Tutorial to CellAutoCpp

Cellular Automata Library implemented in C++

By Tongfei Guo, Xiaotian Hu, Haomin Long

What is it

CellAutoCpp is a library that helps users to define and play **cellular automata** with customizable rules.

A cellular automaton consists of a regular grid of cells, each in one of a finite number of states, such as on and off. For each cell, a set of cells called its neighborhood is defined relative to the specified cell. An initial state (time t = 0) is selected by assigning a state for each cell. A new generation is created (advancing t by 1), according to some fixed rule (generally, a mathematical function) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood. Typically, the rule for updating the state of cells is the same for each cell and does not change over time, and is applied to the whole grid simultaneously.

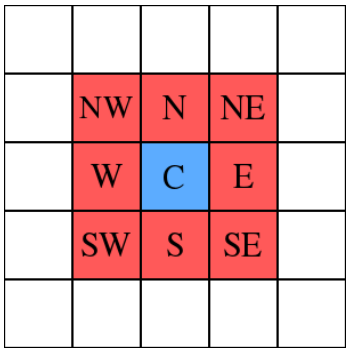


Figure 1 Moore neighborhood

The library of CellAutoCpp uses C++ 14 features which brings benefit to not only readability and flexibility of of user-defined code, but also performance of the generated program.

Basic Logic

The most basic class of the library is `Cell` (defined in `Cell.h`). A `Cell` object stores not only its intrinsic properties, such as x-y coordinates, but also its type of cell and value of states. Besides, every type of `Cell` has its own rules for initialization and processing and all such rules are stored in `Cell` class's static member.

Inherited from `Cell`, `CellHistBounded` and `CellHistUnbounded` are two classes that maintain history information about cells. The difference between these two derived classes is that, `CellHistBounded` only records a limited number of historical copies of the cell and deletes any outdated one, while `CellHistUnbounded` maintain the whole history of the cell. With these two classes, we can easily access any moment of the world and, more importantly, we can introduce history information into the rule of processing.

The world is implemented and stored in `CAWorld` class, with the help of Cell class. A `CAWorld` object maintains a two dimensional array of pointers to Cell objects. For base class and derived classes of `Cell`, the ways of storage and initialization vary but the interfaces are the same for all member functions of `CAWorld`.

To run your own cellular automata, first you need to determine types of `Cells` (like woods and fire). Then for each type of Cell, you need to provide three functions: an initialization function, a reset function and a process function. It is optional to provide a get-color function that returns an index of color given type of `Cell` and `Cell` object's specific states (like woods and fire, as different types of Cell, have different sets of colors; and each wood has its own color given its prosperity).

After setting up all information about the world (not only functions mentioned above, but also other basic information like width, height and depth of history storage), `CAWorld` class provides manipulation including:

1. Do one round of simulation for all cells and the key logic is:

```
For each cell c do
       Find the initialization function I of c's cell type
       Call I(c)
Done
For a number of times do
       For each cell c do
              Find the reset function R of c's cell type
              Call R(c)
       Done
       For each cell c do
              Find the process function P of c's cell type
              Call P(c)
       Done
Done
```

2. Do one round of simulation for a specific cell;
3. Get a/all snapshot(s) of the world at a specific moment (depending on the property of cell class you've chosen, you might be able to access a distant history or only the most current one);
4. Add some measurements to the world so that additional statistics will be recorded during simulations;

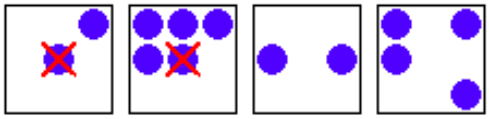
5. Combine two (possibly different) worlds together; (this is useful when you try to simulate water in a cave that is generated using another set of rules)
6. Print snapshots to file or reload them from file.

A Game of Life Example

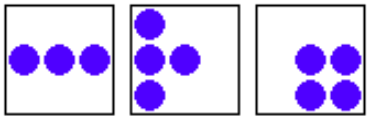
The Game of Life, also known as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. [1]

The setting of the Game of Life is a two-dimensional grid of cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight Moore neighbors. At each step in time, the following transitions occur:

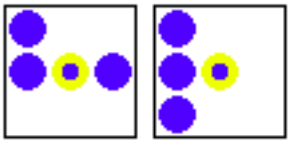
1. Any alive cell with fewer than two live neighbors dies (as if caused by under-population); any live cell with more than three live neighbors dies (as if by overpopulation).



