CLLAUS – THE CELLULAR AUTOMATA SIMULATOR

PROGRAMMER'S GUIDE

VERSION 1.0

INTRODUCTION

Cellular automaton is a discrete model consisting of a regular grid of cells, which can be any finite number of dimensions. Time is also discrete, thus the state of a cell at time *t* is a function of the states of finite number of cells (called its neighbourhood) at time *t - 1*. The *neighbourhood* is a constant selection of cell positions relative to specified cell (which either might or might not be in its own neighbourhood). Automaton evolves to its next generation once the *transitional function* is simultaneously applied to each cell in the grid.

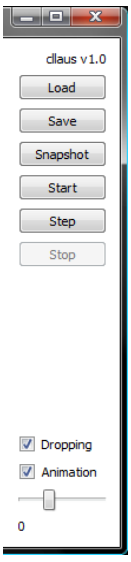
Conway's Game of Life is a special cellular automaton described by mathematician John Horton Conway in Scientific American (October 1970). The universe of the Game of Life is an infinite two-dimensional rectangular grid of square cells, each of which is in one of two possible states, *live* or *dead*. The neighbourhood of each cell consists of 8 cells, which are directly horizontally, vertically and diagonally adjacent. Transitional function between two generations is application of two simple rules. First rule says that any live cell with two or three live cells in its neighbourhood survives to the next generation or else dies. Second rule says that any dead cell with exactly three live cells in its neighbourhood returns to life in the next generation or else remains dead.

SPECIFICATION

Program *cllaus*, as an acronym for *cellular automata simulator*, currently implements so called Conway's Game of Life automaton. If there would be any future version, there could be more automatons implemented. Primary goal of program is to provide real time resizable view of evolving universe, which can be easily zoomed and scrolled by mouse or keyboard. Program is able to load quite large patterns since it can address over $4 \cdot 10^9$ cells in each dimension, although very large patterns are CPU and memory intensive and the simulation is thus slower. Simulation can be accelerated by turning off universe rendering.

User interface offers some basic features such as simulation control, speed adjustment or file handling. Simulation can be started and stopped. It is also possible to step only one generation forward. Patterns can be loaded from or saved into RLE files, which is at this time only supported file format. In current version it is not possible to interactively edit patterns via user interface. Visible area of the universe can be saved into BMP, PNG or JPG image.

CONTROLS



In order to load pattern into the universe, click on *Load* button and locate the file in opened dialog. Once the pattern is loaded into universe the simulation can be started by clicking on *Start* button. While the simulation is running the speed of evolution can be adjusted by shifting the slider in the bottom right corner to the right (faster) or to the left (slower). Number which is located under the slider represents how many generations were computed until now. To make simulation running even faster it is possible to temporarily turn off animation by *Animation* checkbox. Note that when certain speed limit is exceeded the frame dropping feature takes effect in order to speed up evolution. However if there is very large pattern loaded, due to which the simulation is slow even at maximum speed, frame dropping can be inconvenient. Therefore it can be turned off by *Dropping* checkbox. Evolution can be paused any time by clicking on *Stop* button.

When the simulation is paused current state of universe can be captured into file by clicking on *Save* button. In order to save current view of the universe into image file click on the *Snapshot* button. If you want to step only one generation forward just click on *Step* button while the simulation is not running. Paused simulation can continue to evolve current state of the universe by clicking on *Start* button again. If you want to replace current pattern by some different pattern just click on *Load* button and choose new file to open.

View of the universe can be interactively adjusted by zooming and scrolling. Zoom function is realized by steering mouse wheel or by pressing plus/minus keys. Scroll function is realized by dragging and dropping the mouse or by pressing arrow keys.

DATA STRUCTURES

Due to obvious sparse occurrence of living cells in the universe it is convenient to memorize only living cells. Internal representation of the universe thus consists only of active areas of the universe. This approach saves time and space since empty parts of the universe need not to be computed and stored. The universe is finite and its size is determined by some unsigned ordinal data type. In current implementation the size is bounded by 32 bit unsigned long integer which gives 2^{32} addressable elements in each dimension of the universe. Universe itself is represented by instances of two classes – *World* and *Line*.

