NTNU – Trondheim
Norwegian University of
Science and Technology

Growing Cellular Structures with Substructures Guided by Genetic Algorithms

Using Visualization as Evaluation

Trond Klakken

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

In a research project working within fundamental questions regarding growth and growth processes the grand challenge is to grow a skyscraper. The project is in the area of unconventional computation.

As part of the project growth based on cellular structures is to be investigated. As a preliminary approach multidimensional non-uniform Cellular Automata (CA) is a candidate that includes a possibility to explore cellular rules as the growth process and the growing cellular automata as a physical structure.

In this project the aim is to investigate the possibility to define a 3-dimensional cellular space where building-like structures can grow. This task includes definition of cellular neighborhood and cell states, as to be able to express structures with sought properties. Further, a multi scale approach is to be investigated, i.e. multiple CAs growing together to grow the building structure with secondary sub-structures.

The growing structures are to be evaluated by visualization. The visualization of the growing structure, with sub-structures, should be able to handle visualization at different detail levels. At the basic levels the growth process should be visualized using only the underlying cellular structure. At higher detail level a visualization should be able to represent the growing structure by include graphical elements that resemble actual building materials.

The ultimate goal of this project is to present a demonstrator that illustrates a first attempt to grow a virtual skyscraper.

Assignment given: 9. January 2012
Adviser: Gunnar Tufte, IDI, NTNU

Abstract

A dream about evolvable structures that change to fit its environment could be a peak into the future.

Cellular automata (CA) being a simple discrete model, it has the ability to simulate biology by growing, reproducing and dying. Along with genetic algorithms, they both simulates biological systems that can be used to realize this dream.

In this thesis, a skyscraper is grown using multiple cellular automata. The skyscraper is grown in a CA simulator and visualizer made for this thesis. The result is a stable structure containing floors, walls, windows and ceilings with lights.

Genetic algorithms have been used to grow electrical wiring from a power source in the basement up to power outlets on each floor, powering the lights.

The dream is a house that covers all your needs.

This thesis is a proof of concept, that it is possible to grow a stable skyscraper using a CA with multiple sub-CAs growing lights and electrical wiring inside.

The project is in the area of unconventional computation, done at NTNU Trondheim.

Abstrakt (Abstract in Norwegian)

En drøm om en evolverende struktur som forandrer seg for å passe med miljøet, kan være et syn inn i framtiden.

Cellulære automater (CA) er en enkle diskret modell, som har evnen til å simulere biologi ved å gro, reprodusere og dø. Sammens med genetiske algoritmer, simulerer begge biologiske systemer som kan bli brukt for å realisere drømmen.

I denne avhandlingen vil en skyskraper bli grodd fram ved hjelp av flere cellulære automater. Skyskraperen er grodd fram i en CA simulator og visualiserer lagd for dette prosjektet. Resultatet er en stabil byggning med gluv, vegger, vinduer og tak med lys.

Genetiske algoritmer er brukt for å gro elektriske ledninger fra en strømkilde i kjelleren, opp til strømuttak i hver etasje, for å gi lysene strøm.

Drømmen er et hus som utvikler seg etter dine behov.

Denne avhandlingen er et konseptbevis på at det er mulig å gro stabile byggninger i en CA og ved hjelp av flere del-CAer gro lys og elektrisk anlegg inni veggene.

Dette prosjektet er en del av forskningen på ukonvensjonelle beregning, gjort ved NTNU Trondheim.

Acknowledgements

First of all I would like to thank Gunnar Tufte, my adviser. He gave me inspiration by sharing his work and thoughts, and helped me when I was astray.

Secondly I would like to thank the LWJGL(see 5.2.1-team for making a superb visualization library for Java. To Mark Napier, thanks for making the GLApp extension for LWJGL.

Last, but not least, I would like to thank my father, Edvar Klakken, and Kim Røen for helping me with spell checking and proofreading.

Contents

Problem Description i

Abstract iii

Acknowledgments v

Contents vii

List of Figures xi

Acronyms xiii

Glossary xv

1 Introduction **1**
1.1 Report organization 2
1.1.1 Theory 2
1.1.2 Theory 2
1.1.3 Results 2
1.1.4 Appendices 2

I Theory **3**

2 Cellular automata **5**
2.1 One-dimensional Cellular Automata 5
2.2 Neighborhood 7
2.3 Rules 7
2.3.1 Number of possible rules in a d-dimensional CA 7
2.4 Classification 8
2.4.1 Class 4 9
2.5 Macro CA 9

2.6 Two-dimensional Cellular Automata 9
2.6.1 Game of Life 11
2.7 Three-dimensional Cellular Automata and usage 11
2.7.1 Stopping growth 11

3 Genetic algorithms **13**
3.1 Selection 13
3.2 Reproduction 13
3.2.1 Crossover 13
3.2.2 Mutation 14
3.3 Fitness function 14
3.4 Usage 15
3.5 Criticisms 15

4 Searching **17**
4.1 Euclidean distance 17
4.2 A* search 17
4.2.1 Concept 18
4.2.2 Example 18

II Technology **19**

5 Technology **21**
5.1 Java 21
5.2 Renderer 22
5.2.1 LWJGL 22
5.2.2 GLApp 22

III Results **25**

6 The Cellular Automata **27**
6.1 The simple CA 28
6.2 Making of the cell grid 29
6.3 Rulebook 29
6.3.1 Hash Rules 29
6.3.2 Don't care states 30
6.4 Macro CA 30
6.5 Rule editor 31
6.6 Visualizer 32
6.7 Design choices 33
6.8 States 33

7 The structure **35**
7.1 The Square 36
7.2 Floor 37
7.3 Levels 37
7.3.1 Sub-structures 41
7.4 Rules 42

8 Genetic Algorithms **43**
8.1 First try 43
8.2 Second try 43
8.3 First results 44
8.4 Divide and conquer 45
8.5 Sophisticated fitness function 45
8.6 Activating the lights 46

9 Optimizations **49**
9.1 Optimizations 49
9.2 Hashing 49
9.3 Active cells 49
9.4 Caching 50
9.5 Reversion system for cells 50

10 Conclusions and future work **51**
10.1 Conclusions 51
10.2 Future work 51
10.2.1 A* search in fitness function 51
10.2.2 Add models 51
10.2.3 Light distribution system 52
10.2.4 Ventilation system 52
10.2.5 Extend CA support 52
10.2.6 Power outlets 52
10.2.7 Fill limit of the active cell list 52
10.2.8 Fixing the revision system 52
10.3 The dream 52

References **53**

Appendices **57**

A Source Code **59**
A.1 Genetic Algorithm as a simplified Python script 59
A.2 Java code 61
A.2.1 Running the CA 61

A.2.2 Genetic algorithm for wiring the lights 61

B Rules **77**

B.1 Macro CA rules 77

B.2 Micro CA rules 79

B.3 Rules produced by genetic algorithm 79

List of Figures

2.1 Wolfram's Rule 30, Top row is $t = 0$, displaying one row per time step. Picture is owned by Wolfram Research, Inc.[8] . . . 6
2.2 The neighborhood for c is [4, 2] (range = 1) . . . 7
2.3 Neighborhood of pink cell visualized in the Visualizer(Section 6.6) . . 8
2.4 Macro CA with multiple micro CAs building a skyscraper . . . 10
2.5 A simple CA . . . 10
2.6 The 4 rules of "Game of Life" . . . 11

3.1 Types of reproduction . . . 14
3.2 ST5-33.142.7 antenna developed through evolutionary design. U.S. NASA. (Public domain) . . . 15

4.1 Example of A* search . . . 18

6.1 Two 8 floor skyscraper with lights and electrical wiring. Both stable, multi-scale Cellular Automata (CA) structure grown from two cells. . . 27
6.2 First CA with "Game of Life", with the oscillator pattern "Beacon" . . . 28
6.3 Neighborhood text string from a hash rule . . . 29
6.4 The visual difference between the macro and the micro structure. Camera is in the same position in both pictures. The glass windows have a window ledge in the micro structure. Also, in every ceiling, electrical wiring and lights are installed . . . 31
6.5 Rule editor - On the left a neighborhood is visualized. The center text box represents the next state if the rule is a match. At the top left the description of the rule can be written. The top right corner has a dropdown menu to choose which rulebook to use. On the right side all rules in the rulebook are sorted by "next state". Clicking on a rule here loads the rule into the neighborhood on the left. Legal states are all numbers and x, x being the "don't care"-state . . . 32
6.6 Hiding uninteresting cells when working . . . 33
6.7 Cell states used in the final structure . . . 34

7.1 Two skyscrapers growing 35
7.2 Making too generic rules may end with unwanted structures 37
7.3 Growing a floor using 3 cell types over 11 generations 38
7.4 Growing levels/floors. Figure continues in 7.5 39
7.5 Continue growing levels/floors. 40
7.6 Difference between active and inactive lights 41
7.7 Electric wiring in a early version. Inactive wires are pink, while active are red 42

8.1 First results 44
8.2 GA getting stuck at the first outlet. Red is the "snake", blue is the outlets and the gray being the Euclidean distance 45
8.3 Inactive wiring over inactive lights, electrical outlet as blue 47
8.4 The final light configuration, all lights connected to the power outlet (blue). Numbers corresponding with Figure 8.5 48
8.5 The final rules for the light wiring, best of 1000 solutions. Number corresponds with Figure 8.4 48

A.1 Command for starting visualizer on linux 61
A.2 Command for starting visualizer on Mac 61

Acronyms

CA Cellular Automata. 5–7, 9, 11, 17, 22, 27–31, 33, 36, 43–46, 49, 51, 52

GA Genetic Algorithm. 13–15, 43–46, 50

JIT Just in Time. 21

Glossary

chromosomes Structure for storing genetic information. 13, 14

Java Popular programming language used in this thesis. 21

Chapter 1

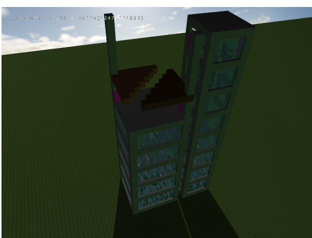
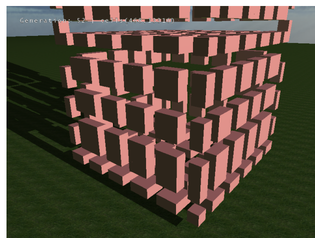
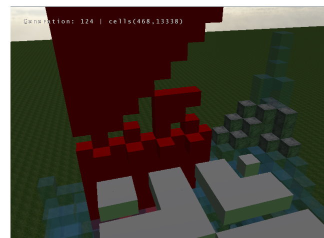
Introduction

The field of Cellular Automata (CA) has been around for 40 year. Over these years a lot of research has gone into patterns, classifications and growth control. The topic of growth control is in focus in this thesis, as a proof of concept, if it is possible to grow a stable skyscraper structure using a CA with multiple sub-CAs growing lights and electrical wiring inside.

A skyscraper is grown using multiple Cellular Automata. The skyscraper is grown in a CA simulator and visualizer made for this thesis (see Chapter 7). The result is a stable structure containing floors, walls, windows and ceilings with lights.

Cellular Automata (CA, see Chapter 2) is a simple discrete model, modeling a grid of cells. Each cell has a state and are only influenced by their neighbors, creating interesting patterns and structures when viewed over multiple time steps.

CAs can be used to simulate biological functions, such as growing, reproducing and dying. Another type of algorithms simulating biology is Genetic Algorithms (see Chapter 3) selecting, crossing and mutating offspring. In this project, genetic algorithms have been used to grow electrical wiring from a power source in the basement up to power outlets on each floor, powering the lights. See Chapter 8.

1.1 Report organization

The report is divided into 4 parts. Theory, Technology, Results and Appendices.

1.1.1 Theory

- Cellular Automata - What is a CA?
- Genetic Algorithms - How does a genetic algorithm work?
- Searching - Theory behind the search algorithms used

1.1.2 Theory

- Technology - Which technology used in the simulator / visualizer.

1.1.3 Results

- The Cellular Automata - How the CA was implemented
- The structure - How the structure was build and results
- Genetic Algorithms - How the GA was used and the results
- Optimizations - What optimizations were done in the visualizer.
- Conclusions and future work

1.1.4 Appendices

- A - Python code of how the GA works
  How to run the visualizer?
  And Java code for The Genetic Algorithm
- B - Rules - The finished rules used by the visualizer

Part I

Theory

Chapter 2

Cellular automata

A Cellular Automaton[1][2][3][4][5][6] (plural cellular automata, abbreviation CA) is a discrete model found in mathematics, physics, computability theory, theoretical biology and micro-structure modeling. Consisting of a n-dimensional gird of cells, each with a finite number of *states*, where n is a finite number. A cell evolves deterministically in discrete time steps accordingly to the given rule set[4]. The rules can be defined as a mathematical function or a boolean expression, using the cells current states and the states of the nearest neighbors (*the neighborhood*) to determine which rule to apply. A state is a integer in the range 0 to $k-1$, where k is the number of colors[4] (states). The neighborhood is defined by the range r, the number of cells included in the neighborhood each direction. All cells are updated before stepping to the next time step.

Other names used are "cellular spaces", "cellular structures", "homogeneous structures", and "iterative arrays". [4]

CA can be generalized into 3 parameters:

- k = colors / states
- d = dimensions
- r = neighborhood range

2.1 One-dimensional Cellular Automata

Using the same notation as Wolfram in [1], cell i can be denoted as $a_i^{(t)}$, where t is the time step. One-dimensional CA can be seen as a regular uniform lattice (or array) of discrete variables (states)[4].

$$a_i^{(t)} = \boldsymbol{F}[a_{i-r}^{(t-1)}, a_{i-r+1}^{(t-1)}, ..., a_i^{(t-1)}, ..., a_{i+r}^{(t-1)}] \tag{2.1}$$

In Equation 2.1 the arrays of $a_i^{(t-1)}$ and the neighborhood is sent to $\boldsymbol{F}$, the rule function, determining the next state of $a_i^{(t)}$. $\boldsymbol{F}$ can be a boolean expression, mathematical function or any other type of function, as long as it returns the next state.

Called "elementary" CA by Wolfram[1], the simplest type of CA is a binary ($k = 2$ states), one-dimensional ($d = 1$), nearest neighbor ($r = 1$) CA. Because of the limited number of states ($k = 2$) and dimensions ($d = 1$) the maximum number of rules is shown in Equation 2.2, read more in Section 2.3.1

$$k^{k^{(2*r+1)}} = 2^{2^{(2*1+1)}} = 256 \tag{2.2}$$

The rules for these "elementary" CAs was named Rule 0 to Rule 255 by Wolfram[1]. Different characteristics and deeper studied of some of these rules can be found in [1][4][7].

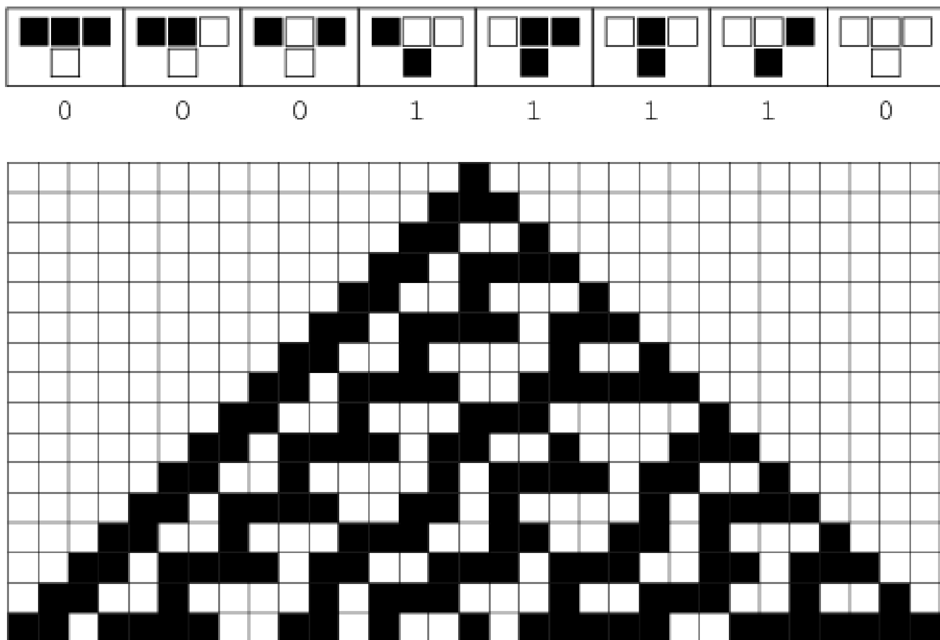


Figure 2.1: Wolfram's Rule 30, Top row is $t = 0$, displaying one row per time step. Picture is owned by Wolfram Research, Inc.[8]

2.2 Neighborhood

The neighborhood of a cell is the surrounding cells in a range r in all directions[1]. In a one-dimensional CA with range 1 the neighborhood would simply be the cell to the left and to the right, as seen in Figure 2.2.

