A Conway's Game of Life Python implementation

Federico Malato
Student ID: 7017325
federico.malato@stud.unifi.it

Abstract

This project aims to recreate from scratch a working version of John Conway's Game of Life, using Python, PyQt5 and the Model View Controller architectural pattern. This implementation follows (or at least, it tries) common coding good practices in order to have simple, readable and efficient code. Altough the code implements a good number of features, there's no claim of exhaustiveness with respect to the original version.

1. Introduction

The "Game of Life" is a cellular automaton created by mathematician John Conway at the end of 1960s. It aims to modelize the way in which organic cells work, starting from a number of them and reproducing, staying stable or dying following some rules given by Nature. Such an auto-incremental behavior has been of interest for a wide range of people, and so the game became very popular during 1970s.
The game is a "no players game", meaning that while the game is progressing there's no need of a user at all: after he has chosen the initial state of the cells, then the game begins and updates autonomously.
Conway's Game of Life can be summarized as a grid of white squares that the user can paint in order to determine the initial state. Then, when the simulation starts, the rules of the game determine which squares will be colored when the next generation occurs.

2. Game Rules

Even if it's common to look at very complex patterns, the rules of the game are a few number and they're also quite simple. Given a state of the cells grid, the game progresses by computing the next generation state following four simple rules:

- if a living cell has less than 2 neighbors, it dies from loneliness;
- if a living cell has more than 3 neighbors, it dies from overpopulation;
- if a living cell has exactly 2 or 3 neighbors, it survives to the next generation;
- if a dead cell has exactly 3 neighbors, it comes to life to the next generation.

3. Setting and previous work

To develop the game I've been using Python 3.7 and PyQt5 5.9.2 on the PyCharm IDE by JetBrains [1]. The project has been developed mainly on MacOS Catalina 10.15.0 but has also been tested on Windows 10.
Moreover, to structure a first, basic implementation of the game I've been learning the idea of [2] and then editing it in order to fit my idea.

4. Project Structure

The project has been built around the idea that the cells could be seen as a grid of 0s and 1s, where a 0 in a given position stands for a dead cell, while a 1 means that the cell in that position is alive. Starting from this point, there have been a lot of improvements in order to reduce computation as much as possible. For example, as the Game of Life requires to iterate over a bidimensional array, I've implemented a list, called *coloredPositions*, containing the (row, col) positions of each living cell, from which computing the smallest area of the grid useful for the current generation and hence speeding up the computation when updating the state.

4.1. Code Structure

The program is composed by a single MainWindow (inheriting from QtPy's *QMainWindow*) that contains a bunch of objects that inherits from some *QWidgets*:

- *StartButton, StopButton, StepButton, ClearButton:* they all inherits from *QPushButton* and override the mousePressButton() event in a unique way;

- *FPSSlider:* this Widget inherits from *QSlider* and it is used to control the framerate of the simulation;
- *KnownPatternsBox:* a widget used to load predefined patterns into the cell grid. It inherits from *QComboBox*;
- *HistoryCheckBox:* a simple checkbox inheriting from *QCheckBox* that enables/disables the cell history view;
- *CanvasView:* the main component of the GUI. It inherits from *QLabel*, contains and interacts with a *QPixmap* to draw the cells grid.

There are also a few of other widgets that implement the Model View Controller pattern, which is described below.

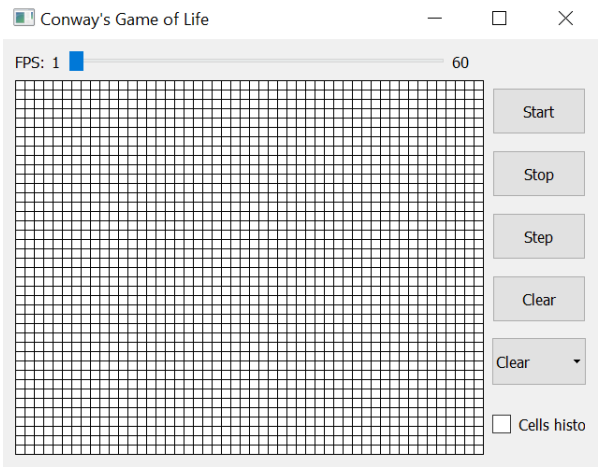

Figure 1. A View of the GUI (on Windows).