2. Any live cell with two or three live neighbors lives on to the next generation.



3. Any dead cell with exactly three live neighbors becomes a live cell (as if by reproduction).



The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously. The rules continue to be applied repeatedly to create further generations.

To implement a very simple Game of Life example using CellAutoCpp library, we will need to implement four basic functions that are used in the simulation of the game:

1. An initialization function that is called on each cell, before start of the game:

```
auto init = init_type(
    [](Cell *self)
    {
        (*self)["alive"] = (rand()>RAND_MAX/2);
    }
);
```

, where `self` is the pointer to the cell upon which this function is called. Here we randomly assign the alive-or-dead status with ½ probability. Notice that this is also the place where we declare and initialize all states of a cell.

2. A reset function that is called on each cell, before every simulation round:

```
auto reset = reset_type(
    [](Cell *self)
    {
        (*self)["wasAlive"]=(*self)["alive"];
    }
);
```

, where `self` means the same thing as in the initialization function. The purpose of this function is to enable users to store information before a new round of simulation and possibly retrieve them in the processing stage.

3. A process function that is called on each cell in each round, implementing the rules of Game of Life:

```
auto process = process_type(
    [] (const grid_type &grid, Cell *self)
    {
        auto neighbors = get_neighbors(grid, self->x, self->y);
        auto cnt=countSurroundingCellsWithValue(neighbors, "wasAlive");
        (*self)["alive"]=(cnt==3 || (cnt==2 && (*self)["alive"]));
    }
);
```

. The variable `grid` is a reference to the two-dimensional grid where pointers to all cells are stored row by row, column by column. Notice that the x and y coordinates of a cell can be accessed directly, as `Cell`'s class members. We then use a helper function `countSurroundingCellsWithValue(neighbors, state_name)` to count the number of neighbors which satisfy certain constraint. (For more helper functions please refer to the library document.). Finally, we write down the simple rule to update alive-or-dead status, using statistics we just get from current cell's neighbors.

4. A get-color function that is called to determine the index of color in the palette:

```
auto getcolor = getcolor_type(
    [](Cell *self)
    {
       return (*self)["alive"]==0)?1:0;
    }
);
```

. This function is used for output purpose, to determine the color of current cell given its cell type and all states it currently has. The library's rendering part then uses the color in a palette with such an index. The palette is defined as follows:

```
std::vector<bitcolor> palette = {
    {68, 36, 52, 255},
    {255, 255, 255, 255}
};
```

, where the members of four-tuples are Red, Green, Blue and Alpha.

With all four functions, we can build a Game of Life model with certain size:

```
CAWorld world(
  Model(
    world_param_type(96, 64, 6),
    {grid_param_type("living", 100, process, reset, init)},
    1,
    getcolor
  )
);
```

. This world is of size 96 by 64, of grid size 6 and contains only one kind of cell ("living") which evolves with the rules we defined.

Now we can do a lot of things to this world:

1. Run it a certain amount of rounds:

```
world.forall_step(42);
world.forall_step(1000000000);
```

2. Add measurement (such as distribution of state values) to the world:

```
world.AddMeasure(new CADistributionMeasure());
world.forall_step(10007);
for(auto m: world.GetMeasures()){
    std::cout<<m->GetName()+": "+m->Str_All();
    std::cout<<std::endl;
}
```

. Each added measurement will be run on each cell in each round to record useful statistical information about the whole world. At any time (possibly at the

middle of game) we can obtain the stringified result of a measurement. A possible output of it is like:

```
> Cell Type = “living” :
       State Name = alive : ("0"=5880), ("1"=264)
       State Name = wasAlive : ("0"=5880), ("1"=264)
```

3. Add measurement to the world and run it immediately:

```
world.AddMeasureAndRun(
   new CADistributionMeasure("Final Dist")
);
```