[0, 0, 4, c, 2, 0, 0]

Figure 2.2: The neighborhood for c is [4, 2] (range = 1)

An example of a 3-dimensional neighborhood can be seen in Figure 2.3, where a pink cell is surrounded by cells in another state. The neighborhood includes vertical, horizontal and diagonal cells, creating a neighborhood of 26 cells.

2.3 Rules

CA rules can be boolean expressions, sets or as in Conway's "Game of Life" (Section 2.6.1) mathematical functions. "Don't care" rules can be defined, decreasing the rule space and number of possible rules.

If no rules match the neighborhood, a cell must either remain its current state, or change back to a default state.

2.3.1 Number of possible rules in a d-dimensional CA

In Wolfram's Universality and Complexity in Cellular Automata[1], he presents the Equation (2.2) describing the total number of possible rules for a one-dimensional CA. In need of a d-dimensional equation, the following calculations were made.

Known from statistics, given k possible states, and length x, the total number of possible combinations will be k^x. In the case of a CA the x would be the neighborhood, making it $k^{(2*r+1)}$ for one dimension. $(2 * r + 1)$ describes the width of the line made by the neighborhood. In two dimensions the line becomes a square, and a cube in tree dimensions. $(2 * r + 1)^d$ describes the number of cells in the neighborhood, making $x = k^{(2*r+1)^d}$ the maximum numbers of patterns for a neighborhood. Hence the maximum number of rules becomes k^x, as seen in Equation 2.3.

$$k^{k^{(2*r+1)^d}} \tag{2.3}$$

[1]Vertical, Horizontal, Diagonal - The choice is yours

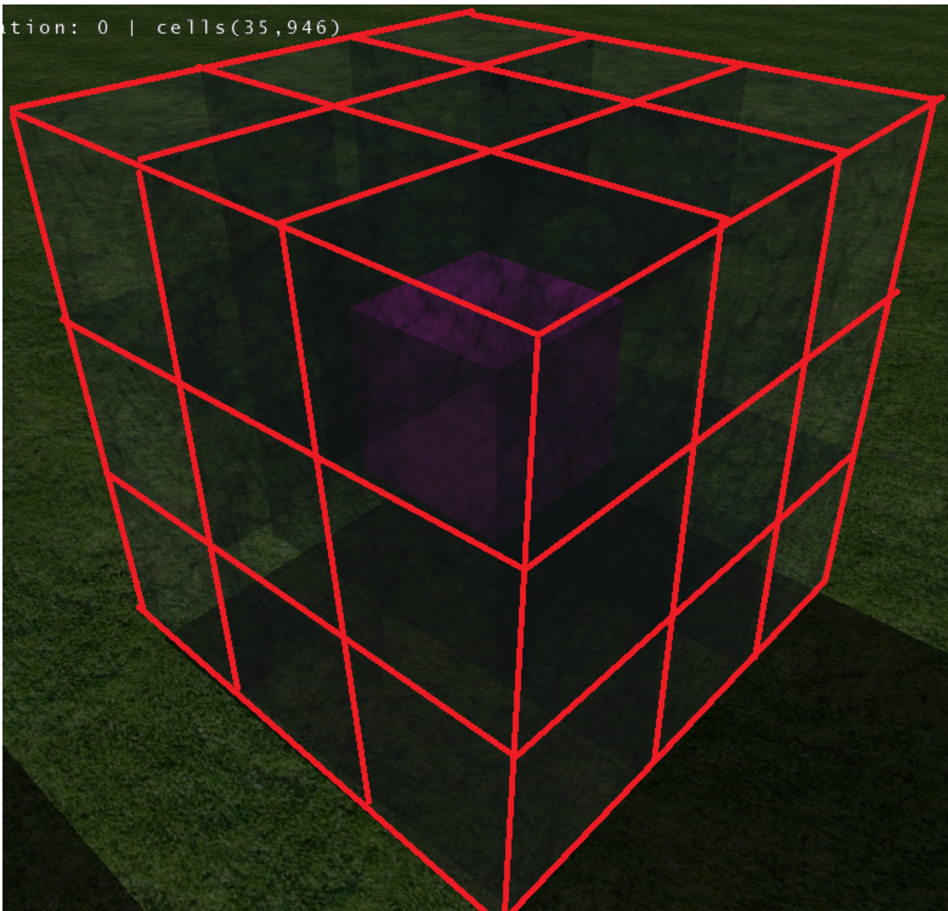


Figure 2.3: Neighborhood of pink cell visualized in the Visualizer(Section 6.6)

2.4 Classification

In "A new kind of science"[5] Stephen Wolfram presented a classification[9][10] for Cellular Automata. Like in biology, classification is a way to organize and placing everything into a system, making further studies easier and more structured. The order of the classifications is sorted by increasing complexity.

Wolfram's 4 classifications as defined by J.S. Hallinan in [9]:

1. Evolution leads to a homogeneous state (fixed point)
2. Evolution leads to a set of separated simple stable or periodic structures
3. Evolution leads to a chaotic pattern

4. Evolution leads to complex localized structures, sometimes long-lived

Other classifications do exist. In [11] 6 classes are described. These are specializations of Wolfram's classes, where class 6 is the same as Wolfram's class 4.

2.4.1 Class 4

Listed last as the most complex, class 4 may also be places between class 2 and 3, ordering by activity levels[5]. Class 4 is where you find the complex structures and most of the CA computations in [1][5][9][10][12][13]. The structure created in Chapter 7 would be placed in this class.

2.5 Macro CA

A macro (from the Greek $\mu\alpha\kappa\rho\acute{o}$ for "big" or "far") CA[14] contains one or multiple micro CA. By diving the CAs into hierarchies, the fine details in the micro CA can be abstracted away, simplifying the problem in the macro CA(s). Complex problems can also be divide into simpler solvable problems, using multiple CA, one or more for each problem.

The use of a macro CA can be read about in Section 6.4.

2.6 Two-dimensional Cellular Automata

Adding an additional dimension, only increases the number of possible rules drastically (see Equation 2.4). All mechanics of the one-dimensional CA applies for the two-dimensional CA.

$$k^{k^{(2*r+1)^d}} = 2^{2^{(2*1+1)^2}} = 1.4078079 * 10^{154} \quad (2.4)$$

Growing in two dimensions can, produce some interesting patterns and effects. Repeating patterns, data storage, CA Machines[15] with processing abilities, traffic[16] and crowd[17] simulation are just some of the possibilities.

In Figure 2.5 a simple 2-dimensional CA time step is shown. The different colors represent different states. The gray cell is the current active cell being compared with the rules. The number beneath each rule is the next state of the cell, if the cell matches the rule. The arrow represents the time step. The active cell does match the first rule, changing the active cell from state 0 to state 1. Whether the rest of the cells will stay unaffected or return to state 0 (or empty), is up to the specific implementation.

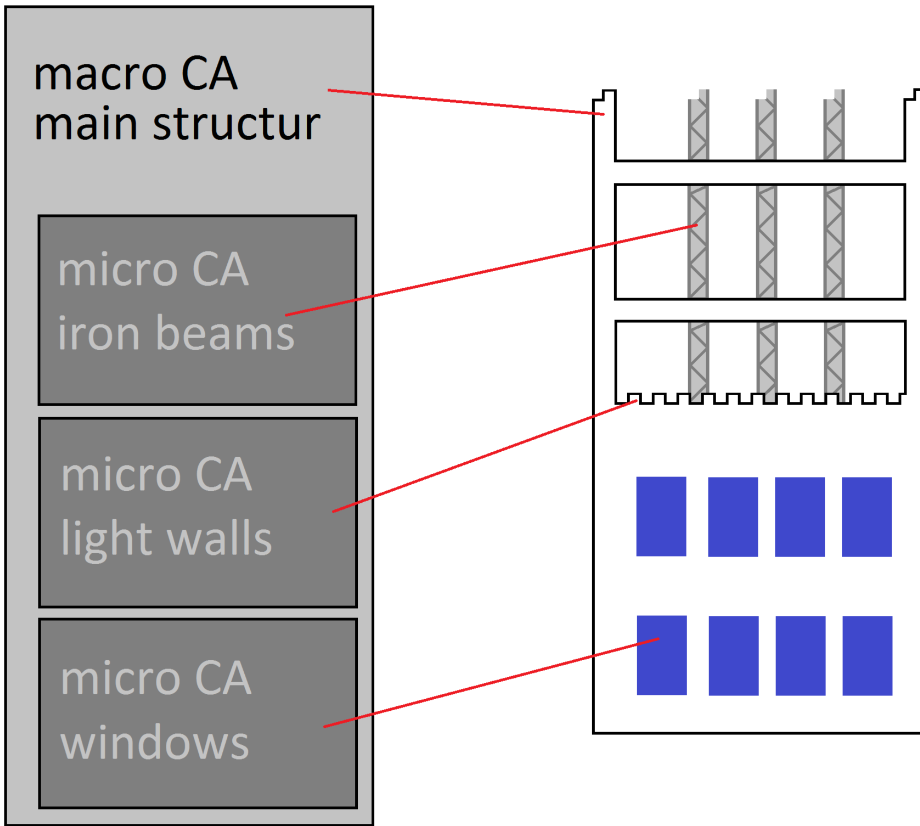


Figure 2.4: Macro CA with multiple micro CAs building a skyscraper

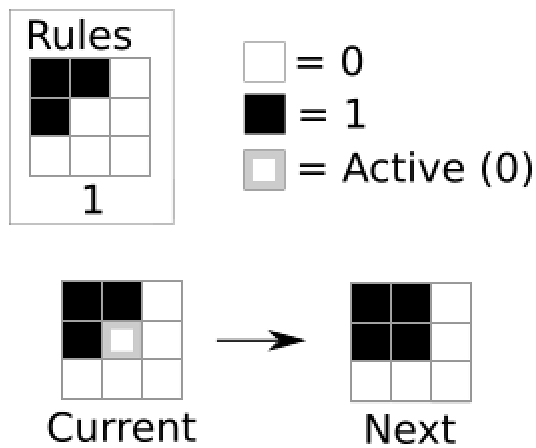


Figure 2.5: A simple CA

2.6.1 Game of Life

John Conway's "Game of life" [18][9] is a binary, two-dimensional, nearest neighbor CA. "Game of Life" simulates organisms reproducing and dying by starving or by overpopulation, where each cells either are alive or dead. The "game" is a study of evolution and self-replication, and the four rules provided in Figure 2.6.

1. Any live cell with fewer than two live neighbors dies, as if caused by starvation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Figure 2.6: The 4 rules of "Game of Life"

Patterns

Interesting patterns were early discovered, making it possible for stable patterns, oscillators and spaceships flying around! On May 18. 2010, Andrew J. Wade announced "Gemini"[19], a self-replication pattern, duplicating itself while destroying its predecessor. Only using 34 million generations, the finding came a decade[20] earlier than expected.

2.7 Three-dimensional Cellular Automata and usage

When adding the third dimension the CA becomes really interesting. Complexity skyrockets and the possibilities unlimited. Growing brain tumors[21], simulating recrystallization[22] or growing a building automatically (as done in Chapter 7), are just some few examples.

2.7.1 Stopping growth

In a nearest neighbor CA, the only way to stop growth is by "colliding" into another cell. Using temporary states the CA can make temporary structures to stop growth or to make new structures not connected to the starting structure. Read more about the use of temporary states and growth stopping in Sections 77.2 and 7.3.

Chapter 3

Genetic algorithms

A Genetic Algorithm (GA)[12][23] is a search heuristic mimicking natural evolution, used to solve optimization and search problems. New generations of *genomes*[24] are created through selection(3.1) and reproduction(3.2). Genomes are candidate solutions tested to find a more optimal solution. The candidate solutions with the highest score are sent to the next generation. Genomes consists of multiple chromosomes, much like chromosomes in an individual's DNA, controlling the behavior of the system.

All genomes are testes against a "fitness function", a function measuring the quality of a genome.

3.1 Selection

During each generation a fitness function selects the "fittest" genomes to succeed to the next generation, much like Darwin's "survival of the fittest". Some algorithms also use random selections; too increase the search scope in the solution space.

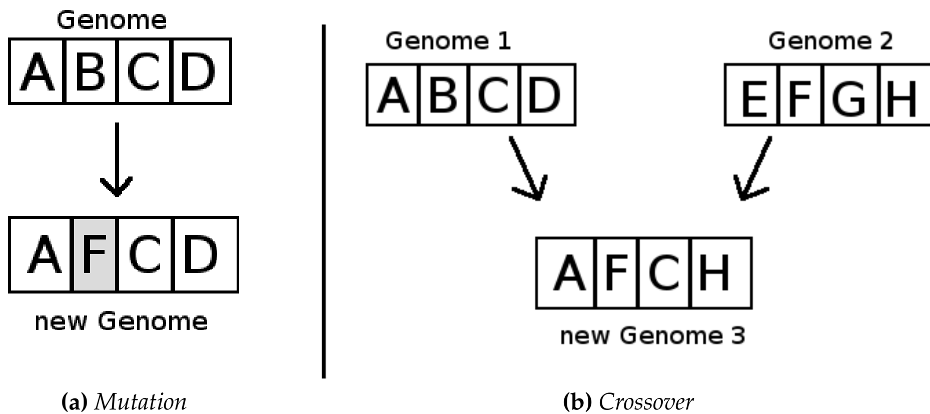
3.2 Reproduction

After a selection is done, a new series of genomes are produced through mutation or crossover.

3.2.1 Crossover

Crossover (Figure 3.1b) is the process of joining chromosomes from two parents. In this case the parents would be two selected genomes from the last genera-

tion. By taking chromosomes from both parents a new and hopefully better, fitter, genome is created.



(a) *Mutation* **(b)** *Crossover*

Figure 3.1: Types of reproduction

3.2.2 Mutation

Mutation (Figure 3.1a) is when taking chromosomes from one parent and randomly changing them.

A crossover can be viewed as a mutation, a mutation of two selected genomes into one. Since the crossover is based on two genomes, the new genome will be closer to its parents in the solution space, than a mutation would be to its parents. While the crossovers function is to move locally, the mutations are used to jump out of local optima in the solution space, trying new and different types of genomes.

Often a mixture of both is required to find a good solution.

3.3 Fitness function

The design of a fitness function may be the hardest part of making a successful GA. Measuring the GAs performance and filtering away "bad" genomes, the fitness functions has to be correct, but often more important, *fast*. Complex problems with 2^{1000} of solutions could take thousands of years finding an optimal solution. However, with a smart fitness function, a sufficiently good solution could be found in a fraction of the time.

Starting wide, the fitness function will gradually narrow the search space as generations go. A bad fitness function can get stuck in local optimal solutions or just perform a tedious search through the *whole* solution space.

3.4 Usage

In 2006 NASA used an evolutionary algorithm to "grow" an antenna design[25] for use in outer space. With complex magnetic fields and no gravitation, a straight forward solution made by a human was not the most optimal. After 4 weeks of evolving the design, NASA had their final design, the ST5-33.142.7 antenna which can be seen in Figure 3.2.

Figure 3.2: ST5-33.142.7 antenna developed through evolutionary design. U.S. NASA. (Public domain)

3.5 Criticisms

GAs are far from the answer to every problem. Over-usage has become a problem later years because of genetic algorithms are viewed as "hip". While it is tempting to go use a genetic algorithm when the number of possible solutions increases exponentially, it's easy to get stuck in local optimal solutions, not optimal for the global solution. Using random mutation can solve this, but it will increase the need for computation as well. Local optimal solutions are created by limitation and presumptions in fitness function to speed up the search. By limiting the search space, a global optimal solution may be lost. However if a non-optimal solution is good enough, a genetic algorithm will always be preferred above brute forcing the solution.

Chapter 4

Searching

A searching algorithm was required to find the shortest path between two cells CA. These are the search algorithms used

4.1 Euclidean distance

In mathematics the Euclidean distance (see Equation 4.1) is the "ordinary" distance between two points given by the Pythagorean formula.

$$d(p,q) = d(q,p) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2} \tag{4.1}$$

4.2 A* search

Pronounced "A star", the A* search[26] is a search algorithm used to find a least-cost[1] path between to points with one or more obstacles. A* is a modified version of Dijkstra's algorithm[27], using heuristics to increase the performance (with respect to time). It is a best-first[2] search, using a distance-plus-cost heuristic function ($f(x)$) to determine the search order for the nodes not yet visited.

