The design and development of a program to play Conway's Game of Life

The Game of Life

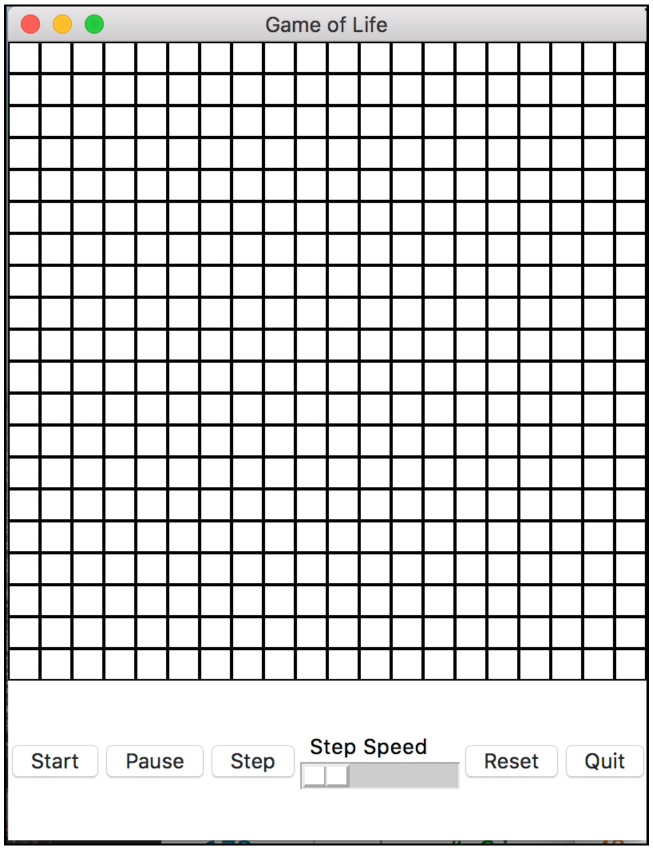
The Game of Life is a cellular automaton devised in 1970 by British mathematician John Conway. (See https://en.wikipedia.org/wiki/Conway's_Game_of_Life.) It is played on a rectangular grid of cells, where each cell can be in one of two states: alive and dead. The player of the game initializes each cell (to alive or dead) to start the game, and then, with no further input from the player of the game, the state of the grid updates in an unending sequence of discrete time steps according to a set of rules. The state (alive or dead) of a cell in a given time step depends only on the state of its 8 neighbor cells in the previous time step, the 8 neighbors being the cells that are (using directions to refer to their relative positions) north, south, east, west, northwest, northeast, southwest, and southeast, of the cell. Here are the rules that dictate the state of a cell in a time step:

1. If a cell is alive in a time step and has fewer than 2 or greater than 3 alive neighbors (in the same time step), then it is dead in the next time step. Otherwise, it stays alive in the next time step.
2. If a cell is dead in a time step and has exactly 3 alive neighbors (in the same time step), then it is alive in the next time step. Otherwise, it stays dead in the next time step.

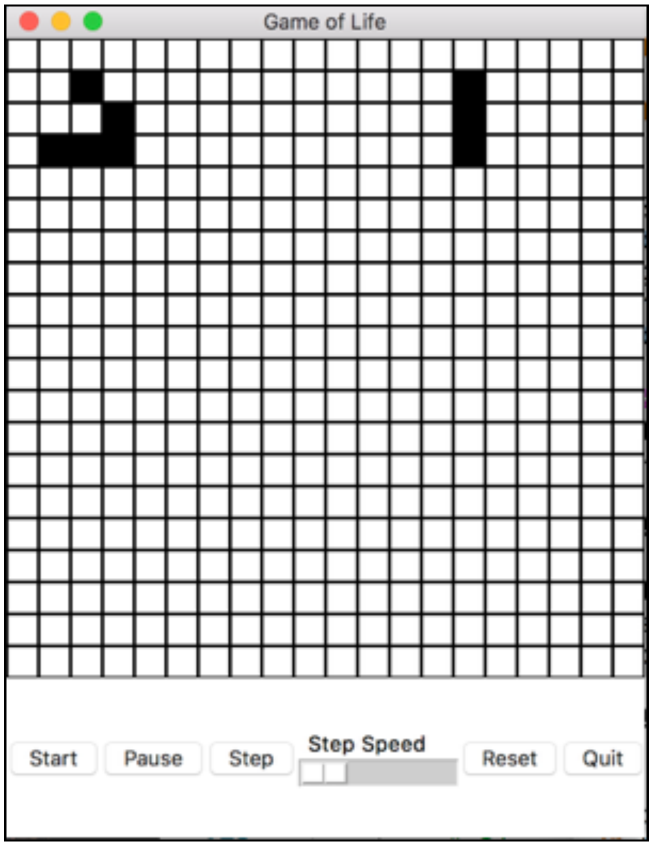
So the role of the player of Life is to decide which cells are to be alive initially. After that, the grid state evolves with no player intervention (other than to pause or quit the game). So playing the game is about experimenting with different patterns of initially alive cells, and watching how the state of the grid then evolves with time. Many interesting patterns have been discovered. (See the Wikipedia page linked to above.)

