ENSAE Paris

C++ project report

Simulation of the Conway's Game of life with GUI

Lucas Degeorge, Baptiste Bupré, Valentin Tordjman–Levavasseur

February, 3rd 2023

Contents

Introduction

The study of cellular automata and its applications in computer science have been a topic of interest for many years. One of the most famous examples of such a system is the Conway's Game of Life. The Game of Life is a cellular automaton that was first introduced by John Horton Conway in 1970. The game is based on simple rules, yet it produces complex behavior in a two-dimensional grid of cells.

In this project, we have implemented a simulation of the Game of Life. The simulation allows users to see the behavior of cells in the game and to interact with the game by changing the state of cells in the grid. The simulation was implemented using computer science principles and algorithms, including data structures and mathematical models.

The report is structured as follows: In the next section, we provide an overview on how to run the Game of Life. In the following section, we present the rules and implementation of the Game of Life, including the algorithms and data structures used. The graphical user interface is then explained in the next section, detailing the choice of using Direct2D and explaining the global structure of the interface. The following sections focus on the initialization of the Game of Life, through initial configurations and also using an Excel file reader, that allows the user to create initial configurations at their own will. Eventually, we list the key binds that can be used to play the Game of Life in an Appendix.

Besides, being a creative game that creates fascinating behaviours, the implementation of the Game of Life simulation also provides a valuable contribution to the field of computer science and serves as a platform for further research in the area of cellular automata.

1 Program's operation

1.1 How to run the program

The project is composed of different files. First, an executable file named `Game of life.exe` contains the simulation. One could open it. A window containing will open. In the bottom right-hand corner, the current generation and the name of the initial configuration can be read. The rest of the window is the grid containing the cells. The first pattern displayed is the blinker. Other patterns are available (see section 4.3)

Several keyboard inputs have been defined to use this executable file. One could navigate through the grid with the up, down, left and right arrow keys. One could zoom in and out with the + and - keys of the numpad. A library of outstanding patterns and initial configurations is available. See section 4.3 and the appendix in section 5 to find out the corresponding keyboard inputs.

The project also contains all these files:

- `Winmain.cpp` : the "main" file of the project.
- `Living_cells.cpp` and `Living_cells.hpp` : files managing the back-end part of the simulation. See section 2.2 for more details.
- `Gamerules.cpp` and `Gamerules.hpp` : files managing the rules of Game of life. See section 2 for more details.
- `basewin.h` : header file managing the graphic user interface. See section 3.2.1 for more details.
- The folder `Excel_reader` contains the files managing the Excel file reader module. See section 4 for more details.

1.2 How to add a new configuration with xls file

The user could choose the initial configuration she desires. A list of several configurations is available. See the section 4.3 and the appendix in section 5 for more details.

The user could also define her own initial configuration. One may follow the following steps to do so.

- Open the xls file named "example1.xls".
- Select the worksheet named "Custom".
- Tape a 1 in each living cell (must be an integer or a double).
- Save and close the xls file. Warning: the file must be saved as a xls file (excel 97 - 2003 workbook) and not as a xlsx or xlsm file.
- Open the application (the EXE file) and press the keyboard input "Shift key".
- The custom initial configuration is then loaded.

Details about how the Excel file is read could be found in the section 4.

2 Rules and implementation of the Game of life.

2.1 Gamerules class

In order to implement and to have the possibility to modify the rules of the Game of life, we have defined the class `Gamerules`. The attributes of that are:

- `m_automatic` : boolean, which default value is `false`. If it is false, the user needs to press space to move from one generation to the next one.
- `m_Bordeless` : boolean, which default value is `true`. If it is true, the game's grid does not have any limit. The version with borders is not implemented.
- `m_diag_count` : boolean, which default value is `true`. If it is true, the diagonals cells are considered as neighbors.
- `m_neighbor_to_die_l` : integer, which default value is 2.
- `m_neighbor_to_die_h` : integer, which default value is 4.
- `m_neighbor_to_born` : integer, which default value is 3.