The distance-plus-cost heuristic is given by $f(x) = g(x) + h(x)$, where:
$g(x)$ is the path-cost function, the cost from the start node to the current node, and $h(x)$ is the admissible heuristic estimate of the distance to the goal. $h(x)$ must be admissible, in other words not overestimate the distance to the goal.

[1] Cost on all nodes = 1, gives distance
[2] Uses most promising paths first, determined by a rule

4.2.1 Concept

While traversing the graph, the A* search always chooses the path with the lowest *known* cost, keeping a sorting priority queue of alternative sub paths.

4.2.2 Example

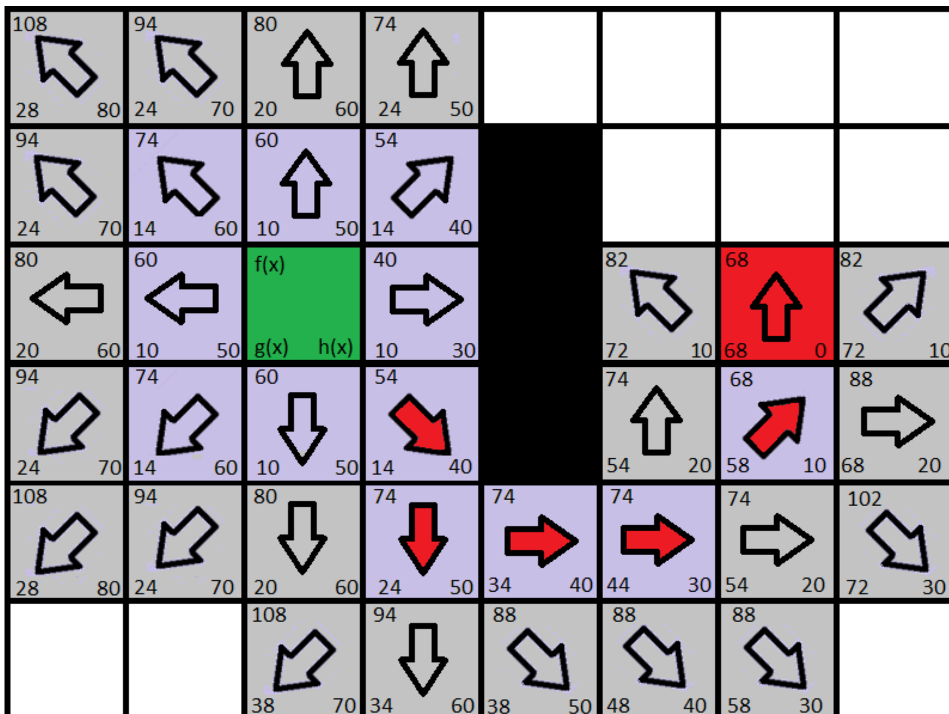


Figure 4.1: Example of A* search

In Figure 4.1 a simple example can be viewed. The green cell is the start position, the red cell is the goal, and the red arrows are the shortest path. The number in the left top corner is f(x), the distance-plus-cost heuristic. In the left bottom corner g(x) is shown and h(x) is shown in the bottom right corner of every cell. Note that the shortest path to each cell diagonally out from the start is the Euclidean distance, making $g(x) = \sqrt{10^2 + 10^2} = 14.142... \approx 14$.

Part II

Technology

Chapter 5

Technology

5.1 Java

Java[28] is a object oriented, multi-paradigm, Just in Time (JIT)-compiled programming language widely used. All code used for making results in this project is written in Java, a programming language supporting:

1. All major operating systems (Windows, Mac OSX and Linux)

2. Fast and easy to build and compile

3. Support heavy graphic operations for viewing 3D models

The reason for choosing Java is more a choice of habits and flavor. Nearly all courses on NTNU focus on Java, and the rest focuses on Python or C. Graphic intense applications are usually written in C++, since it offers a wider variety of libraries and are much faster since it is compiled before used. Java's JIT-compiling does shorten the build time, but increases the work while running the program.

In C++ nearly all memory usage must be allocated and freed when finished, while Java has a automated garbage collection, using even more resources. Another reason for using Java was the portability[1], since I used both Windows and Mac under this thesis. Also supporting OpenGL (Open Graphic Library), one of the computer industry's most widely used graphic library[29].

[1] Portability - support on multiple operating systems

Is Java fast enough for games? The answer is yes. Minecraft[30][2] has a smooth game-play with thousands of blocks simultaneously shown, making Java more than sufficient for this project.

5.2 Renderer

To visualize the CA a renderer has to be used. There are many ways to render[3] a screen image, but only a handful can be done in real time. Since this thesis focuses on growing a skyscraper, a real time environment enhances the experience by giving continues visual feedback, making it possible to inspect every cell from every angle while growing. Voxel renders is often used for discrete data models such as points and boxes. Finding a voxel renderer with good documentation proved to be harder than expected.

5.2.1 LWJGL

LWJGL (LightWeight Java Game Library)[31] offers access to high performance libraries such as OpenGL and OpenCL(Open Computing Library). It is available under the BSD license, which makes it free to use. The documentation for the solution is really good, with multiple example projects. With OpenGL-like function calls, it is easy to use for someone with OpenGL experience. The most used functions send function calls directly to the OpenGL driver, making it fast.

Not being a game engine, like "jMonkey Engine"[32] or "GLApp"[33], it gives you direct access to low level resources and is not cluttered with plug-ins and extra content.

For my usage it was nearly perfect, only missing some few functions. Loading of models, cameras and lighting.

5.2.2 GLApp

GLApp[33] is a small library written by Mark Napier in 2009. It is based on LWJGL and adds cameras, lighting, shadows, models, textures, dynamic fonts and many more functions. It is still lightweight, making it easy to start up. While other alternatives (jMonkey Engine, Jake2 and Ardor3D) have a lot of stuff, GLApp was sufficient for my usage, making it perfect.

Another bonus was good (well good enough) documentation and project examples, making implementation really easy and quick.

In less than a day I had the basic flat terrain and could show the blocks from the CA. Supporting model loading, each cell type could have a 3D-model loaded and

[2] A sandbox video game in Java, made popular in 2010

[3] Converting 3D models and lines into the picture seen on the screen

viewed, showing 130 000 models without any problems. However, the final result only uses blocks with texture. This is because my skills as a 3D artist are horrible. However, with some help the skyscraper would have look magnificent.

Part III

Results

Chapter 6

The Cellular Automata

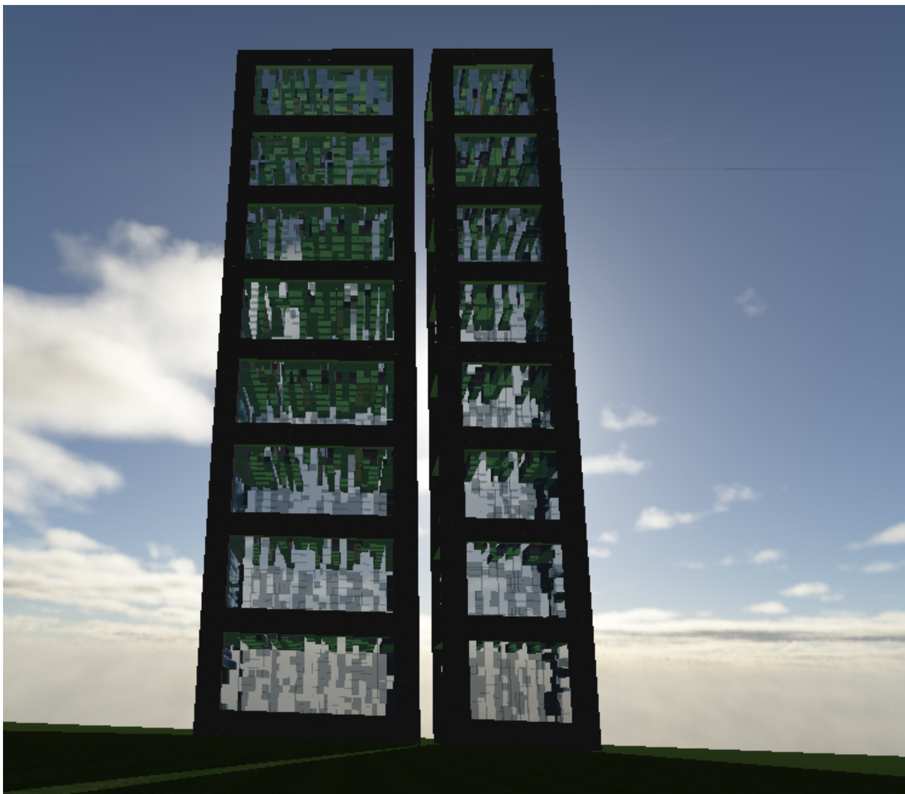


Figure 6.1: Two 8 floor skyscraper with lights and electrical wiring. Both stable, multi-scale CA structure grown from two cells.

This thesis is a proof of concept, that it is possible to grow a stable skyscraper in a CA with multiple sub-CAs growing lights and electrical wiring inside. This is not the most optimal way, nor the faster, but a different approach not tried before (as to my knowledge).

6.1 The simple CA

Early on a simple 2-dimensional CA was made (Figure 6.2, implementing a rule system to simulate "Game of Life"(Section 2.6.1). While being simple, it made the foundations for the final 3-dimensional non-uniform multi-scale version.

z\x	0	1	2	3	4	5	6	7	8	9
0	X	X	X	X	X	X	X	X	X	X
1	X	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	X	X	X
3	X	X	X	0	0	0	0	X	X	X
4	X	X	X	0	1	1	0	X	X	X
5	X	X	X	0	1	0	0	0	0	X
6	X	X	X	0	0	0	0	1	0	X
7	X	X	X	X	X	0	1	1	0	X
8	X	X	X	X	X	0	0	0	0	X
9	X	X	X	X	X	X	X	X	X	X

Figure 6.2: First CA with "Game of Life", with the oscillator pattern "Beacon"

6.2 Making of the cell grid

Although obvious, the CA contains a n-dimensional cell grid. At first the cell grid was a n-dimensional array of Cell objects, in true Object Oriented style, but this had a massive drain of memory and performance. Instead the cell grid was replaced with a n-dimensional Integer array, where the state of the cell was saved, and empty cells where set to Java's "null" value. Strictly speaking, the cell state is the only information needed. However traversing the whole array for each generation is a waste of computation if the array is less than half full. Thus a list of all active cells was added. More on this in the optimization section, in Section 9.1.

6.3 Rulebook

Rulebook is my fancy name for the CAs set of rules. The rulebooks functionality mostly consists of saving, loading, adding and removing rules. It also contains the function getNextState(), discussed in Section 6.3.1.

6.3.1 Hash Rules

A lot of effort was put into the rulebook, and especially the hash rule. The need for lightning fast rule checking drove the result into a simple and fast way to check hundreds of rules towards thousands of cells in just milliseconds.

The "Hash rule" consists of a text string representing the neighborhood, the next state and a description. The neighborhood text string has a variable length, consisting of 27 states separated by a "-" character. The next state is placed between two "@" characters, with a description at the end. An example can be viewed in Figure 6.3.

Under: Plane: Over:
0 1 2 9 10 11 18 19 20 Becomes:
3 4 5 12 13 14 21 22 23 3
6 7 8 15 16 17 24 25 26

0-1-2-3-4-x-6-7-8-9-10-10-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-
@3@Description

Figure 6.3: Neighborhood text string from a hash rule

The neighborhood string consists of 27 states, the 26 neighbor cell's states, and the current cell's state. The x seen in Figure 6.3 is a "don't care" state, meaning the

rulebook will allow any state when checking that particular cell. The format is not human friendly, so a rule editor was made (Section 6.5).

The name "Hash Rule", describes the way the rules are stored and check. By using a HashMap, the next state can be found by looking up the neighborhood string, without search through every rule. However, the HashMap does not support "don't care" strings. A simple Algorithm 6.1 solved this.

Listing 6.1: Function for finding a matching rule for cell

```
1  def getNextState(x, y, z):
2    neighborhoodString = getCellNeighborhoodString(x, y, z)
3
4    if hashRules.has(neighborhoodString):
5      return hashRules.getState(neighborhoodString)
6
7    else:
8      for rule in dontCareHashRules:
9        if rule.matches(neighborhoodString):
10         return rule.nextState
11     # No matching rules, return current state
12     return getCellState(x, y, z)
```

6.3.2 Don't care states

The use of "don't care" states in rule drastically decreases the number of needed rules. If used 10 times in one rule, the rule can match $15^{10} = 576650390625$ neighborhoods (see Equation 6.1). To handle "don't care" states, a regular expression is made along the rule object. Regular expressions are costly to make, but super fast to match against.

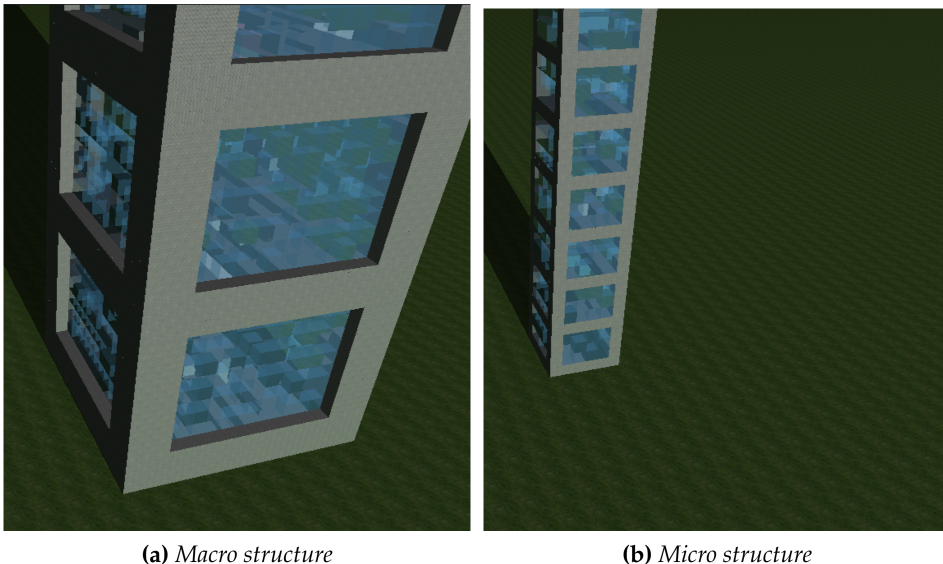
$$\text{total number of states}^{\text{number of don't cares}} \tag{6.1}$$

6.4 Macro CA

Designing a fast multi-scale CA for Object Oriented Java was harder than anticipated. The result was a more C procedure approach. The implementation is good, but the overall design does not support multiple micro CAs (from a Object Oriented perspective).

Only one macro CA and one micro CA is available. However, practically speaking, the micro CA can act like multiple CA if given multiple sets of rules. In other words, the results are as wanted, but a Java architect would cry.

The relationship between the macro CA and the micro CA is $1 : 27$. Each macro cell contains 27 cells (3x3x3) sub-cells controlled by the micro CA. The difference between the macro and the micro structure can be viewed in Figure 6.4 One nifty function is that when a macro cell changes, the corresponding 27 micro cells gets updated to the same state, this way the sub grid is a finer detailed version of the main grid. This makes the structure a dynamic structure able to change drastically if wanted.



(a) *Macro structure* **(b)** *Micro structure*

Figure 6.4: The visual difference between the macro and the micro structure. Camera is in the same position in both pictures. The glass windows have a window ledge in the micro structure. Also, in every ceiling, electrical wiring and lights are installed

6.5 Rule editor

Writing the rules as >54 character strings became a tedious job and led to many sources of error. A rule editor was designed and integrated with the CA and later, the visualizer (Section 6.6). As seen in Figure 6.5, the design is simple, easy to understand and fast to use. Keyboard shortcuts for jumping between cells do exist.