Line class stores its vertical address, pointer to the next line and pointer to the array of addresses. Each element in this array represents a horizontal address of living cell in corresponding line. *Line* also provides information about how many cells are allocated and how many of these allocated cells are actually populated. Note that while it is not possible to represent dead cell, line itself can be empty.

World class contains pointers to the first and last line of allocated space. Similarly to *Line* class the *World* also provides information about how many lines are allocated and how many of these allocated lines are actually populated. It also implements mechanism for finding non empty lines. As was mentioned in the previous paragraph there can be some empty lines in the *World*. However there will never be more than two adjacent empty lines since each of these two lines is a boundary line of some active area in the universe. Such a line was added to the *World* because there was possibility that some of its cells would become alive. Thus if the line is empty this possibility didn't arise.

Both of these classes are designed to grow during execution. It is not possible to shrink them. If the next generation of the universe needs more lines or more cells in any line, corresponding data structures are widened. On contrary if the next generation of the universe is smaller than currently allocated space nothing is done. Information about how many allocated elements actually represents current universe can be obtained from amount of populated lines in world and cells in each line.

Transitional function is represented by array of cell states. Since in Conway's Game of Life the neighbourhood consists of the rectangular area of nine cells in which each cell can be either live or dead, it is possible to use index of mentioned array as a representation of the neighbourhood. Thus the size of the array equals $2^9 = 512$ and index into this array is ordinal data type that consists of at least 9 binary digits each representing live (1) or dead (0) cell in the neighbourhood. Cell state stored on particular index in the array determines the state of central cell of neighbourhood in next generation.

ALGORITHMS

Implemented simulation algorithm is one of the effective known algorithms for Conway's Game of Life. It provides efficient space compression and quite fast pass through the universe so it can handle even largest known patterns like *Caterpillar*. Since the algorithm is still fully interactive it is convenient to use it for simulator that provides graphical view of the universe. Because of this fact the algorithm is used in some popular simulators available on the web. However note that there is even faster known algorithm. Its name is *Hash Life* and it provides not only space compression but even time compression. The shame is that *Hash Life* is not suitable for real time drawing, since it evolves various parts of the universe by different speed.

Simulation algorithm handles two instances of the universe. While the current universe is passed through by two nested loops, the universe of next generation is constructed respectively to obtained information. After the successful computation of next generation, both universes are switched and thus prepared for any future generation step.

Outer loop handles the pass through the lines of the universe. Loop maintains pointers to three adjacent lines and detects whether there is a gap represented by series of empty lines. If such a gap is detected, it is skipped in order to speed up execution. After the three adjacent lines are determined the allocation of new line in constructed universe takes place. According to rules of Conway's Game of Life the worst case scenario is that the amount of living cells within the current line in the next generation cannot be greater than sum of living cells over the current and two adjacent lines of current generation. It is possible that constructed universe had already allocated sufficient amount of lines and cells since it was used in some previous generation step. Anyway if constructed universe cannot provide more lines or provided line has insufficient amount of allocated cells, the universe is widened in needed direction. This ensures that inner loop will have enough space to populate new line.

Inner loop handles the pass through the living cells of each line. Since it is needed to determine the neighbourhood of each cell, the loop maintains three cell pointers each belonging to the corresponding line of the outer loop. Pointers are initialized to point to the first living cell of its line. The neighbourhood itself is represented by its map (see *Data structures* section) and address of its rightmost column. Each cycle of the