4.2. Model View Controller

Model View Controller (MVC) is an architectural pattern which goal is to separate the logic and the appearance of the program. It uses three main components:

- **Model**: the Model component is responsible for data manipulation, meaning that there's no way to access the application data without referencing to it. Moreover, the Model is the only component that performs computations. It can be seen as the "brain" of the application;
- **View**: a View is a component whose responsibilities are to update the GUI when the model's data changes and to fetch the user's inputs and dispatch them to the Controller;
- **Controller**: the Controller mediates between a View and the Model. It is responsible for calling the Model's methods based on the events fetched by the View and, when the computation for the current generation has ended, to tell the View that it's time to update the GUI.

In the project the Model and the Controller are implemented as classes, while the Views are represented by CanvasView and KnownPatternsBox. At first, it could seem obvious to implement the MVC making the MainWindow the only View of the paradigm but, since there are a lot of widgets that don't interact with the data at all, I thought that it would have been better to limit the MVC to those components. To explain my point I'm using an example: I've been thinking of the MVC as a "restaurant situation", in which the Views are the **Customers** of a restaurant, the Controller is represented by a **Waiter** and the Model is the **Chef** of that restaurant. Each **Customer** knows what the **Chef** can cook by looking at the menu, then interacts with a **Waiter** when ordering, and the **Waiter** passes the order to the **Chef**, which cooks the meal and then calls back the **Waiter** to serve it to the **Customer**. At this point, I thought that there could be an old **Friend** of one of the **Customers**, who sees his friend from the window and (maybe being a bit rude, though) enters just to greet him. He politely ask if he's allowed to do that to the **Waiter**, which kindly agrees. Now, the **Friend** had an interaction with the **Waiter**, but it doesn't care at all to bother the **Chef** and has nothing to do with the restaurant, so it can't be called **Customer**!
This is the reason why I decided to limit the MVC to some Widgets: there are Widgets, like the StartButton, that barely interacts with the program and also doesn't care of the data at all. For them, there's no need to become Views, altough they need to know who the Controller is.

4.3. GUI

As can be seen from Figure 1 the GUI is clear and simple. It features four buttons, a combobox from which the user can select a pattern and a checkbox to enable the "Cells history view". In order to be interesting, the grid has been set at 40 rows and 50 columns. When a square in the grid is clicked, it becomes green and, if it is pressed again, it is cleaned. With just this interaction (or alternatively by selecting a pattern from the combo box) the user can set a initial state of the grid, from where the simulation starts. If the "Cells history" checkbox is ticked, at each generation the pro-

gram will paint the squares of the previous generation with a red brush.

5. Known issues and conclusion

The main known issue is the one that comes with the size of the grid: adding more rows and columns can drastically increase the computational time for each generation. This is due to the fact that the grid is a bidimensional array and hence it requires $O(n * m)$ to be updated, being n the number of rows and m the number of columns. This problem is addessed with the *coloredPositions* list and the minimum useful rectangle method, altough they only mitigate the effect when the useful rectangle doesn't grow too much: for instance, a "Glider" pattern won't be affected at all since it leaves the minimum rectangle constant at each iteration, while the "Gosper Glider Gun" will slow down anyway on wide grids.
As future work, it would surely be interesting to find a feasible solution for this issue and to implement an infinite grid.

References

[1] *PyCharm reference page*
`https://www.jetbrains.com/pycharm/`

[2] *Coding Challenge: the Game of Life*
`https://www.youtube.com/watch?v=FWSR_7kZuYg`