Cellular:
User documentation

`Cellular` is a (2D square grid) cellular automata simulator. A cellular automaton of this type is a grid of colored cells that change colors in discrete steps. The set of rules by which such changes occur is a part of the automaton and each rule specifies how cells of certain color change depending on the colors of certain surrounding cells. Many interesting properties can be discovered in such structures, but this program only deals with their visual demonstration.

Example: John Conway's Game of Life

The Game of Life is undoubtedly the most famous cellular automaton. It only has two colors, one representing an empty cell, usually demonstrated as a light color and the other representing a populated cell, usually dark.

In a single step, a populated cell only survives (keeps being populated) if two or three of its 8 directly neighboring cells are populated. Otherwise it's said to have died by underpopulation or overpopulation. An empty cell becomes populated if exactly three of its 8 neighbors are populated, as if the cells reproduced.

The author, mathematician John Horton Conway, unfortunately developed symptoms of COVID-19 and died on 11 April 2020 at the age of 82. The Game of Life, which helped develop the field of cellular automata theory, is merely a tiny fraction of his far reaching and fruitful research.

Configuration file format

This program needs to be properly configured to simulate a certain automaton. That's why a dialog asking to choose a file is the first thing that shows after starting it. We will be using a text based format using the suffix `.aut` . In this section, we will look at how the format works.

First of all, `//` will be used for comments - anything following two slashes on the same line is ignored. And let's use those here in the manual as well to indicate where everything will be placed.

Color definitions are the most important parts of the configuration. The following is an outline of how they will look:

```
name #colorHex {
  // rule 1
  // rule 2
  // ...
}
```

So first, we declare a name for our color, then we specify an actual color to be displayed. `#colorHex` is a placeholder for a real color definition in the standard hex format, *or* we can also declare the *RGB* in parentheses, divided by commas, so that for example `#86b300` and `(134, 179, 0)` are equivalent.

Rules are sets of conditions separated by commas in square brackets, that are followed by an arrow (`->`) pointing to the name of the color that should be taken if all the conditions apply. Before we get to conditions, we need to see how we will specify surrounding cells. That's done with coordinates separated by a colon, the first declaring the position to the right from the current cell, the second the position going down. For example `-3:2` is the cell three to the left and two positions down.

There are two types of conditions. First is based on the actual colors in selected surrounding cells. We may look at an example like `(3:1, 2:1, 1:1) = (red, green)` . This condition is satisfied if all three specified neighbors are `red` or `green` .

The second type of condition is concerned with the count of specific colors, so we need to specify both cells and colors that interest us, ie. `count(3:1, 2:1, 1:1)(red, green)` yields the number of `red` and `green` colors among specified squares. Notice that we use two parentheses, first to specify the squares of interest, second to choose the colors. Now we have to use an operator and a number to use that for a condition. `>` , `<` and `=` are legal operators and so `count(3:1, 2:1, 1:1)(red, green) > 1` could be a valid condition of this type.

Game of Life is a good example of the use of the `count` conditions and its configuration can be found as `conway.aut` in the `examples` directory to this project. File `sand.aut` , placed in the same place, uses both count and kind conditions.

To sum up what we have learned, let's think of an example rule we can configure. Say that we have color names `red` , `green` and `blue` and we want a red cell to turn green whenever two of the three squares to its left are blue and the one three to the left is green. Then we could write:

```
red (255, 0, 0) {
  // ..possibly some other rules..
  [ count(-1:-1, -1:0, -1:1)(blue) = 2,
    (-3:0) = (green) ] -> green
  // ..possibly some other rules..
}
```

If conditions to any rule written before this one are satisfied, that rule is followed and latter ones are ignored.

The order in which we write the cell color definitions only matters in that the very first one is going to occupy all squares in the beginning. Additionally, our grid has a fixed size and in the computation of next cell colors, the squares out of grid bounds are taken as having the first specified color at all times.

To avoid glitchy cell behavior, in the specification of surrounding cells, the distance in one direction is limited to 8*, so* `-8:8` *is legal, but* `9:0` *is forbidden.*

Macros

What we've explained so far is all there is to our configuration format and we could stop at that. But for convenience, macros are there to help out with configuration. They are words starting with a capital letter that get rewritten to their substitution. We define them like so:

```
!MACRO_NAME any substitution text we may want
```

We start a new line with an exclamation mark immediately followed by our macro name. Everything further on the line is the macro substitution and any time „MACRO_NAME“ is encountered in the file, it gets rewritten to „any substitution text we may want“.

Multiline substitution texts are possible too. When we put `\\` in a substitution definition, we tell the program the next line is what follows in the substitution. Even macros inside macros are allowed.

All words starting with a capital letter are understood as macros. This means that ***color names need to start with lowercase letters*** *(otherwise they would get confused for macros).*

Just to clarify, cell color and macro names may contain letters, digits and underscores (as long as cell names start with lowercase letters and macro names with uppercase ones).

The visualization

When an `.aut` file is read successfully, we move on to the visualization itself. The window is initially in a 'stopped' state where you can set the colors inside cells. Just pick a color on the right and click on the squares you want to color. The `next` button then lets you see the next state the squares take with the given config. There's a button on the bottom right for switching configurations and finally, the 'play' button on the top right lets you see the visualization in constant motion, with the speed you pick via the slider under the `next` button.