A program that plays the Game of Life

The file `lifeIteration6.py` contains a GUI Python program that plays the game of life. Here is a picture showing the window when the program is first started:



The top part of the window shows the grid of cells, which by default are all dead when the program starts. The user can then toggle the state of a cell by clicking on it, and here is a picture showing the window after some cells have been toggled to alive.



The pattern in the upper left corner is called a glider, and the pattern in the upper right is called a blinker. (The reasons for these names become apparent when you try playing the game with them.)

After the user initializes some cells to alive, the start button begins the game. The pause, step, reset, and quit buttons behave as you would expect; the step speed slider adjusts how quickly time steps are simulated; and if the game is in the paused state, clicking the step button advances the simulation by one time step.

Run the `lifeIteration6.py` program to try it out yourself. Try experimenting with different initial patterns of alive cells, getting ideas from the Wikipedia page.

Designing the Life program

The life program is relatively complex, at least by the standards of a second semester programming class, and so it benefits from being designed before it is written. Abstraction is the tool that programmers use to deal with complexity in programs: procedural abstraction seeks to break a task into multiple, simpler subtasks, and data abstraction looks to represent data types as the composition of simpler data types. Some programs benefit from using procedural abstraction, while some benefit from data abstraction, and programs that use data abstraction typically use procedural abstraction when implementing the data types.

The life program (and this is true of GUI programs in general) was developed using data abstraction, with three classes defined at the highest level:

1. A Model class: This class represents and updates the state of the life game. It uses a 2D (nested) list to store the state (alive or dead) of all the cells. It also contains the logic that implements the rules of the game (a dead cell becomes alive if it has exactly 3 alive neighbors, …) to compute the state of the grid in the next time step.
2. A View class: This class is responsible to displaying the game window. The top frame shows the grid, and the bottom frame (the control frame) displays the control buttons.
3. A Controller class: This class is responsible for controlling the game. It responds to user actions (e.g., clicking on the start button, or clicking on a cell), schedules the next time step, and takes charge of updating the game in response to a user event or time step.

Each of these classes by itself is less complex than the entire program, and so should be easier to develop. For example, the model class simply keeps track and updates the state of the game. It knows nothing about how the game is displayed or how the user interacts with it. (Writing the model class could have been an exercise in your first programming class.) Likewise, the view class only knows about the window that is displayed. It knows about the buttons, and the grid with its cells, but it knows nothing about the logic of the game. It draws all of the components (the widgets) in the window; it responds to user generated events (a button click, or a click on a cell) by telling the controller about the event; and it changes the state of a cell when it is told to by the controller.

Finally, there is the Controller class. The controller responds to events (user generated events like a button click or a click in a cell, and the self-scheduled time step event). To respond to events, the controller must talk to both the model and the view, and so the controller is coordinating the work of the model and the view. For example, when a time step event occurs, the controller first tells the model to update the state of the game (using the rules of the game). Then, for each cell in the grid, it asks the model what the state of the cell is, and then it tells the view to display that state for the cell. As another example, if the user clicks on a cell, the view signals to the controller that a cell was clicked on (and which cell it was), and the controller then tells the model to toggle the state of that cell, and it tells the view to change the display of that cell.

For this data abstraction to have a payoff - a lower overall complexity - the interfaces of the model, view, and controller classes must be clear and easy to understand. Put another way, each method of these classes must perform one well-defined and easy to understand task. For example, the model class in the game of life has a method with the header `def make_alive(self, row, col)` that makes alive the cell in a specified row and column.

This use of data abstraction is common enough that it has a has a name - the Model-View-Controller design pattern.

Writing the Life program

For programs of even moderate size (like Life), use of software development techniques when writing your code can be useful. One such technique is called incremental development, and in it the program is written in iterations, with each iteration being a complete working program (that can be tested) that introduces new features to the program. So if bugs are found in developing one iteration, the source of the bugs is likely (hopefully) in the code that was introduced since the previous iteration.

The life program that you have was developed in 6 iterations, and you have the source code for each iteration: (`lifeIeration1.py`, `lifeIteration2.py`, …, `lifeIteration6.py`).

Here are descriptions of the features introduced in each iteration:

1. Display the game window with two frames: the grid frame on top, and the control frame on the bottom. Don’t put any widgets into the frames; just make the top frame red, and the bottom frame blue. Testing can be done visually.
2. Put widgets in the grid frame. The widgets here will be the cells of the grid.
   Testing can be done visually.
3. Put widgets in the control frame. These will be the start, pause, step, reset and quit buttons; and the step speed slider. Testing can be done visually.
4. Connect events in the view class widgets to stub event handler functions in the controller class. These stub event handler functions should simply print a statement indicating that they have been called. For the event where the user clicks on a grid cell, the stub function should print the row and column of the cell that was clicked on. Testing can be done by clicking on the buttons, sliding the step speed slider, and by clicking on different cells.
5. Adds the model class to the program. It sets up the data structures and logic to track the state of the game. Testing cannot be done visually yet, so unit testing of the model class methods is set up.
6. Fills in the event handlers in the controller class. Testing can be done by running the finished program.

Developing each of these iterations is a reasonably managable programming task, each representing some tens of lines of code. For each iteration, You should review the features that are added by that iteration, and study the new code to see how it adds the features. And of course, run the code for that iteration to see the features in action.