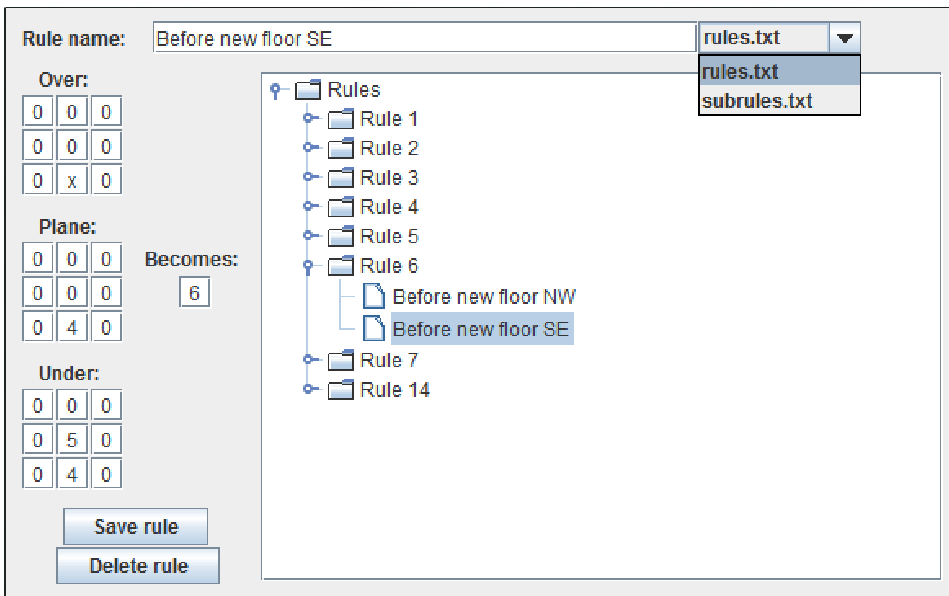


Figure 6.5: Rule editor - On the left a neighborhood is visualized. The center text box represents the next state if the rule is a match. At the top left the description of the rule can be written. The top right corner has a drop-down menu to choose which rulebook to use. On the right side all rules in the rulebook are sorted by "next state". Clicking on a rule here loads the rule into the neighborhood on the left. Legal states are all numbers and x, x being the "don't care"-state

6.6 Visualizer

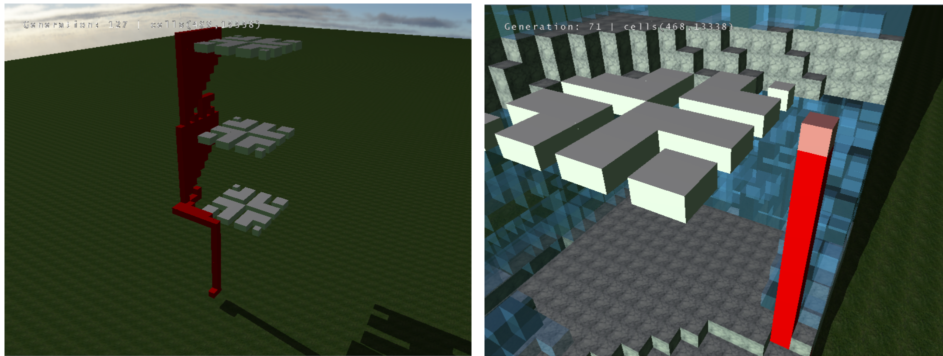
The visualizer is written in Java, using the GLApp(Section 5.2.2) library. Working with GLApp was like a walk in the park, on a sunny warm day. Good documentation, example projects and good coding standard (relatively), everything a programmer could want. The visualizer scale is 1:1 with the cell blocks. Meaning every block is 1 wide, 1 high and 1 deep. This makes debugging a real ease. When started, it loading the texture for every state (Section 6.8). It then loops through the active cells[1] placing them on the grass plane.

The visualizer uses the same rulebook as the rule editor, keeping all rules up to dates. It also has a revision system for each cell, making it possible to jump between generations, a really nice feature when trying out new rules. A generation lock is also implemented, making it easy to jump back to a specific state again and again.

For debugging purposes, it is possible to hide all cells of a given type, either by

[1]See Section 6.2 for the definition of a active cell

pressing a button, or by flying close to the building. This way it is possible to work with the micro CAs cell, while ignoring the macro CA. Both functions are demonstrated in Figure 6.6.



(a) *Hide all macro cells* **(b)** *Hide close macro cells*

Figure 6.6: Hiding uninteresting cells when working

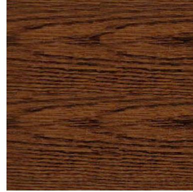
6.7 Design choices

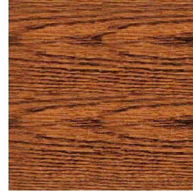
- To minimize computation, I've chosen to set $r = 1$ (see Section 2.2), meaning only nearest neighbors are checked.
- Cells not matching any rules will remain it's current state. Reasons for this choice is discussed in Section 9.1.

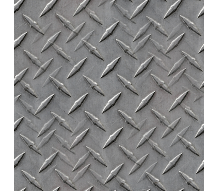
6.8 States

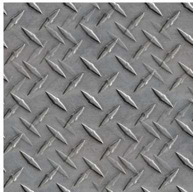
In Figure 6.7 all states used in the final structure are discussed.

(a) *State 0 - The empty state, normally hidden*

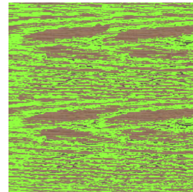

(b) *State 1 - Temp. state used when growing floors*


(c) *State 2 - Temp. state used when growing floors*

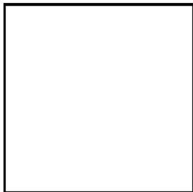

(d) *State 3 - Final floor state*

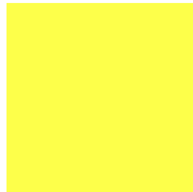

(e) *State 4 - Wall state, often replaced by 14*

(f) *State 5 - Temp. state used to make new levels*


(g) *State 6 - Temp. state used to make new levels*

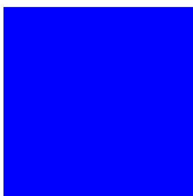
(h) *State 7 - Room state, normally hidden*

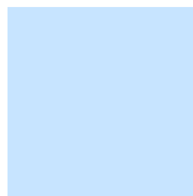

(i) *State 8 - Inactive light state*


(j) *State 9 - Active light state*

(k) *State 10 - Inactive wire state*

(l) *State 11 - Active wire state*


(m) *State 12 - Power outlet state*


(n) *State 14 - Glass window state*

(o) *Missing texture*

Figure 6.7: Cell states used in the final structure

Chapter 7

The structure

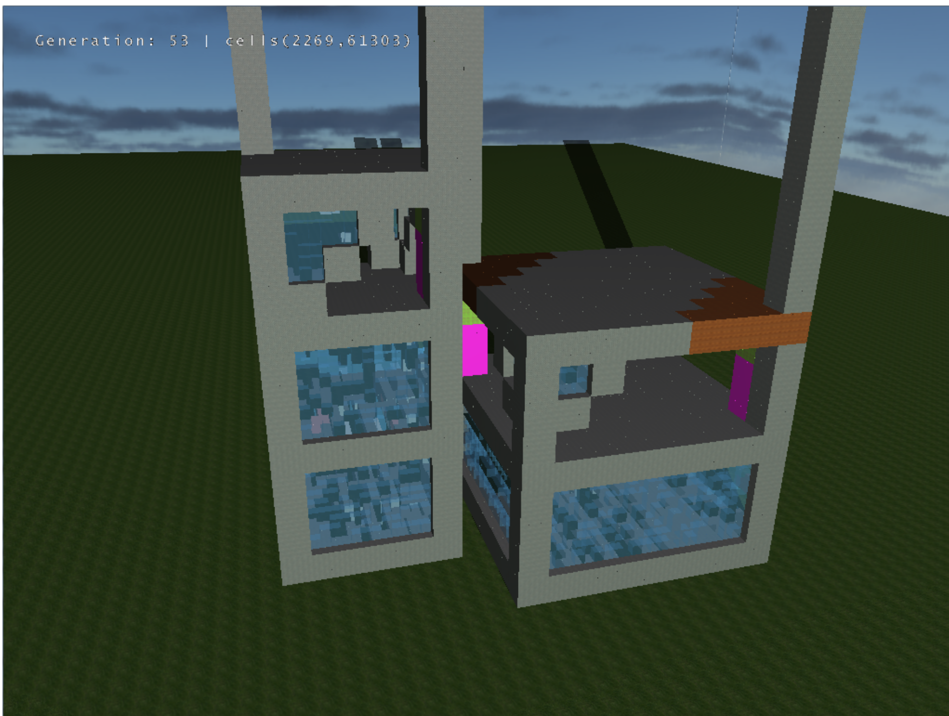


Figure 7.1: Two skyscrapers growing

7.1 The Square

After a lot of coding the first version of the CA was finished and I was finally able to start working on the structure. Using the the Rule Editor (Section 6.5), I played around with lots of concepts. The first one was making a square, representing the ground floor.

Being a simple geometrical figure, consisting of 4 lines, the square is easy to draw. However making one in a CA is some what more complicated.

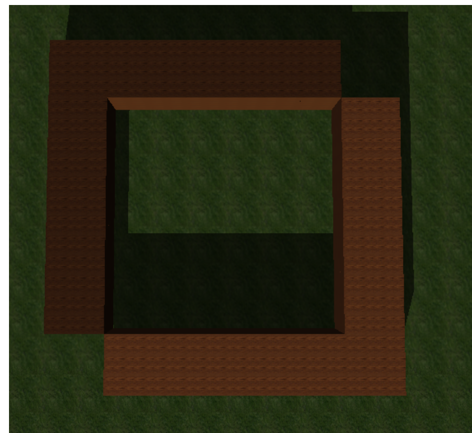
Since the CA only sees it's neighbor, it has no concepts of stopping after n steps. The only way to stop is by "colliding" with another cell.

Starting with two cells in the state 1, the first attempt(Figure 7.2a) had 4 rules.

- Go north
  (if cell state is 0, south cell on same plane is 1. Then become 1)
- Go south
- Go west
- Go east

It did create a square, plus some unwanted lines going out in each direction, as seen in Figre 7.2a.

(a) *First attempt of square*



(b) *Quadratic square consisting of 2 types of cells and 2 rules each. The darker cells in the top left corner is in state 1, while the cells in the bottom right are of state 2*

The problem is, all 4 rules apply for both cells, being the same cell type. Instead, start with two different types of cells. This way you get a perfect quadratic square, as seen in Figure 7.2b.

7.2 Floor

Having a quadratic floor with a big hole in the middle does not make a good floor. To fill the hole, you can use a rule like: "If cell north for current cell is of type 1, current state is 0. Then become state 1". If you take a look at Figure 7.2b again, and try to execute the rule you would find a problem. The cells going downwards at the west border won't stop, as seen in Figure 7.2.

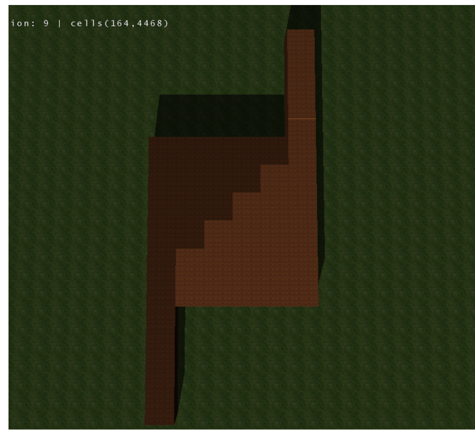


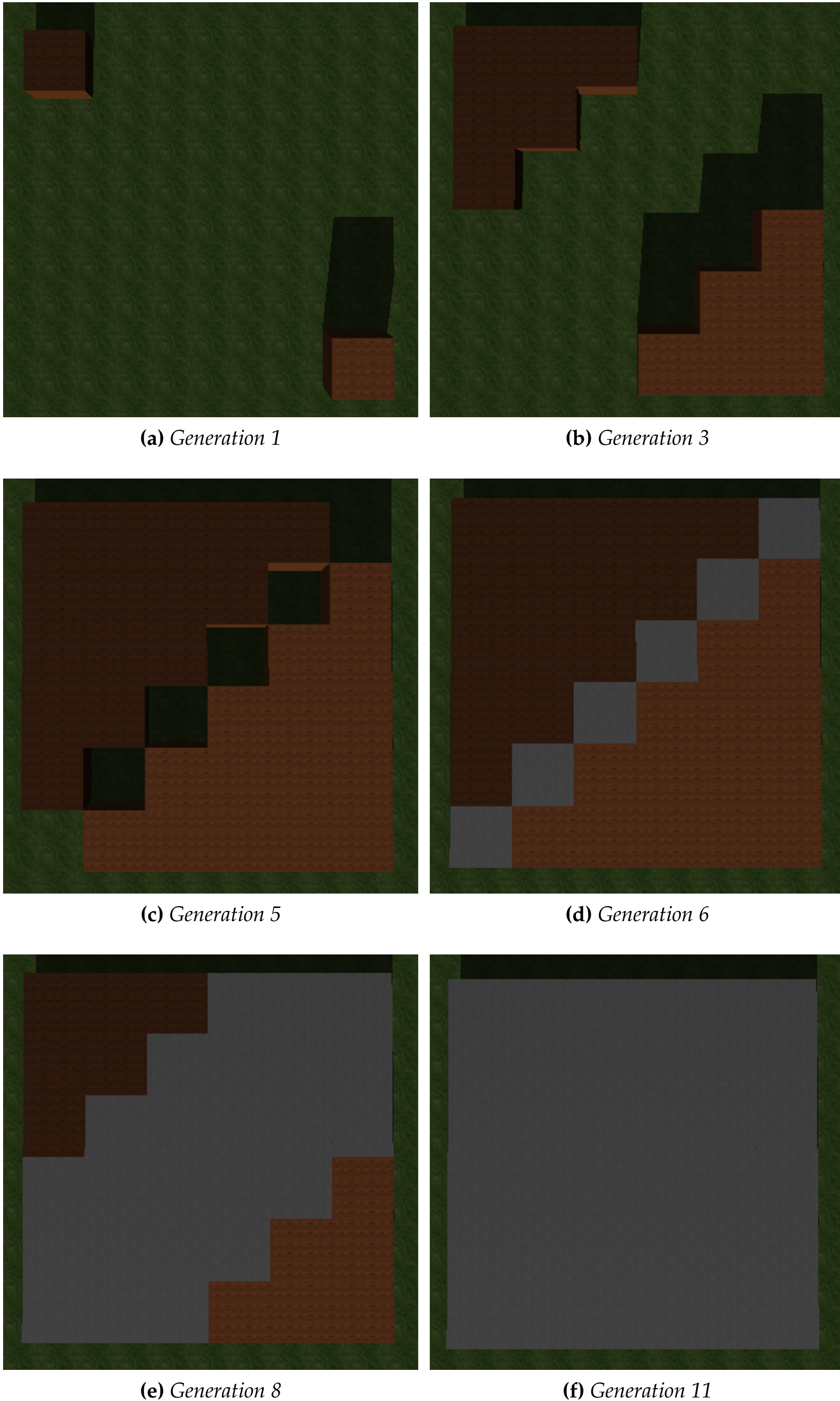
Figure 7.2: Making too generic rules may end with unwanted structures

A floor existing of two different states is not ideal. Instead of making lots of rules to convert one of the states into the other, you simple pick a new state that both of them should become. When both cells meet at the middle, start making the third type of cell, consuming the old cells. Figure 7.3 shows the growth patterns of making the floor, over 11 generations.

7.3 Levels

Starting on the levels (stories / floors) the first thought was sending a vertical diagonal beam, like a support beam, from one corner to the other, while making both corners grow upwards. This would make the level height the same the width, only making quadratic rooms. This solution was there for scraped.

The final chosen growth sequence for levels are shown in Figure 7.4 and 7.5. By constructing a temporary structure, 4 cells high, the growth are limited and a new floor is created.