How to run it

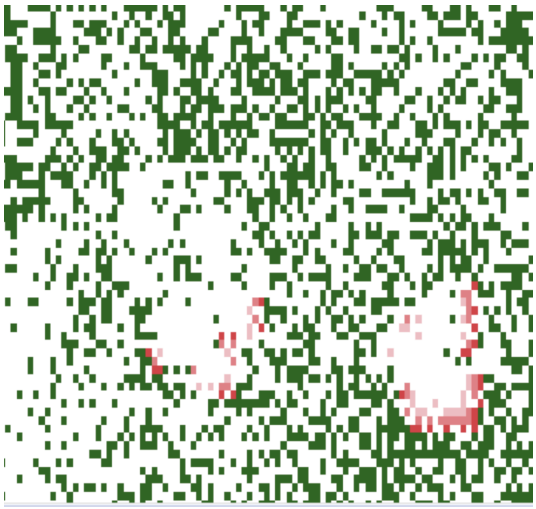
To use library, follow steps:

1. Download repository from: https://github.com/Tongfei-Guo/CellAutoCpp;
2. Feel free to use it! This library has no external dependency.
3. Example models of cellular automaton are provided in following source files:
    a. `cave.cpp` (a model that generates a cave-like map);
    b. `cyclic.cpp` (cyclic color repeating pattern);
    c. `forestfire.cpp` (a model that portraits the relationship between the occurrence of forest fire and the lives and deaths of forest);
    d. `fractal.cpp` (a fractal generated by rule-146, i.e. a variation of Game of Life model);
    e. `gameoflife.cpp` (the famous Life model)
    f. `IsingRun.cpp` (a mathematical model of ferromagnetism in statistical mechanics)
    g. `lava.cpp` (a lava-like pattern simulated by randomly putting ‘drops’ on the surface of ‘mountain’ and distribute them in a radiant way)
    h. `maze.cpp` (a model that generates four-connected maze)
    i. `splashes.cpp` (a model that simulates the effect of rain drop falling on the surface of a lake)
    j. `watercave.cpp`: possibly the best one here because it demonstrates the generalization of our library (at first a cave is formed; then water was randomly added to the empty cells of the cave and they flow and accumulate upon the gravity)

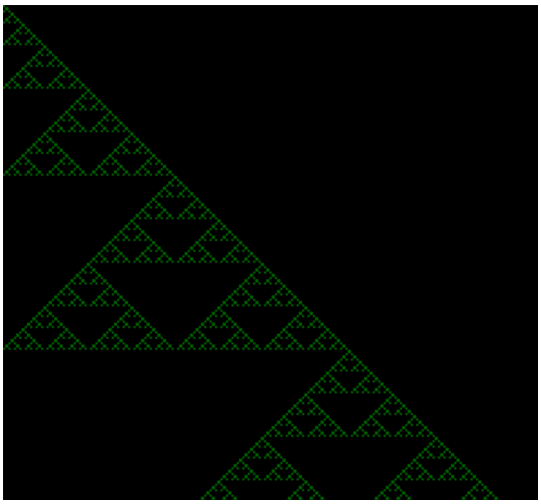
Currently the GUI of CellAutoCpp is available under Ubuntu distribution. To use it, access https://github.com/Tongfei-Guo/CellAutoCpp/tree/visualization.

More examples

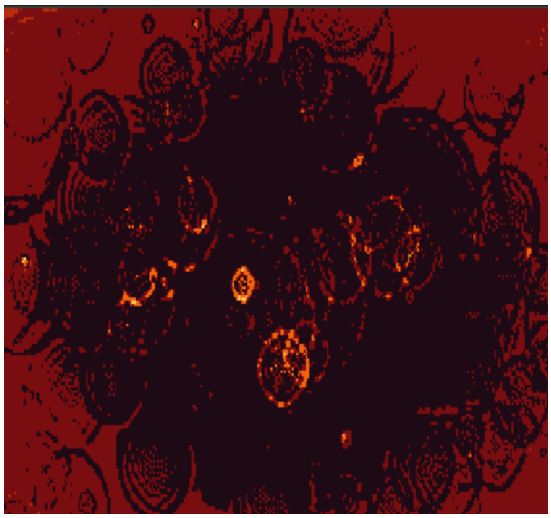
Cycle of Color



Forest and Fire



Fractal



Lava Pattern

Reference

The library is mainly based on the work of rileyjshaw's terra.js and Sanojian's cellauto.

Other references include:

[1] Gardner, Martin (October 1970). "Mathematical Games – The fantastic combinations of John Conway's new solitaire game "life"". Scientific American.

[2] http://web.stanford.edu/~cdebs/GameOfLife/