Cellular:
Code documentation

Reading the configuration files

`ConfigurationReader.cs` , that is responsible for loading config files, is a large part of the program. Writing it, an important requirement was that it not only reads correct config files correctly, but also provides valuable error messages for incorrect ones. That made the somewhat easy approach of using regular expressions unfit. Another option would be to use an existing tool for compiler preprocessors / frontends, but that just seemed like overkill. So although this requires a few lines of unsightly hard to structure code, I decided to simply write the reader myself - let's look at how I did it.

First things first, the class takes a `TextReader` as its constructor parameter, which enables the read. The load itself is induced with the `Load` method. The task it really solves is converting the text based config into a `List` of `CellColor` objects. It receives a reference to an instantiated empty `List` as a parameter, and returns a `LoadResult` , which, using its `result` field enables three options: `OK` , `WARN` and `ERROR` . `WARN` and `ERROR` results indicate a `message` associated with them. This version of the project doesn't really utilize WARN results, but further versions may to indicate minor possible mistakes in the configs.

Before we move further and see what `Load` really does, we should state that any method it calls can throw a custom `LoadException` to conveniently indicate an error in the config file. This exception than gets caught by `Load` and turned into an ERROR `LoadResult` . Now about the methods.

The first thing that gets processed are macros. They are read with `ReadMacro` simply getting the names and raw text they substitute to. Nested macros are allowed, so then `PrepareMacroGraph` is called, which reads the substitutions again and searches for words starting with capital letters to substitute for other macros. Those words are removed from the strings and marked inside `nestedMacros` . Then, a modified topsort algorithm is used to both find the topological sort of the macros, so that the correct final substitutions may be found, and detect and report a possible cycle (more in the comments inside the code).

After macros are figured out, `NextChar` is used to read through the file as if it didn't contain macros, but only the preprocessed cell color configurations. It does so by setting `currentChar` and `iterateLine` (indexed from zero) to be able to point out exact line numbers where errors occured.

The rest is fairly straight forward. Cell color configurations are read sequentially, first into raw versions of `CellColor` , then turned into actual `CellColors` that reference each other.

Rules and the computation of next cell colors

As stated in the user manual, rules are sets of conditions that, when satisfied, lead to a certain switch in a cells color. Code-wise, we have `Condition` to represent the conditions and also detect if they were met. Every rule has a set of surrounding cells its conditions refer to. Their colors get fetched before evaluating the conditions, and so `Condition.ConditionMet` may only look up colors via a list provided by its `Rule` .

This explains how a change to a single square is computed. All changes to the whole grid are computed in parallel inside `CellMapController` , by slicing its rows into `Environment.ProcessorCount` subsets of approximately equal sizes and using the built in thread pool (`Tasks`) to process those.

It turns out that redrawing the simulation Panel itself in parallel could be desirable to avoid flashy refreshes, but if I'm not mistaken, `Winforms` don't allow parallelization for updating the UI.