The last three parameters set the number n of neighbors a cell must have in it neighborhood to stay alive, born or die. If $n \in [\mathrm{m\_neighbor\_to\_die\_l}, \mathrm{m\_neighbor\_to\_die\_l}[$, a living cell will stay alive, if $n \notin [\mathrm{m\_neighbor\_to\_die\_l}, \mathrm{m\_neighbor\_to\_die\_l}[$, a living cell will die and if $n = \mathrm{m\_neighbor\_to\_born}$, a turned-off cell will be born.

Here is an illustration of the rules :

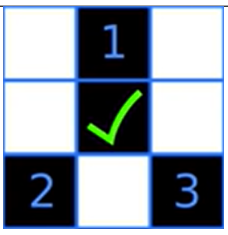
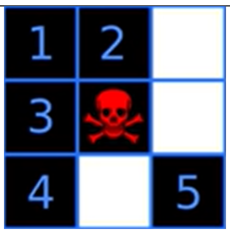
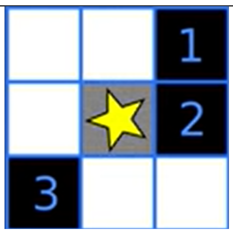
		
Will be still alive	Will die	Will be born

Table 1: The rules

One could change all those parameters (except for `m_automatic`) thanks to the `#define` lines in the beginning of the file `Gamerules.cpp`. However, the initial configuration given within the Excel file example1.xls are only outstanding with the default settings and rules.

In addition to those attributes, the class `Gamerules` has several methods allowing to get the value of a specific attribute.

2.2 Living_cells class

This class includes all the methods and parameters necessary for the simulation of the Game of life. It first includes an attribute called `m_rules` which is a pointer toward a `Gamerules` object. It also has an integer attribute `m_generation` which value correspond to the current generation of the simulation.

2.2.1 The grid and the cells

The first idea we had at the beginning of this project was to define a matrix that embodies the grid where the cells live. However, this solution has several disadvantages.

- First, it would have required to scan the entire matrix during each evolution. The complexity of the simulation is then at least O(d^2) (with d the dimension of the matrix, that could become huge).
- As the major part of this matrix will be empty (containing the value 0 or -1), this solution is very costly in terms of spatial complexity.
- If a living cell moves outside the dimensions of the matrix, it would be necessary to resize the matrix and define new coordinates for each cell/elements of the matrix. Again, this operation will be very costly in time and space complexity.

We have therefore opted for another implementation. The grid is embodied by a vector of pairs (its type is `std::vector<std::pair<int,int»` in C++) containing the coordinates of the living cells. As one may notice that each newborn cell must be included in the neighborhood of an already living cell, it is sufficient to retain only living cells. Thus, the class has an attribute of type `std::vector<std::pair<int,int»`, named `m_living`.

2.2.2 Evolution and methods

The main method of the class is `evolve`. It calls on two other methods:

- `count_neighbors : std::pair<int,int>` $\longrightarrow$ `int` : for a given cell (actually a pair of integers), it computes the number of living cells in its neighborhood by scanning the vector `m_living`.
- `potential_new_cells :` $\emptyset \longrightarrow$ `std::vector<std::pair<int,int»` : it returns a vector of pairs containing all the cells that could possibly be a living cell at the next generation, by scanning the neighborhood of each living (so by scanning the vector `m_living`).

The method `evolve` first runs through `m_living` and determines the living cells which will stay alive. Then, it runs through the potential new cells and find out which ones will be born.

Concerning the constructors, the class has four constructors. In particular, the constructor taking an integer as argument (`Living_cells(int worksheet)`) is the one actually used within the simulations. It is used to start a simulation with an initial configuration given by a key. See the section 4 for more details.

For the purpose of this class, a function `belongs` has been defined to if a cell belongs to a vector of cells.