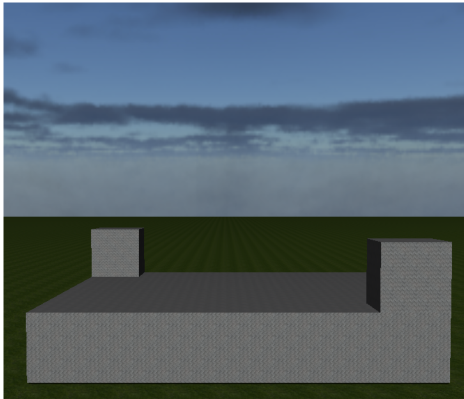


(a) *Generation 1* **(b)** *Generation 3*

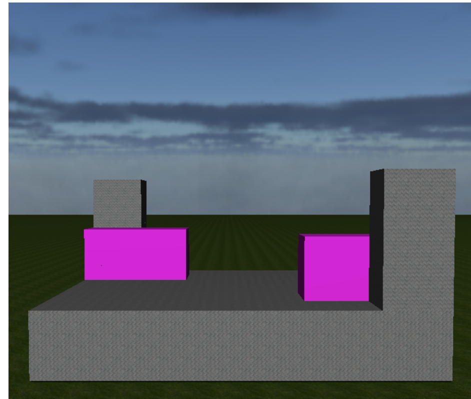
(c) *Generation 5* **(d)** *Generation 6*

(e) *Generation 8* **(f)** *Generation 11*

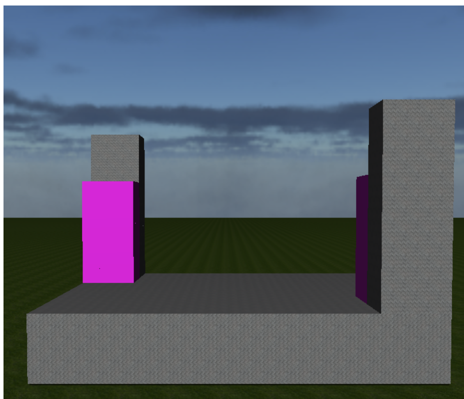
Figure 7.3: Growing a floor using 3 cell types over 11 generations



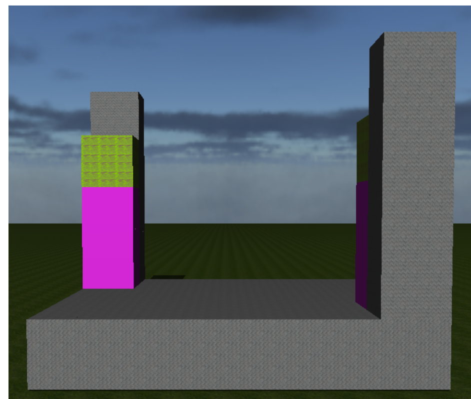
(a) *Start new level*



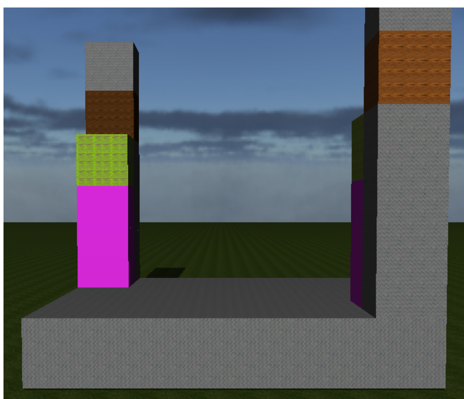
(b) *Make a temp. structure*



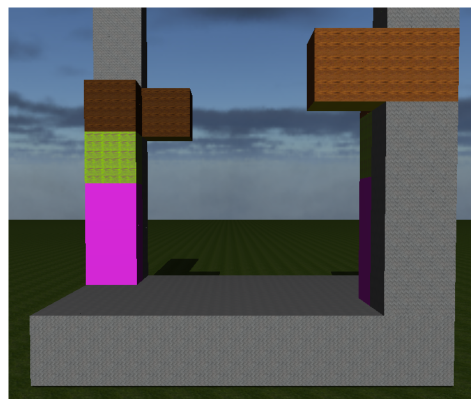
(c) *Continue growing the temp. structure*



(d) *Make a new temp. state to finish the growth*

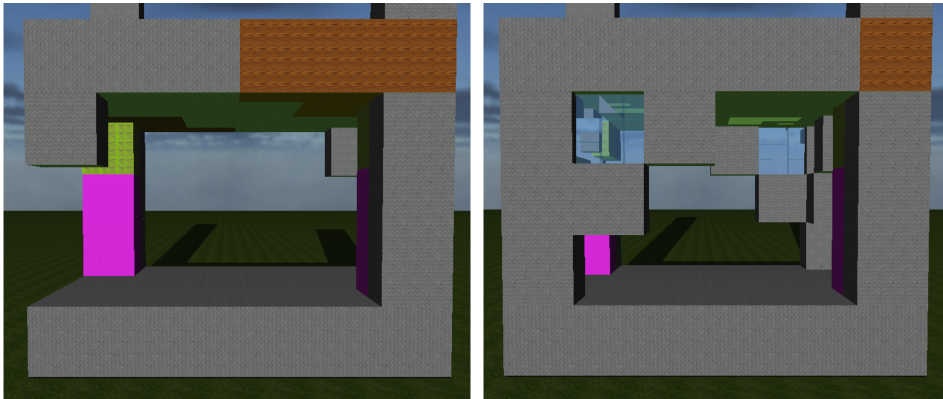


(e) *Start a new floor, just like the first generation of the floor*

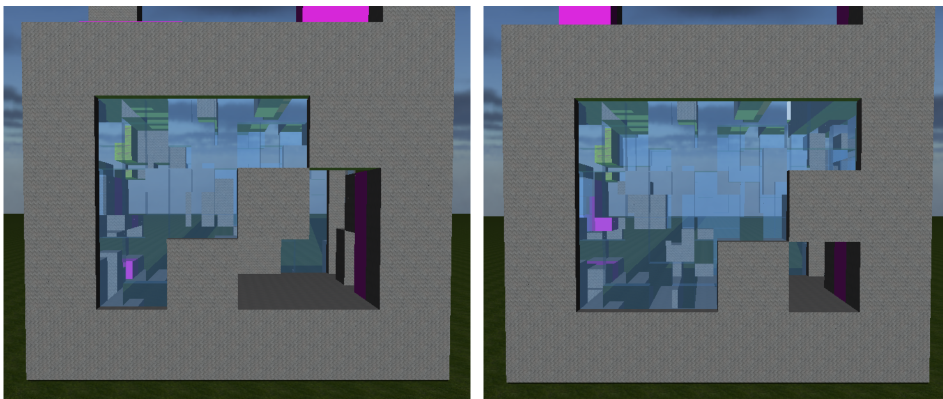


(f) *Floor continues*

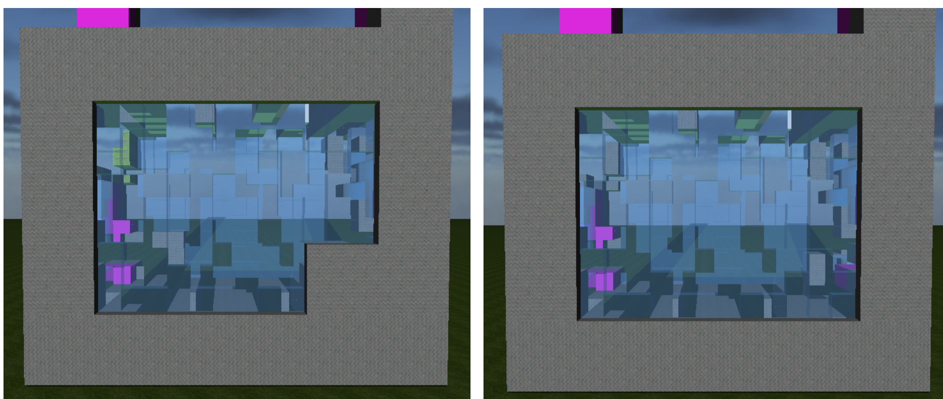
Figure 7.4: Growing levels/floors. Figure continues in 7.5



(a) *Floor finished, start growing walls* **(b)** *Start growing windows*



(c) *Continue growing windows and walls* **(d)** *Continue growing windows and walls*



(e) *Continue growing windows and walls* **(f)** *Finished growing level*

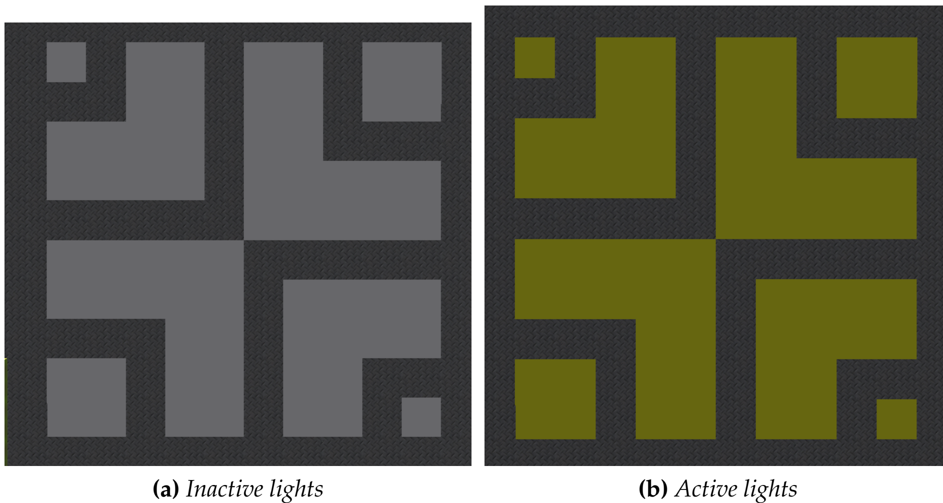
Figure 7.5: Continue growing levels/floors.

7.3.1 Sub-structures

Growing the structure is in it self pretty cool, but it has no practical use alone. Adding additional sub-structures opens new areas of usage. Windows, chairs, desktops, toilets, lights and electrical wiring makes the simulator useful for real life usage. Imaging adding physical rules to the simulating, growing skyscraper instead of a architecture designing them.

Lights

In the roof lights will be grown, first as inactive lights. Lights will then change to an active state if the cell above is a active electrical wire. The difference visually can be seen in Figure 7.6. I did not want the whole roof to consist of lights, so I made a rule creating a "snow crystal"-like pattern. This is because the rule for creating a light: "if cell is of state 3(floor), all neighbors on the same plane is 3, all cells beneath are 7(room), and all over is 3(floor), then become 8 (inactive light)". If you look back at Figure 7.3, the making of the floor, the lights will be inserted in waves, making the pattern.



(a) *Inactive lights* **(b)** *Active lights*

Figure 7.6: Difference between active and inactive lights

Wiring

To simulate electrical wires, a wire is either active or inactive. Inactive lights will only become active when a neighbor, vertically or horizontally, is an active light. Only active wires can power lights. A complex structure of wires can be seen in Figure 7.7.

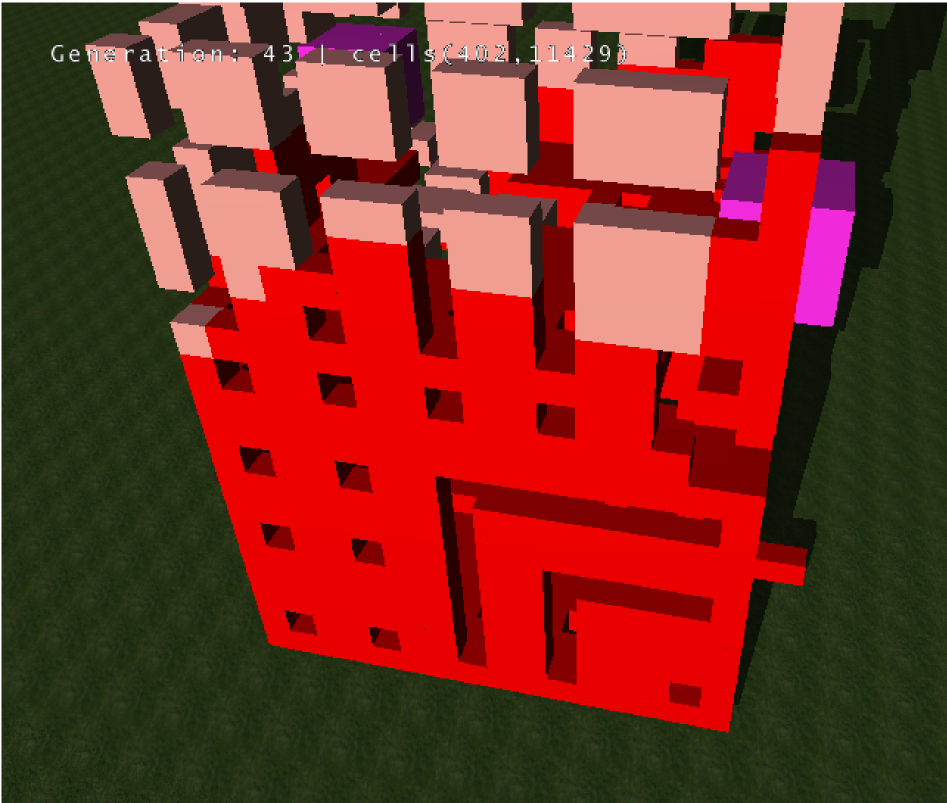


Figure 7.7: Electric wiring in a early version. Inactive wires are pink, while active are red

7.4 Rules

All rules used in the final building is listed in the appendix.

Chapter 8

Genetic Algorithms

With lights in the roof and the ability to add other electrical devices, wiring is needed all over the building. A simple rule for solving this problem would be "if current cell is 3(floor) or 4(wall), and no neighbors are of state 0, 7 (room) or 14 (windows), then become 10 (inactive light)". Solving the problem, it produces more wiring than needed.

A better solution is wanted, making genetic algorithms a fun and interesting choice. CA being evolvable in nature, GA fits nicely making the building more or less an organism. Instead of moving electrical devices to the outlet, a outlet could be grown where it's needed. Just set the device into the wall, and seconds later the device is active.

8.1 First try

Before this project, I had never implemented a GA. I did know the basics about GA and how human cell reproduction worked, but not how to design one. My first try was a really naive implementation where I took a rule, a 27 number array, and more or less brute forced a solution, with a fitness function giving points if a light came active. *Shockingly*, after one hour of calculation, nothing had happened. Calculating the number of possible rules with the Equation 2.3, a worst case scenario would take over 500 000 years. The brute force way of solving was quickly abandoned.

8.2 Second try

Having too many possible rules in my first try, I tried a twist. Waiting until the structure became stable, I looped through all floor and wall cells saving the neigh-

borhood strings in a set. Only choosing neighborhood string containing the wall and floor state, I got all the cells "inside" the walls and floors[1]. What I discovered was that there were only 8 types of neighborhoods. By making a rule for each one of these neighborhoods a worst case solution was found.

I then used mutation on these generic rules, hoping for a more specific rule, reducing the wire length and maintaining the number of active lights. After calculating the number of possible mutations, I quickly gave up this method.

8.3 First results

Instead of using the GA to "guess" random rules, I placed the GA inside the CA. Making the GA traverse through the CA search for lights and building electrical wiring on its way. Much like the video game "snake", crawling its way through the CA, either going vertically or horizontally. This narrows down the number of possibilities to 6 ways per step (using elitism). The fitness function sums the distance to every light and uses it as it's score.

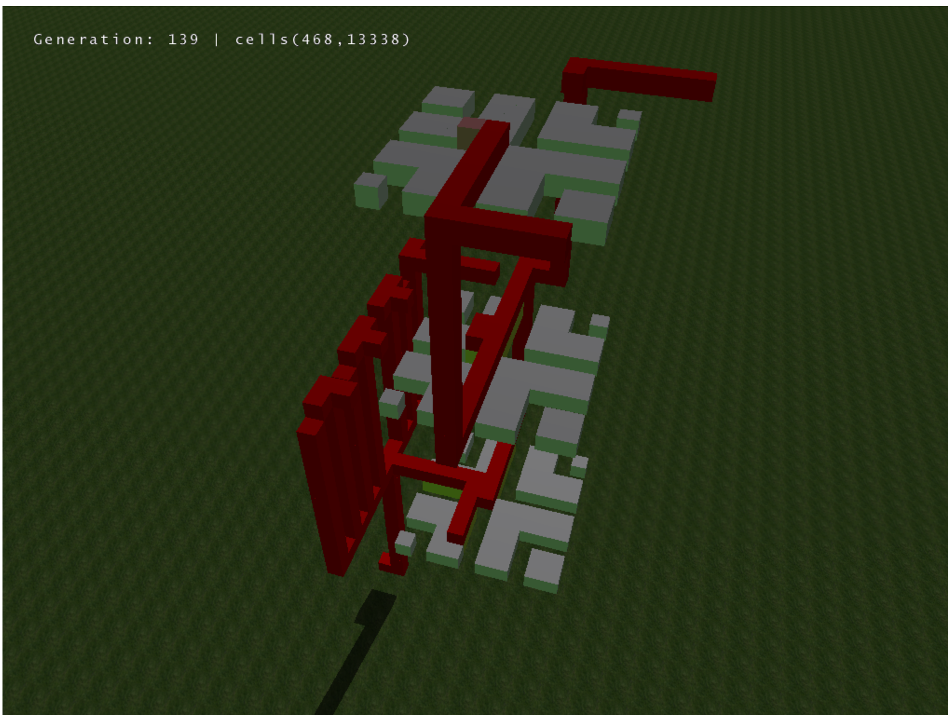


Figure 8.1: First results

[1]By inside I mean cells concealed by other floor or wall cells

After some few minutes I had my first results. In Figure 8.1 you can see the first solution activating any lights. It did have unnecessary wiring and did not cover every light, but it was something.

