cell color is changed, grid lines are toggled, or the board is cleared or randomized.

Lastly, this class holds the functions to draw the board and also to draw the basic patterns when the user places them. There are two *Reset()* instances, one of which takes a starting pattern and applies it. To draw the board, the program scans over the board and draws a filled square at the locations of live cells.

Model.cs contains the classes for the board and cells.

Cell is a class to represent the individual cells in the board, and its properties are two bools (if it is alive, and if it will live) and a List of other adjacent cells (its neighbors).

Board consists of a 2D array of cells, a size integer, and dimension integers. It has 3 functions: to find and set the neighbors of a cell, to set the next state of the board by setting cells' alive status, and to randomize the board by setting cells' alive status randomly.

Patterns contains a dictionary of patterns and their names. Each pattern is represented as a string, with newlines delimiting each line. The function *Fetch()* takes a string and finds the pattern with the matching name. The dictionary also contains a "not found" pattern.