3 Graphical user interface

3.1 Motivation

The choice of using Direct2D and Win32 API to develop a C++ windowed application can offer several advantages compared to other solutions. Firstly, Direct2D and Win32 API are a part of the Windows operating system and have been widely adopted, which makes them highly accessible and easy to use. Besides, there are plenty of well thought tutorials available on the Microsoft website.
Besides, using these technologies ensures that the application is reproducible and runs consistently on different systems. Being available in the standard library, it allowed us to develop a windowed app without further installation.
Furthermore, Direct2D provides significant performance advantages over other graphics APIs, as it is designed to take advantage of hardware acceleration, providing smooth and efficient rendering of 2D graphics, making it ideal for developing high-performance applications.
Additionally, the use of Direct2D and Win32 API leads to better portability of the application as they are part of the Windows platform, which is widely used across many devices.
Overall, the combination of Direct2D and Win32 API provides a stable and reliable foundation for building a simple yet efficient C++ windowed applications.

3.2 Global structure

3.2.1 basewin.h

This code defines a C++ class template called "BaseWindow". This class is used as a base class for creating Windows application windows, and it implements basic functionality for window creation, message handling and event processing. The class is designed to be used as a base class for other window classes, and the derived class is required to provide the window class name and message handling behavior.
The template parameter `DERIVED_TYPE` allows the derived class to be specified when creating an instance of the BaseWindow class. The class uses the Window Procedure mechanism to handle Windows messages sent to the window, and the static method "WindowProc" serves as the Window Procedure for windows created using this class. The method "Create" creates the window and registers the window class if it hasn't been registered. The method "HandleMessage" is left for the derived class to implement, providing the custom message handling behavior for the window.

3.2.2 WinMain.cpp

This code uses the Direct2D and DirectWrite libraries to draw the simulation window and render the text. It also uses the Win32 API to create the window and handle user interactions.
We have created a derived class from BaseWindow, named MainWindow. The class has the following attributes :

- Pointers to the Direct2D and DirectWrite factories
- Pointers to a black and a colored brush
- A pointer to the render target
- A pointer to the text format

- A pointer to a Living_cells instance
- A pair of floats which is the center point at which the user is viewing
- An integer which is the number of cells to devide the window into
- A handle to an icon
- A pointer to a `WCHAR` which is the name of the currently displayed setup
- An integer which is the length of the name of the currently displayed setup

It has the proper methods to create and discard the graphic and text ressources, to draw on the screen, to resize the window, and to handle user inputs.
Our program features a "SafeRelease" function, which is a utility function that releases an object interface. This function checks if the input pointer is not `NULL`, if it is, it calls the Release method on the object and sets the pointer to `NULL`. This is a useful safety feature to ensure that resources are properly released, avoiding memory leaks.

3.3 Detailled methods overview

3.3.1 Constructor

The class constructor uses an initializer list to set the pointers to `NULL`, and the number of cells to 40. Then, it declares a new instance of Living_cells, sets the center point to (0,0), and sets the correct name for the default structure (which is the blinker).

3.3.2 CreateGraphicResources

This method creates the graphic and text resources for the window. At each step at which the memory allocation may fail, the function continues only if it has not. More precisely, it first initiates the graphical resources, which are :

- The Direct2D factory
- The render target
- The brushes

And the DWrite resources:

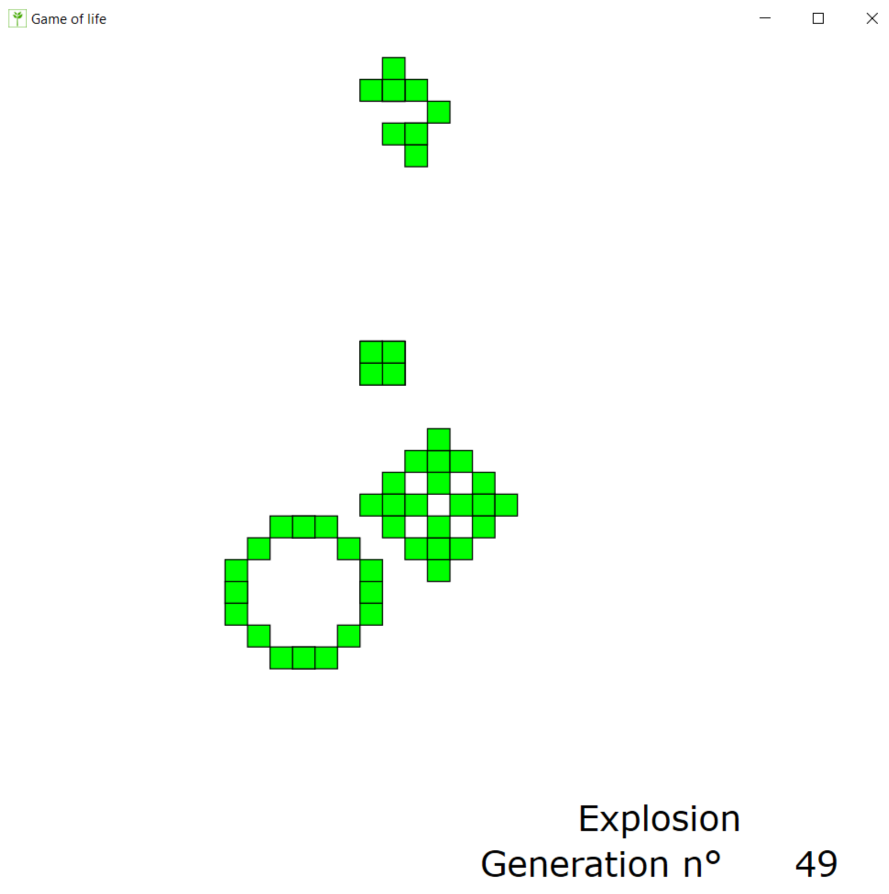
- The text factory
- The text format, which contains what font and fontsize will be used

3.3.3 DiscardGraphicResources

This method calls the SafeRelease function on all the pointers that were initialized in the CreateGraphicResources method.

3.3.4 OnPaint

This method declares a `HDC` (Handle to a Device Context) and a `PAINTSTRUCT` object to begin painting.
After having called GetSize and get_to_frontend, the method fills the rectangles representing the cells with the colored brush, and draws a black contour with the black one.
The methods then displays the iteration number and the name of the chosen setup. To do this, it requests the iteration number from the instance of Living_cells, uses some `WCHAR` manipulations, and finally displays the text in rectangles at the bottom right corner. Here is the result :



Explosion
Generation n°   49

Figure 1: Illustration for the OnPaint method

3.3.5 Resize

This method is called when the user tries to resize the window. It calls GetClientRect to have the requested size, sets it to the window with the Resize method of the render target, and redraws everything with OnPaint.

3.3.6 HandleMessages

This method is here to handle the user inputs.

WM_CREATE This is the create window message. It sets the initial window position and shape for it to be a `1000x1000` square, and initializes the window icon.

WM_DESTROY This is the destructor message. It calls the DiscardGraphicResources method and deletes the pointer to the Living_cells instance to avoid memory leaks.

WM_PAINT This calls the OnPaint method.

WM_KEYDOWN This reacts to the user keyboard inputs. The user inputs can be summed up into a few different ones :

- Inputs to move at where the user looks at, with the up, down, left and right arrow keys
- Inputs to zoom in and out with the + and - keys of the numpad
- Inputs to choose a new setup, see appendix for the list of keys

4 Excel file reader and initial configurations

4.1 Motivation

While focusing on implementing a library of outstanding patterns, it appears that creating a `std::vector<std::pair<int,int»` for the initial configuration of each outstanding patter is quite time-consuming and tedious, as the number of cells may exceed one hundred. However, it was way easier to redeign the configuration or the pattern in an Excel file.

Thus, we have decided to implement a file reading module. This module should be able to load an .xls file and read the cells within the workbook.

4.2 BasicExcel class

In order to read Excel file, we have used the library published on codeproject.com by Yap Chun Wei, an indonesian researcher, named `BasicExcel`. From this labrary, we only use the reading features. This librairy defines three (among others) classes: `BasicExcel`, `BasicExcelWorksheet` and `BasicExcelCell`. From thoses classes and some of their methods, we define a function, named `read_excel` which follows those instructions: it

- Opens and loads the file "example.xls".
- Loads the worksheet with the indice `worksheet` (an integer given as the argument of the function). Actually, it creates a pointer to a `BasicExcelWorksheet` object.
- Runs through the cells of the worksheet (by creating pointers to `BasicExcelCell` instances.
- Gets the value stored within each cell and checks if this value is a 1 (either an integer or a double).
- constructs and returns a vector of pairs (of integers) containing the coordinates of the cells with a value of 1.