8.4 Divide and conquer

Having a too simple fitness function the "snake" can easily get confused and stuck in a local optima. Instead of making the GA grow toward every light, I added a outlet on each floor, decreasing the noise in the fitness function, and dividing the problem into sub-problems. The snake then went directly to the first outlet, and got stuck, going around and around the outlet (Figure 8.2).

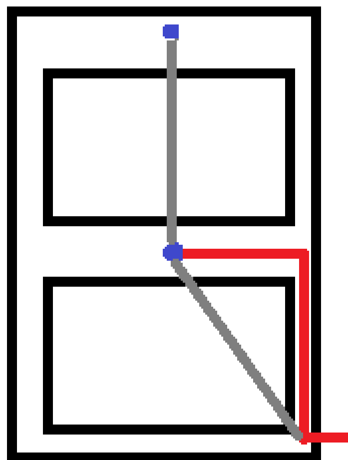


Figure 8.2: GA getting stuck at the first outlet. Red is the "snake", blue is the outlets and the gray being the Euclidean distance

The problem with using the Euclidean distance (Section 4.1) in a fitness function, is that it does not handle obstacles. The shortest path is always through the object, making it stuck at the wrong side, trying to go through. The solution was making the fitness function more sophisticated, using a shortest path algorithm supporting obstacles. The choice fell on the A* star algorithm (Section 4.2).

8.5 Sophisticated fitness function

You may ask "Why use the A* start search inside the fitness function, when you can used it to find the shortest path, and therefore solving the problem?". The answer is, we are not interested in the shortest path and the optimal solution, we want to see if a GA could find all outlets in a CA.

By running the A* search first, on the finished building without wiring, you get the shortest path to every outlet by running the algorithm n times, where n is the number of outlets. Later using the heuristics in the fitness function. This way a the GA finds all outlets in the CA.

The fitness function must also punish genomes adding extra wiring, keeping the total of wiring to a minimum.

The implementation of the A* fitness function unfortunately fell short, not giving any usable results, making it future work.

8.6 Activating the lights

Dividing the problem in Section 8.4 solved the problem of finding and powering the outlets, but the lights still lacked power. A "brute force" GA would probably find a small[2] enough solution in some hours, but it would be against the purpose of this thesis.

Looking at the initial structure, a touch crossed my mind. No matter how you design the GA, a light has to have a wire above to work. By adding a wire above each light, the search space was decreased from 48 possible rules to just 24 (In a 18x18 structure). The initial structure can be seen in Figure 8.3.

Filling the holes between the inactive wires, the worst case scenario is "just" $24!$, a much smaller number than my first brute force solution. Using a fitness function counting the number of active lights and giving penalties for extra wiring, the first solution came after 10 minutes. Having just 2 unnecessary wires, the solution was as good as hoped. The result can be viewed in Figure 8.4.

Using elitism, the GA always picked the same rules, this was a problem. While giving a good solution, the optimal solution was scraped. Always picking the last rule if multiple rules had the same amount of points. Instead, a random rule was chosen.

Being late in the thesis, I did not have the time to run the GA enough times to find a more optimal solution. However, these 5 rules kept coming, in different orders.

The algorithm used can be seen in Appendix A.2.2, or as simplified python code in Listing A.1.

[2]Using minimal wiring

Figure 8.3: Inactive wiring over inactive lights, electrical outlet as blue

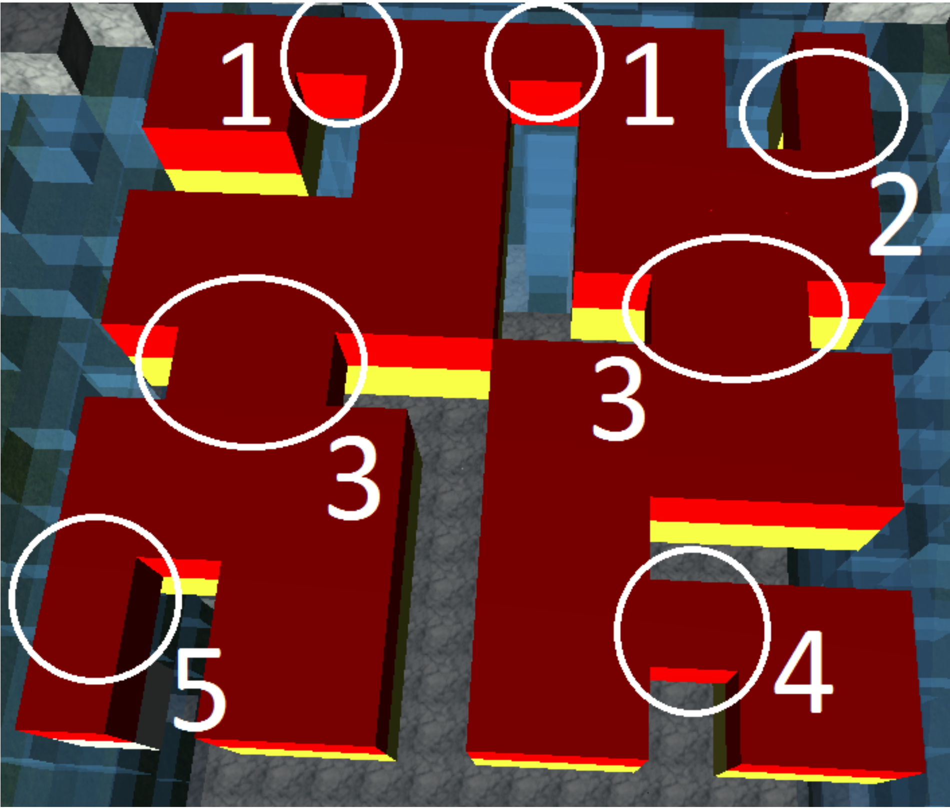


Figure 8.4: The final light configuration, all lights connected to the power outlet (blue). Numbers corresponding with Figure 8.5

1. 3-3-3-8-3-8-8-3-8-3-3-3-10-3-10-10-3-10-3-3-3-3-3-3-3-3-3-@10@GA made in generation 28 with p261
2. 3-8-3-3-3-3-8-8-3-3-10-3-3-3-3-10-10-3-3-3-3-3-3-3-3-3-3-@10@GA made in generation 29 with p583
3. 8-8-8-3-3-3-8-8-8-10-10-10-3-3-3-10-10-10-3-3-3-3-3-3-3-3-3-@10@GA made in generation 30 with p537
4. 8-3-3-8-3-8-8-3-8-10-3-3-10-3-10-10-3-10-3-3-3-3-3-3-3-3-3-@10@GA made in generation 31 with p576
5. 3-8-8-3-3-3-3-8-3-3-10-10-3-3-3-3-10-3-3-3-3-3-3-3-3-3-3-@10@GA made in generation 32 with p585

Figure 8.5: The final rules for the light wiring, best of 1000 solutions. Number corresponds with Figure 8.4

Chapter 9

Optimizations

9.1 Optimizations

The focus in this thesis was not optimizations; however some optimizations had to be done to get results in time.

9.2 Hashing

In a naive implementation to find the next state of a cell, would be to check every active cell, and compare the cell's neighborhood against all the rules. The neighborhood string presented in Section 6.3.1, is a string representation of a cell's neighborhood, and can be represented as "String" class in Java. Instead of compare every rule's neighborhood string against the neighborhood string of each cell, a hashing function can be used. In Java the HashMap class can make objects with a specified lookup key and a value for that specific key. By using the neighborhood string as a key, and the "next state" as a value, you can have instant lookup, speeding up the CA significantly.

9.3 Active cells

Depending on the structure, a CA can have clustered cells. For nearest neighbor CA, a new cell may only spawn next to an existing one. By storing a list of active cells and only calculating the next state for those, increase the performance drastically. Finding the fill limit when it stops being beneficial would be a interesting result for future work.

The observant soul would point out that only storing cells with a state would lead to no growth. That is correct. By storing all cells with state 0 in the neighborhood

of active cells with states other than 0, a membrane is created around the active ones.

9.4 Caching

Caching, temporarily storing information used often, is often used to increase performance. Calculation of the neighborhood string is done in a triple for loop, looping through the 27 cells in the neighborhood (included itself). When the structure becomes stable, only some hundred cells will change from generation to generation, leaving half a million unchanging cells recalculating and exhausting the system in vain. By caching the neighborhood string and checking against a boolean if the string has changed, a boost in performance is gained.

Each time a cell changes its state, all neighbors are alerted and the boolean is set to true. The next time the getNeighborhoodString() function is called, the string is recalculated and the boolean set to false.

This way a neighborhood string is only calculated when changed, saving millions of operations.

9.5 Reversion system for cells

Saving a state history for a cell makes jumping backwards and forwards faster. Especially practical when testing GAs, starting at a selected generation each time a genome is tested, instead of starting at generation 0 each time.

Chapter 10

Conclusions and future work

10.1 Conclusions

Cellular automata still has a lot of uncharted territory.

In recent years the interest have grown, researching traffic simulations, city planing, brain tumor simulations and recrystallization. Not able to find any other paper studying growth of buildings using multiple CA, one can assume this paper has a new angle on the usage of CAs.

As a proof of concept, I have shown that it possible to grow a stable skyscraper using multiple CA. The skyscraper has windows, floors, walls and automatically grows lights and electrical wiring to power the lights.

Whatever the future holds, more research will be put into CA.

10.2 Future work

In this chapter, possible future work is presented.

10.2.1 A* search in fitness function

Finishing the A* search used in the fitness function is Section 8.5. Then growing wires from the starting outlet in the basement up to all the other outlets in the structure.

10.2.2 Add models

Instead of using boxes with textures, add a model for each state. The building would look much more realistic. The support is there, only the models are missing.

10.2.3 Light distribution system

A system for placing lights and windows could be made, adapting the light to the content of the room.

10.2.4 Ventilation system

Growing a ventilation system to each room, making the wiring grow beside it, would be one vital step closer to realism.

10.2.5 Extend CA support

The simulation only supports one sub-CA. Rewriting the code to support a finite number of sub-CA and studying sub-CAs working together would be an interesting thesis.

10.2.6 Power outlets

Growing power outlets in the near vicinity of electrical devices, making the outlets appear where needed instead of moving the devices to the outlet.

10.2.7 Fill limit of the active cell list

In Section 9.3 , the problem of finding the optimal fill limit of the active cell list is described. A future project could find the fill limit and implement a system for turning on and off the usage of the active cell list. This would make the CA much faster when working on filled cell grid.

10.2.8 Fixing the revision system

The revision system worked nicely up to one week before delivery. I could not find the error in time.

10.3 The dream

An evolvable building, evolving itself to solve any problem occurring.

Instead of searching for a power outlet, you could hold the cable next to the wall. The building would then grow a power outlet with the additional required wiring.

A fire breaks out in the stairs, making it impossible to escape. At once, the building grows a new staircase or grow a fire hose.

Think about having an apartment with just one room. When something is needed, it is grown. The possibilities would be endless.

References

[1] S. Wolfram. (1984) Universality and complexity in cellular automata. [cited at p. 5, 6, 7, 9]

[2] D. Takahashi and J. Satsuma. (1990) A soliton cellular automaton. [Online]. Available: http://jpsj.ipap.jp/link?JPSJ/59/3514/ [cited at p. 5]

[3] S. Wolfram and N. H. Packard. (1985) Two-dimensional cellular automata. [Online]. Available: http://www.springerlink.com/content/w112414p4x464421/ [cited at p. 5]

[4] S. Wolfram. (1983) Statistical mechanics of cellular automata. [Online]. Available: http://www.ifsc.usp.br/~lattice/artigo-wolfram-cellular-autom.pdf [cited at p. 5, 6]

[5] ——, *A New Kind of Science*. Wolfram Media, 2002. [Online]. Available: http://www.wolframscience.com [cited at p. 5, 8, 9]

[6] M. Sipper. (2004) Evolution of parallel cellular machines - the cellular programming approach. [Online]. Available: http://www.cs.bgu.ac.il/~sipper/papabs/epcm.pdf [cited at p. 5]

[7] S. Wolfram. (1984) Computation theory of cellular automata. [Online]. Available: http://www.springerlink.com/content/kg133p4857gr3422/ [cited at p. 6]

[8] E. W. Weisstein. (2012) "rule 30. from mathworld–a wolfram web resource". [Online]. Available: http://mathworld.wolfram.com/Rule30.html [cited at p. xi, 6]

[9] J. Hallinan. Game of life. [Online]. Available: http://www.staff.ncl.ac.uk/j.s.hallinan/pubs/Life.pdf [cited at p. 8, 9, 11]

[10] C. G. Langton. (1990) Computation at the edge of chaos: Phase transitions and emergant computation. [Online]. Available: http://wiki.3xo.eu/pub/Education/SPM955xABMofCAS/LectureIntroductionToComplexity/Computation_at_the_edge_of_chaos__Langton.pdf [cited at p. 8, 9]

[11] W. Li, N. H. Packard, and C. Langton. (1990) Transition phenomena in cellular automata rule space. [Online]. Available: http://nslij-genetics.org/wli/pub/physicad90-no-figure.pdf [cited at p. 9]

[12] M. Mitchell, J. P. Crutchfield, and R. Das. (1996) Evolving cellular automata with genetic algorithms: A review of recent work. [Online]. Available: http://www.cs.unibo.it/babaoglu/courses/cas06-07/resources/tutorials/Evolving_Cellular_Automata.pdf [cited at p. 9, 13]

[13] S. Wolfram. (1985) Twenty problems in the theory of cellular automata. [Online]. Available: http://iopscience.iop.org/1402-4896/1985/T9/029 [cited at p. 9]

[14] G. Engelen, R. White, I. Uljee, and D. Paul. (1995) Using cellular automata for integrated modelling of socio-environmental systems. [Online]. Available: http://terra-geog.lemig2.umontreal.ca/donnees/geo3532/CA_integrated_model_ex.pdf [cited at p. 9]

[15] N. Margolus and T. Toffoli. (1987) Cellular automata machines. [Online]. Available: https://www.complex-systems.com/pdf/01-5-5.pdf [cited at p. 9]

[16] M. Rickert, K. Nagel, M. Schreckenberg, and A. Latour. (2008) Two lane trafic simulations using cellular automata. [Online]. Available: http://arxiv.org/pdf/cond-mat/9512119.pdf [cited at p. 9]

[17] K. Yamamoto, S. Kokubo, and K. Nishinari. (2007) Simulation for pedestrian dynamics by real-coded cellular automata (rca). [Online]. Available: http://copper.math.buffalo.edu/urgewiki/uploads/Literature2010Carbonara/Yamamoto.pdf [cited at p. 9]

[18] M. Gardner, *Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life"*. Scientific American 223: p120 - 123, 1970. [cited at p. 11]

[19] Gemini: Oblique life spaceship created. [Online]. Available: http://pentadecathlon.com/lifeNews/2010/05/oblique_life_spaceship_created.html [cited at p. 11]

[20] Gemini: Earlier than expected. [Online]. Available: http://conwaylife.com/forums/viewtopic.php?f=2&t=399&p=2327#p2327 [cited at p. 11]

[21] A. Kansal, S. Torquato, G. Harsh IV, E. Chiocca, and T. Deisboeck. (2000) Cellular automaton of idealized brain tumor growth dynamics. [Online]. Available: http://cherrypit.princeton.edu/papers/paper-175.pdf [cited at p. 11]

[22] D. Raabe. (2002) Cellular automata in materials science with particular reference to recrystallization simulation. [Online]. Available:

http://dierk-raabe.com/app/download/4037415602/Annu.+Rev.+Mater.+Res.+2002+vol+32+p+53+overview+cellular+automa.pdf [cited at p. 11]

[23] J. R. Koza and R. Poli. Genetic programming. [Online]. Available: http://dces.essex.ac.uk/staff/poli/papers/KozaPoli2005.pdf [cited at p. 13]

[24] G. Tufte and S. Nichele. (2011) On the correlations between developmental diversity and genomic composition. [Online]. Available: http://dl.acm.org/citation.cfm?id=2001779 [cited at p. 13]

[25] G. S. Hornby, A. Globus, D. S. Linden, and J. D. Lohn. (2006) Using cellular automata for integrated modelling of socio-environmental systems. [Online]. Available: http://alglobus.net/NASAwork/papers/Space2006Antenna.pdf [cited at p. 15]

[26] P. E. Hart, N. J. Nilsson, and B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics SSC4 4 (2): 100–107, 1968. [cited at p. 17]

[27] E. Dijkstra. (1959) A note on two problems in connexion with graphs. [Online]. Available: http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf [cited at p. 17]

[28] Y. D. Liang, *Introduction to Java Programming, Comprehensive Version, 6E*. PEASON Prentice Hall, 2006. [cited at p. 21]

[29] Why use opengl? [Online]. Available: http://blog.wolfire.com/2010/01/Why-you-should-use-OpenGL-and-not-DirectX [cited at p. 21]

[30] (2012) Minecraft - java game. [Online]. Available: http://www.minecraftwiki.net/wiki/Minecraft [cited at p. 22]

[31] Lightweight java game library. [Online]. Available: http://lwjgl.org/ [cited at p. 22]

[32] jmonkey engine - java game engine. [Online]. Available: http://jmonkeyengine.com/ [cited at p. 22]

[33] M. Napier. (2009) Glapp - java opengl demos with lwjgl. [Online]. Available: http://potatoland.org/code/gl/ [cited at p. 22]

Appendices

Appendix A

Source Code

The following chapter includes the most important source code files. Instructions for running the Java application can also be found in Section A.2.1. The rules for the final structure can be found in Section B.