loop the neighbourhood is shifted at least one address towards the end of the line and its rightmost column is invalidated. Thus the task of the inner loop is to determine this column. It is done by comparing address of the column with the addresses of living cells pointed by pointers. The invariant of the loop is that pointers are either pointing to the same address as the column address or further to the end of the line. Note that even if pointers are aligned under each other from the view of data structure the addresses on which they are pointing can be different. Consequently only those pointers that point to the minimal horizontal address can affect the rightmost column of the neighbourhood. So if the pointer points to the same address as the column address, the corresponding cell in the column is considered alive and the pointer is incremented. Also note that in a case of empty (dead) neighbourhood there is no difference in shifting it only one address and shifting it more addresses towards the end of the line. Thus if such an empty gap is detected the minimal horizontal address of pointers is determined and considered as a new address of rightmost column of the neighbourhood. Since line stores only living cells this can lead to quite large horizontal jump. Once the neighbourhood is completely determined it is looked up in the transitional function table and according to function result the new living cell is added into the new line of the next generation universe. Note that the address of this cell is equal to the decremented address of the rightmost column of the neighbourhood.

FILE FORMATS

Currently only supported file format for loading and saving patterns is RLE. It is the best suitable format for storing large patterns since it uses run length encoding compression. RLE format is widely used and there are many life patterns databases on the web saved in this format. Following paragraph contains the definition of the file format.

There can be any number of comment lines in the beginning of the file. Comment line is started by "**#C**" and ended by platform specific end of line. After the series of comment lines there is one special compulsory line in the form "**x =** *w*, **y =** *h*" where *w* determines width and *h* determines height of the pattern. Rest of the file contains the definition of the pattern ended by exclamation mark "**!**". Each line should be shorter than 70 characters excluding platform specific end of line. Series of dead cells is encoded like "*x***b**", series of live cells is encoded like "*x***o**" and series of line terminators in the universe is encoded like "*x***$**". Value *x* always represents the number of corresponding element in the series. If *x* equals 1 then its value is omitted and only corresponding element character is printed. Note that each line in the universe is ended by some amount of live cells, so there should be no occurrence of "*x***b**" before "*x***$**". The following example shows a simple pattern saved according to definition above.

```
#C two gliders separated by 3 empty lines
x = 3, y = 10
bo$2bo$3o4$bo$2bo$3o!
```

ARCHITECTURE

Program uses open source version of *Qt 4.4.3* framework by *Nokia Corporation*. Framework handles graphical user interface, threading support, event handling and output image file formats. Program itself consists of three main classes communicating with each other – *GUI*, *Engine* and *Life*.

The *GUI* class handles user interface, mouse and keyboard input, resizing of the window, repainting the client's area and file handling. It also stores output bitmap which contains current view of the universe. When the location of the view into the universe is changed by scrolling, zooming, resizing or other user action the bitmap is quickly adjusted by *GUI* itself in order to reflect immediate change. It is obvious that when user scrolls or zooms the bitmap there would be some empty areas or insufficient resolution of the image. This approach however ensures that user interface is highly responsive. While the current bitmap is quickly adjusted the *Engine* class is requested to compute fresh bitmap.

The *Engine* class is in fact a worker thread responsible for heavy computations in background. Worker thread is designed to fall asleep when there is no more work to do. Whether the thread is working or sleeping it can be interrupted by *GUI* which leads to abandoning the current work, fetching new parameters and starting new work. *Engine* contains instance of *Life* class which upon request provides raw bitmap view of the selected part of the universe. The bitmap is raw in a sense that each cell is represented by one pixel and cell cannot be sliced by border in the middle. *Engine* adjusts borders and resolution of this bitmap according to fetched information and sends rendered bitmap to *GUI*. Note that *Engine* is also responsible for frame dropping and speed of the universe evolution.

The *Life* class contains universe data structures and main simulation algorithm described in *Algorithms* and *Data structures* section above. Moreover it provides method for rendering raw bitmap of some part of the universe which is further handled by *Engine*. Class also contains methods for encoding and decoding the universe from/into RLE file format.

EXTENSIONS

In current version program provides only basic functionality and can be extended in many ways. First of all it would be convenient if some future version would support fully interactive client's area so the user would be able to remove or add cells directly into the universe by mouse clicking. There could also be possibility to load more patterns into the universe at once. Another extension would be the support for more pattern file formats like MCL, LIF or PLF. In a wider range of view there could be implemented mechanism for custom transitional function or even completely different approach to simulation like non-rectangular shapes of cells and more cell states represented by colours.

LICENSE