This function is declared in the file `Living_cells.hpp` and is defined in the file `Living_cells.cpp`.

4.3 Initial configurations

A library of outstanding patterns and initial configurations is available. One could load one of these patters by pressing the corresponding keyboard (see the appendix in section 5).

Those patterns are divided in three categories:

- The still lives

Those patterns do not change from a generation to another one. Here are some examples:

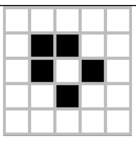
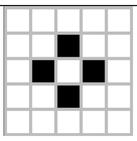
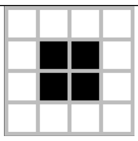
		
The boat (Numpad 3)	The tube (Numpad 4)	The block (Numpad 0)

Table 2: The still lives

- The oscillators

Those patterns return to their initial state after a finite number of generations, the period T. Sometimes, the period could be a large number (more than 150 for some). Here are some examples:

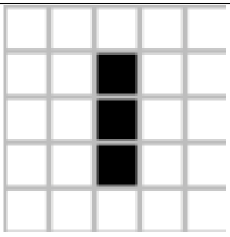
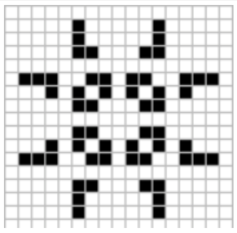
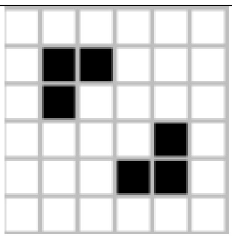
		
The blinker, T=2 (Numpad 2)	The pulsar, T=3 (Numpad 9)	The beacon, T=2 (Numpad 8)

Table 3: The oscillators

- The spaceships

Those patterns which translate themselves across the grid.

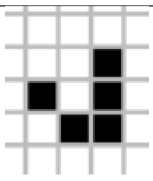
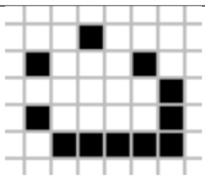
	
The glider (F2)	The middle-weight spaceship (F4)

Table 4: The spaceships

- The glider guns

Those patterns are able to generate an infinite number of gliders. They prove that a finite number of cells could have an infinite evolution. The population of living cells can grow beyond some finite upper limit.

One can find out those patterns thanks to the keyboard inputs F9 and Decimal numpad key.

- Other patterns

Other patterns are available. They have been selected and added because they are outstanding in some way: they are quite chaotic, the expansion of the number of living cells is quite impressive, etc. We let the user find out herself all the patterns.

The user could also create her own pattern. See section 1.2 for more details.

5 Appendix

5.1 List of available keyboard inputs

- Inputs to move at where the user looks at, with the up, down, left and right arrow keys
- Inputs to zoom in and out with the + and - keys of the numpad
- Inputs to choose a new setup :
  - Num pad 0 : Block
  - Num pad 1 : Beehive
  - Num pad 2 : Loaf
  - Num pad 3 : Boat
  - Num pad 4 : Tub
  - Num pad 5 : Cthulhu
  - Num pad 6 : Blinker
  - Num pad 7 : Toad
  - Num pad 8 : Beacon
  - Num pad 9 : Pulsar
  - F1 : Penta-decathlon
  - F2 : Glider
  - F3 : LWSS
  - F4 : MWSS
  - F5 : HWSS
  - F6 : Canada goose
  - F7 : 60P5H2V0
  - F8 : Smoker
  - F9 : Gosper glider gun
  - F11 : 1 Hive to 4
  - F12 : Stairs
  - Decimal num pad key : Simkin glider gun
  - Multiply num pad key : Explosion
  - Divide num pad key : Expansion
  - Shift key : Custom configuration