A.1 Genetic Algorithm as a simplified Python script

Listing A.1: GA for powering lights from outlet

```
1  def power_lights_from_outlet():
2    game.goToGeneration(28) # Finsihed structure
3
4    # Find y coordinate of outlet
5    startY = findCellType(CellType.Outlet).y
6
7    rulesToCheck = []
8
9    for cell in cellsInSubgrid:
10     if cell.y == startY:
11       # skip existing inactive wires
12       if cell.state != CellType.InactiveWire:
13         rulesToCheck.add(c.neighborhoodString)
14
15   # Best rules for each generation is added here
16   chosenRules = []
17
18   while not finished:
19     generation++
20
21
22     for rule in rulesToCheck:
23       game.rulebook.add(rule) # Add rule temporarily
24
25       # Test rule, grow 30 more states
26       game.goToGeneration(28 + 30)
27
28       points.add(rule, fitnessFunction())
29
30       game.rulebook.remove(rule) # remove temporarily rule
31
32       game.goToGeneration(28) # Jump back before next test
33
```

```
34
35        bestRule = points.getRuleWithMostPoints()
36
37        game.rulebook.add(bestRule)
38
39        if getInactiveLights() == 0:
40          finished = true
41
42  def fitnessFunction():
43    # Encourage activating lights and punish extra wiring
44    return getActiveLights() * 10 - getLengthOfWires()
```

A.2 Java code

A.2.1 Running the CA

The application can be found in the attached zip file. On windows, unzip the file "Visualizer.zip" to a folder. Then starting the visualizer and the rule editor by double click the Master.jar file.

For Linux and Mac, try double clicking. If it does not work, the following command seen in Figure A.1 or A.2

java -jar Master.jar -Djava.library.path=drivers/native/linux

Figure A.1: Command for starting visualizer on linux

java -jar Master.jar -Djava.library.path=drivers/native/macosx

Figure A.2: Command for starting visualizer on Mac

A.2.2 Genetic algorithm for wiring the lights

The GeneticAlgorithms class (Listing A.2) contains the algorithm for running multiple instances of the GAInstance (Listing A.3), picking out the best one and sending it to the next generation. The Game class in "GeneticAlgorithms" is just a container for the CellGrid class (Listing A.6). The step function calls the step function in CellGrid, and some other calls to the visualizer. To make it simple, all information needed to be passed from generation to generation is put in a StateInformation object (Listing A.4). The OrderCell used in the GeneticAlgorithms class and the GAInstance class can be found in Listing A.5. At last the HashRule class (Listing A.8) is included, showing how a rule was structured.

Listing A.2: The overall genetic algorithm, starting up GAInstances and counting points

```
1   package genetic;
2
3   import glmodel.GL_Vector;
4
5   import java.io.BufferedWriter;
6   import java.io.FileWriter;
7   import java.text.SimpleDateFormat;
8   import java.util.ArrayList;
9   import java.util.Date;
10  import java.util.HashMap;
11  import java.util.Random;
12
13  import mechanics.CellGrid;
14  import mechanics.Game;
15  import ca.Cell;
16  import ca.CellStructures;
```

```
17  import ca.RuleBook;
18  import ca.rules.HashRule;
19
20
21  public class GeneticAlgorithms {
22
23    private RuleBook rulebook;
24    private SimpleDateFormat sdf;
25    private Random rand;
26    private ArrayList<CellOrder> finalOrder;
27
28
29    public GeneticAlgorithms() {
30      sdf = new SimpleDateFormat("yyyy-MM-dd_HH-mm-ss");
31      rulebook = new RuleBook();
32      rand = new Random();
33      finalOrder = new ArrayList<CellOrder>();
34
35      spreadTest();
36
37      /*
38      for(int i=0; i<100; i++){
39        System.out.println("Test "+i);
40        spreadTest();
41        System.out.println();
42        System.out.println();
43      }
44      */
45      //climbTest();
46      System.out.println("All␣finished");
47    }
48
49    private void spreadTest() {
50
51      int startGeneration = 28;
52      // Only grow one floor
53      Game game = new Game(null, CellStructures.BUILDNING_START, 6, 5, 6);
54      // Jump to finished buildning
55      game.goForwardToGeneration(startGeneration);
56
57      int startY = (int) GAInstance.findStartCell(game, 12).y;
58
59      CellGrid subGrid = game.getCellGrid().getSubGrid();
60      if(subGrid == null){
61        System.out.println("Subgrid␣Null!");
62        return;
63      }
64
65      // Find max/min values for lights in X and Z direction
66      int maxX = 0, maxZ = 0, minX = subGrid.getGridWidth(), minZ = subGrid.getGridDepth();
67      for(Cell c : game.getCellsInSubGrid()){
68        // If cell of type inactive light
69        if(c.getState() == 8){
70          if(c.getX() > maxX){
71            maxX = c.getX();
72          } else if(c.getX() < minX){
73            minX = c.getX();
74          }
75          if(c.getZ() > maxZ){
76            maxZ = c.getZ();
77          } else if(c.getZ() < minZ){
78            minZ = c.getZ();
79          }
80        }
81      }
82
83      // All lights starts with inactive wire above, therfor all cells in state 3
84      // on y level given by startCell, and between minX/maxX and minZ/maxZ is potensial
85      // candidates for a HashRule. TEST THEM ALL!
86      // measure them by a fitness function, and send the best through to the next generation
87      HashMap<String, GL_Vector> hashesToCheck = new HashMap<String, GL_Vector>();
```

```
88
89        for(Cell c : game.getCellsInSubGrid()){
90          if(c.getY() == startY){
91            if(c.getX() <= maxX && c.getX() >= minX && c.getZ() <= maxZ && c.getZ() >= minZ){
92              if(c.getState() != 10) hashesToCheck.put(c.getHash(), c.getPosition());
93            }
94          }
95        }
96
97        System.out.println(hashesToCheck.size());
98
99        if(hashesToCheck.size() == 0 ){
100         System.out.println("NO␣CELLS");
101         return;
102       }
103
104       boolean finished = false;
105       ArrayList<HashRule> chosenRules = new ArrayList<HashRule>();
106
107       HashMap<String, Integer> points = new HashMap<String, Integer>();
108       int generation = startGeneration;
109
110       long startTime = System.currentTimeMillis();
111       Integer highestPoints = 0;
112       String bestHash = "";
113
114       while(finished == false){
115         generation++;
116         System.out.println("NEW␣GENERATION␣=====␣"+generation);
117
118         // For all possible hashes
119         for(String hash : hashesToCheck.keySet()){
120           game = new Game(null, CellStructures.BUILDNING_START, 6, 5, 6);
121           // Jump to finished buildning
122           game.goForwardToGeneration(startGeneration);
123
124           // Add already chosen rules
125           for(HashRule cs : chosenRules){
126             game.getSubRuleBook().addHashRule(cs, false);
127           }
128
129           System.out.println("active␣rules:␣"+game.getSubRuleBook().getHashRules().size());
130
131           // Get hash and add it to rulebook
132           HashRule hashRule = new HashRule(hash, 10, "GA␣generated␣in␣generation␣"+startGeneration);
133           game.getSubRuleBook().addHashRule(hashRule, false);
134
135
136           // Try the rule for 30 generations
137           for(int i=0; i<30; i++){
138             // Test how successfull the rule was, save result in HashMap with position
139             int inactiveLights = game.getStateCount(8);
140             int activeLights = game.getStateCount(9);
141             double percent = activeLights / (double)(activeLights+inactiveLights);
142
143             //System.out.println("test: "+i+" | Percent completed: "+percent);
144             game.step();
145           }
146
147           // Test how successfull the rule was, save result in HashMap with position
148           int inactiveLights = game.getStateCount(8);
149           int activeLights = game.getStateCount(9);
150           int wireLength = game.getStateCount(10) + game.getStateCount(11);
151           double percent = activeLights / (double)(activeLights+inactiveLights);
152
153           System.out.println("G: "+generation+"␣|␣"+hash);
154           System.out.println("Lights:␣"+activeLights+"/"+inactiveLights);
155           System.out.println("Wire␣length:␣"+wireLength);
156           System.out.println("Percent␣completed:␣"+percent);
157
158           // If finished, stop
```

```
159         if((int)percent == 1) finished = true;
160         if(finished) System.out.println("FINISHED");
161 
162         // Calculate score
163         Integer score = activeLights*10 - wireLength;
164         System.out.println("Points:␣"+score);
165         // Save score
166         points.put(hash, score);
167 
168         // remove the rule from testgame
169         game.getSubRuleBook().removeHashRule(hashRule, false);
170 
171           // Always start from the finished building
172         // does not work :( make new game each time then...
173         //game.revertToGeneration(startGeneration);
174       }
175 
176 
177       // Find cell with highest points
178 
179       for(String hash : hashesToCheck.keySet()){
180         if(points.get(hash) > highestPoints){
181           highestPoints = points.get(hash);
182           bestHash = hash;
183 
184         // This just makes it some random, making it more likely to pic one of the two last alike scores
185         } else if(points.get(hash) == highestPoints && rand.nextBoolean()){
186           highestPoints = points.get(hash);
187           bestHash = hash;
188         }
189 
190       }
191 
192       // Clear points
193       points.clear();
194 
195       // Make rule for best cell, and remove it from checking in the future
196       HashRule bestRule = new HashRule(bestHash, 10,
197               "GA␣made␣in␣generation␣"+generation+"␣with␣p"+highestPoints);
198       chosenRules.add(bestRule);
199       hashesToCheck.remove(bestHash);
200       game.getSubRuleBook().addHashRule(bestRule, false);
201       rulebook.addHashRule(bestRule, false);
202 
203     }
204 
205     rulebook.saveRules("geneticAlgorithms/F" +highestPoints +
206               "␣-␣"+sdf.format(new Date())+"␣gen"+generation+".txt");
207     System.out.println("Solution␣found␣after␣"+
208               (System.currentTimeMillis() - startTime) + "␣ms");
209 
210   }
211 
212   private void climbTest() {
213 
214     GAInstance[] instances = new GAInstance[CellOrder.values().length];
215 
216     rulebook = new RuleBook();
217 
218     int startGeneration = 58;
219     boolean isFinished = false;
220 
221     for(int i=0; i<instances.length; i++){
222       instances[i] = new GAInstance(i, startGeneration, 12, rulebook, 6, 13, 6);
223     }
224 
225     for(int gen=0; gen<150; gen++){
226       System.out.println("New␣generation:␣"+gen);
227 
228       // Try all directions
229       for(int i=0; i<instances.length; i++){
```

```
230           instances[i].tryDirection();
231         }
232 
233         // Find best direction
234         StateInformation bestState = instances[0].state;
235         int bestIndex = 0;
236         for(int i=1; i<instances.length; i++){
237           if(instances[i].state.points > bestState.points){
238             bestState = instances[i].state;
239             bestIndex = i;
240             // Favoritise sequence
241           }/* else if(instances[i].state.points == bestState.points && rand.nextBoolean()){
242             bestState = instances[i].state;
243             bestIndex = i;
244           }*/
245         }
246         if(instances[bestIndex].getRule() != null){
247           System.out.println();
248           System.out.println("Chose: " + CellOrder.values()[bestIndex] +
249                         " ("+instances[bestIndex].state.position+")");
250           System.out.println();
251 
252           finalOrder.add(CellOrder.values()[bestIndex]);
253 
254 
255           // Make a rule of the best direction
256           rulebook.addHashRule(instances[bestIndex].getRule(), false);
257           rulebook.saveRules("geneticAlgorithms/p" + bestState.points + " - " +
258                         sdf.format(new Date())+" "+bestIndex+" gen"+gen+".txt");
259           for(int j=0; j<instances.length; j++){
260             instances[j].getGame().getSubRuleBook().addHashRule(instances[bestIndex].getRule(), false);
261           }
262         }
263 
264         // Copy best state to all and run one step
265         for(int k=0; k<instances.length; k++){
266           instances[k].setState(new StateInformation(bestState));
267           instances[k].step();
268 
269           if(instances[k].isFinished()){
270             rulebook.saveRules("geneticAlgorithms/F" + bestState.points +
271                               " - "+sdf.format(new Date())+" "+bestIndex+" gen"+gen+".txt");
272             try{
273               // Create file
274               FileWriter fstream = new FileWriter("geneticAlgorithms/p" +
275                               bestState.points + " - "+sdf.format(new Date())+
276                               " "+bestIndex+" gen"+gen+"-order.txt");
277               BufferedWriter out = new BufferedWriter(fstream);
278               for(CellOrder co : finalOrder){
279                 out.write(co+"\n");
280               }
281               //Close the output stream
282               out.close();
283             }catch (Exception e){//Catch exception if any
284               System.err.println("Error: " + e.getMessage());
285             }
286             isFinished = true;
287           }
288 
289 
290           if(isFinished) break;
291         }
292       }
293     }
294 
295     public static void main(String[] args) {
296       new GeneticAlgorithms();
297 
298     }
299 
300 
```

```
301 }
```

Listing A.3: GA instance crawling around like a snake

```
1   package genetic;
2
3   import mechanics.Game;
4
5   import org.lwjgl.util.vector.Vector3f;
6
7   import ca.Cell;
8   import ca.CellStructures;
9   import ca.RuleBook;
10  import ca.rules.HashRule;
11
12  public class GAInstance {
13
14    public CellOrder lastDirection;
15    public int id;
16    private Game game;
17
18    public StateInformation state;
19
20    private final int[] legalNeighborStates = { 3, 4, 8, 9, 10, 11 , 12};
21    private final int[] legalTakeoverStates = { 3, 4 };
22    private HashRule newestRule;
23    private boolean isFinished;
24    private int cellTypeToFind;
25
26    public GAInstance(int id, int generation, int cellTypeToFind, RuleBook rulebook,
27                                         int gridWidth, int gridHeight, int gridDepth){
28      this.id = id;
29
30      state = new StateInformation();
31
32      state.generation = generation;
33      this.cellTypeToFind = cellTypeToFind;
34
35      game = new Game(null, CellStructures.BUILDNING_START, gridWidth, gridHeight, gridDepth);
36      game.goForwardToGeneration(generation);
37      state.position = findStartCell(game, 11);
38
39    }
40
41
42    public static Vector3f findStartCell(Game game, int startCellState) {
43
44      Vector3f start = new Vector3f();
45
46      for(Cell c : game.getCellsInSubGrid()){
47        if(c.getState() == startCellState){
48          start.x = c.getX();
49          start.y = c.getY();
50          start.z = c.getZ();
51
52          System.out.println("Start:␣("+start.x+","+start.y+","+start.z+")");
53          return start;
54        }
55      }
56      return start;
57    }
58
59    private CellOrder getDirection() {
60      return CellOrder.values()[id];
61    }
62
63
64    private void moveDirection(CellOrder newDirection){
65      switch(newDirection){
66        case UP:
```

```
67              state.position.y++;
68              break;
69            case DOWN:
70              state.position.y--;
71              break;
72            case NORTH:
73              state.position.z--;
74              break;
75            case SOUTH:
76              state.position.z++;
77              break;
78            case EAST:
79              state.position.x++;
80              break;
81            case WEST:
82              state.position.x--;
83              break;
84        }
85      }
86
87      // Checks if current state.position is legal
88      private boolean legalPosition(int x, int y, int z) {
89        if( x < 0 || x >= game.getCellGrid().getSubGrid().getGridWidth()  ||
90          y < 0 || y >= game.getCellGrid().getSubGrid().getGridHeight() ||
91          z < 0 || z >= game.getCellGrid().getSubGrid().getGridDepth()
92           ){
93          return false;
94          }
95
96
97        String hash = game.getCellGrid().getSubGrid().getCellHash(x, y, z);
98        String[] hashArray = hash.split("-");
99
100       // Check if cell in state.positions is in a state legal to take over
101       boolean foundState = false;
102       for(int state : legalTakeoverStates){
103         if(Integer.parseInt(hashArray[hashArray.length/2]) == state){
104           foundState = true;
105         }
106       }
107
108       if(!foundState){
109         return false;
110       }
111
112       String replacedHash = hash;
113       for(int state : legalNeighborStates){
114         replacedHash = replacedHash.replace(state+"-", "");
115       }
116       if(replacedHash.length() > 0){
117         return false;
118       }
119
120       if(hash.contains("10-") || hash.contains("11-")){
121         return true;
122       }
123
124       return false;
125     }
126
127     private boolean legalPosition() {
128       return legalPosition((int)state.position.x, (int)state.position.y, (int)state.position.z);
129     }
130
131     public int getPoints(int failedPoints){
132       int positionPoints = 0;
133       int inactiveWireLength = game.getStateCount(10);
134       int activeWireLength = game.getStateCount(11);
135
136       int extraPoints = 0;
137       double x=0, y=0, z=0;
```

```
138         Cell lowestCell = null;
139
140         // distance to all inactive lights
141         for(Cell c : game.getCellsInSubGrid()){
142             if(c.getState() == cellTypeToFind){
143                 if(lowestCell == null){
144                     lowestCell = c;
145                 } else {
146                     if(lowestCell.getY() > c.getY()){
147                         lowestCell = c;
148                     }
149                 }
150             }
151         }
152
153         // Implement A* search here
154
155
156         x = Math.abs(lowestCell.getX()-state.position.x);
157         y = Math.abs(lowestCell.getY()-state.position.y);
158         z = Math.abs(lowestCell.getZ()-state.position.z);
159         System.out.println("("+x+","+y+","+z+")");
160
161         extraPoints += x*x;
162         extraPoints += y*y*y;
163         extraPoints += z+z;
164
165         positionPoints += extraPoints;
166         extraPoints = 0;
167
168         return -inactiveWireLength - activeWireLength - positionPoints - failedPoints;
169     }
170
171
172     public boolean tryDirection() {
173         state.generation++;
174         CellOrder direction = getDirection();
175
176         float originalX = state.position.x;
177         float originalY = state.position.y;
178         float originalZ = state.position.z;
179
180         moveDirection(direction);
181
182         boolean wasLegal = false;
183         int failedPoints = 0;
184
185         if(legalPosition()){
186
187             String hash = game.getCellGrid().getSubGrid().getCellHash(
188                                 (int)state.position.x,
189                                 (int)state.position.y,
190                                 (int)state.position.z
191                               );
192
193             HashRule newRule = new HashRule(hash, 10, "Generation: "+
194                        state.generation+" Direction: "+direction);
195
196             newestRule = newRule;
197
198             wasLegal = true;
199         } else {
200             // reset, not valid direction
201             state.position.x = originalX;
202             state.position.y = originalY;
203             state.position.z = originalZ;
204
205             System.out.println("Not legal "+direction);
206             newestRule = null;
207             failedPoints = 10000;
208         }
```

```
209
210
211        state.points = getPoints(failedPoints);
212        System.out.println("Points: "+state.points);
213        //state.points = getPoints(wasLegal, direction, failedPoints);
214
215        return wasLegal;
216    }
217
218
219    public void step() {
220        game.step();
221    }
222
223
224    public void setState(StateInformation state) {
225        this.state = state;
226    }
227
228
229    public HashRule getRule() {
230        return newestRule;
231    }
232
233
234    public Game getGame() {
235        return game;
236    }
237
238
239    public boolean isFinished() {
240        return isFinished;
241    }
242 }
```

Listing A.4: Information class for storing data between generations

```
1  package genetic;
2  import java.util.ArrayList;
3  import org.lwjgl.util.vector.Vector3f;
4  import ca.rules.HashRule;
5
6  public class StateInformation {
7
8    public int points;
9    public int generation;
10   public Vector3f position;
11
12   public StateInformation(StateInformation state) {
13     points = state.points;
14     generation = state.generation;
15     position = new Vector3f(state.position.x, state.position.y, state.position.z);
16   }
17
18   public StateInformation() {
19     points = 0;
20     generation = 0;
21     position = new Vector3f();
22   }
23 }
```

Listing A.5: Enum for valid directions

```
1  package genetic;
2
3  public enum CellOrder {
4    UP,
5    DOWN,
```

```
6     NORTH,
7     SOUTH,
8     EAST,
9     WEST;
10  }
```

Listing A.6: CellGrid class containing all cells and operations done on them

```
1   package mechanics;
2
3   import java.util.Collection;
4   import java.util.HashMap;
5   import java.util.Map;
6
7   import ca.Cell;
8   import ca.RuleBook;
9   import ca.gui.Visualizer;
10
11  public class CellGrid {
12
13    private CellGrid subGrid;
14    private Integer[][][] cellGrid;
15    private Map<String, Cell> activeCells;
16    private RuleBook rulebook;
17    private int gridWidth = 0;
18    private int gridHeight = 0;
19    private int gridDepth = 0;
20    private boolean isSubgrid = false;
21    private int subCellsPerCell = 1;
22
23    public CellGrid(RuleBook rulebook, int gridWidth, int gridHeight,
24          int gridDepth, int cellsPerCell, boolean isSubgrid) {
25      cellGrid = new Integer[gridWidth][gridHeight][gridDepth];
26      activeCells = new HashMap<String, Cell>();
27      this.rulebook = rulebook;
28
29      setGridWidth(gridWidth);
30      setGridHeight(gridHeight);
31      setGridDepth(gridDepth);
32      this.isSubgrid = isSubgrid;
33
34      this.subCellsPerCell = cellsPerCell;
35
36      if(isSubgrid){
37        // Place out start block for the active electrical system
38        //setCell(gridWidth-1, 1, gridDepth/2, 11);
39      }
40
41    }
42
43    public CellGrid(RuleBook rulebook, int gridWidth, int gridHeight, int gridDepth){
44      this(rulebook, gridWidth, gridHeight, gridDepth, 1, false);
45    }
46
47
48    public void createSubGrid(Visualizer visualizer, int subCellsPerCell,
49          int gridWidth, int gridHeight, int gridDepth){
50      RuleBook subRules = new RuleBook(visualizer, "subrules.txt");
51      subRules.loadRules();
52
53      // Extra width to place electric systems
54      subGrid = new CellGrid(subRules, gridWidth*subCellsPerCell+1,
55              gridHeight*subCellsPerCell, gridDepth*subCellsPerCell, subCellsPerCell, true);
56    }
57
58    public void populateNeighbourCells(int x, int y, int z) {
59      Integer c = getCellState(x,y,z);
60      if(c == null || c == 0) return;
61
62      for(int localX=x-1; localX<=x+1; localX++){
```

```
63          for(int localY=y-1; localY<=y+1; localY++){
64            for(int localZ=z-1; localZ<=z+1; localZ++){
65              if(localX >= 0 && localY >= 0 && localZ >= 0 && localX < getGridWidth()
66                      && localY < getGridHeight() && localZ < getGridDepth() ){
67                Integer cell = getCellState(localX, localY, localZ);
68                if(cell == null){
69                  setCell(localX, localY, localZ, 0, true);
70                }
71              }
72            }
73          }
74        }
75      }
76
77      public Integer getCellState(int x, int y, int z){
78        if(Game.DEBUG) System.out.println("getCell("+x+",␣"+y+",␣"+z+")");
79        if(x >= 0 && y >= 0 && z >= 0 && x < getGridWidth() &&
80                  y < getGridHeight() && z < getGridDepth() ){
81          return cellGrid[x][y][z];
82        }
83        return null;
84      }
85
86      public void step(){
87
88        if(Game.DEBUG) System.out.println("Number␣of␣cells:␣"+activeCells.size());
89
90        String[] keys = new String[activeCells.size()];
91        activeCells.keySet().toArray(keys);
92        for(int index = 0; index<keys.length; index++){
93          Cell c = activeCells.get(keys[index]);
94          populateNeighbourCells(c.getX(), c.getY(), c.getZ());
95        }
96
97        for(Cell c : activeCells.values()){
98          // Update to next state
99          c.updateNextState();
100       }
101
102       // All cells have calculated next state, now update them!
103       for(Cell c : activeCells.values()){
104         // Update to next state
105         c.updateState();
106         cellGrid[c.getX()][c.getY()][c.getZ()] = c.getState();
107
108         if(subGrid != null && isSubgrid == false && c.hasChangedState()){
109           subGrid.setCellCluster(c.getX(), c.getY(), c.getZ(), c.getState());
110         }
111       }
112
113       // Last, update subgrid
114       if(subGrid != null){
115         subGrid.step();
116       }
117     }
118
119     // Updates a cluster of cells (change of state in grid updates coresponding sub-cells)
120     private void setCellCluster(int x, int y, int z, int state) {
121       if(isSubgrid == false) return;
122       x = x * subCellsPerCell;
123       y = y * subCellsPerCell;
124       z = z * subCellsPerCell;
125
126       if(Game.DEBUG) System.out.println("Setting␣"+subCellsPerCell+"subcells␣for␣x:"+x+"␣y:"+y+"␣z:"+z);
127
128       if(x >= 0 && y >= 0 && z >= 0 && x < getGridWidth() &&
129                 y < getGridHeight() && z < getGridDepth() ){
130         for(int localX=x; localX<x+subCellsPerCell; localX++){
131           for(int localY=y; localY<y+subCellsPerCell; localY++){
132             for(int localZ=z; localZ<z+subCellsPerCell; localZ++){
133               if(localX >= 0 && localY >= 0 && localZ >= 0 &&
```

```
134                         localX < getGridWidth() &&
135                         localY < getGridHeight() && localZ < getGridDepth() ){
136                     setCell(localX, localY, localZ, state);
137                 }
138             }
139         }
140     }
141   }
142 }
143
144 private String planeAsString(int yValue){
145   String plane = "";
146   for(int tempZ = 0; tempZ<getGridDepth(); tempZ++){
147     for(int tempX = 0; tempX<getGridWidth(); tempX++){
148       Integer c = getCellState(tempX, yValue, tempZ);
149       if(c != null){
150         plane += c+"␣";
151       } else {
152         plane += "X␣";
153       }
154     }
155     plane += "\n";
156   }
157   return plane;
158 }
159
160 public void printOutPlane(int yValue) {
161   System.out.println(planeAsString(yValue));
162 }
163
164 public Collection<Cell> getActiveCells() {
165   return activeCells.values();
166 }
167
168
169 public CellGrid getSubGrid() {
170   return subGrid;
171 }
172
173 public boolean setCell(int x, int y, int z, Integer state, boolean updateSubgrid) {
174   if(x >= 0 && y >= 0 && z >= 0 && x < getGridWidth() &&
175             y < getGridHeight() && z < getGridDepth() ){
176     cellGrid[x][y][z] = state;
177     if(state != null)
178             activeCells.put(x+","+y+","+z, new Cell(this, rulebook, x, y, z, state));
179     if(updateSubgrid && getSubGrid() != null)
180             getSubGrid().setCellCluster(x, y, z, state);
181     return true;
182   }
183   return false;
184 }
185
186 public boolean setCell(int x, int y, int z, Integer state) {
187   return setCell(x, y, z, state, false);
188 }
189
190
191 public void clearCells() {
192   for(Cell c : getActiveCells()){
193     setCell(c.getX(), c.getY(), c.getZ(), null, false);
194   }
195
196   getActiveCells().clear();
197
198   if(subGrid != null){
199     subGrid.clearCells();
200   }
201 }
202
203
204 public RuleBook getRuleBook() {
```

```
205       return rulebook;
206     }
207
208
209     public boolean isSubgrid() {
210       return isSubgrid;
211     }
212
213     public void setGridWidth(int gridWidth) {
214       this.gridWidth = gridWidth;
215     }
216
217     public int getGridWidth() {
218       return gridWidth;
219     }
220
221     public void setGridHeight(int gridHeight) {
222       this.gridHeight = gridHeight;
223     }
224
225     public int getGridHeight() {
226       return gridHeight;
227     }
228
229     public void setGridDepth(int gridDepth) {
230       this.gridDepth = gridDepth;
231     }
232
233     public int getGridDepth() {
234       return gridDepth;
235     }
236
237     public String getCellHash(int x, int y, int z) {
238       return activeCells.get(x+","+y+","+z).getHash();
239     }
240
241     public void revert(int stopGeneraiton) {
242
243       for(Cell c : getActiveCells()){
244         setCell(c.getX(), c.getY(), c.getZ(), c.revertStateToGeneration(stopGeneraiton));
245         c.updateState();
246
247         if(subGrid != null && isSubgrid == false && c.hasChangedState()){
248           subGrid.setCellCluster(c.getX(), c.getY(), c.getZ(), c.getState());
249         }
250       }
251
252       if(!isSubgrid()) subGrid.revert(stopGeneraiton);
253     }
254
255
256   }
```

Listing A.7: Cell class, showing information stored about each cell

```
1   package ca;
2
3   import glmodel.GL_Vector;
4
5   import java.util.ArrayList;
6
7   import mechanics.CellGrid;
8   import mechanics.Game;
9
10
11  public class Cell {
12
13    private int posX;
14    private int posY;
15    private int posZ;
```

```
16
17    private int currentState;
18    private int nextState;
19    private RuleBook rulebook;
20    private CellGrid cellgrid;
21    private boolean hasChangedState = true;
22
23    private int currentGeneration = 0;
24
25    private ArrayList<Integer> stateHistory;
26    private String neighbourhoodHash = "";
27
28    public Cell(CellGrid cellgrid, RuleBook rulebook, int posX, int posY, int posZ, int state){
29      this.currentState = state;
30      setX(posX);
31      setY(posY);
32      setZ(posZ);
33
34      this.rulebook = rulebook;
35      this.nextState = state;
36      this.cellgrid = cellgrid;
37
38      stateHistory = new ArrayList<Integer>();
39    }
40
41    public Cell(CellGrid cellgrid, RuleBook rulebook, int posX, int posY, int posZ){
42      this( cellgrid, rulebook, posX, posY, posZ, 0);
43    }
44
45
46    /**
47     * Sets the next state as current state
48     */
49    public synchronized void updateState(){
50      stateHistory.set(currentGeneration, currentState);
51
52      currentState = nextState;
53
54      neighbourhoodHash  = "";
55
56      currentGeneration++;
57    }
58
59    public int getX(){
60      return posX;
61    }
62
63    public int getY(){
64      return posY;
65    }
66
67    public boolean hasChangedState(){
68      return hasChangedState;
69    }
70
71    public int getZ(){
72      return posZ;
73    }
74
75    private void setX(int posX) {
76      this.posX = posX;
77    }
78    private void setY(int posY) {
79      this.posY = posY;
80    }
81
82    private void setZ(int posZ) {
83      this.posZ = posZ;
84    }
85
86    @Override
```

```
87      public String toString() {
88        return "Cell ("+getX()+","+getY()+","+getZ()+") State: "+getState();
89      }
90
91      public int getState() {
92        return currentState;
93      }
94
95      public void updateNextState() {
96        // If state in history, use it
97        if(currentGeneration < stateHistory.size()){
98          System.out.println("Gets history from gen "+currentGeneration );
99          nextState = stateHistory.get(currentGeneration);
100       } else {
101         // else calculate new
102         nextState = rulebook.getNextState(cellgrid, posX, posY, posZ);
103         stateHistory.add(nextState);
104       }
105
106
107       hasChangedState = (currentState != nextState);
108
109       if(Game.DEBUG) System.out.println("Next state "+nextState + " x:"+posX + " y:"+posY + " z:" +posZ);
110     }
111
112     public void setState(int state){
113       currentState = state;
114     }
115
116     public Integer revertStateToGeneration(int generation){
117       if(generation >= currentGeneration) return currentState;
118
119       currentGeneration = generation;
120       /*
121       // Reverting more than possible, revert max
122       if( currentGeneration < generations ){
123         generations = currentGeneration;
124       }
125
126       int stateToRevertTo = currentGeneration-generations;
127       */
128       nextState = stateHistory.get(generation);
129
130       return nextState;
131     }
132
133     public String getHash() {
134       // If not changed, return
135       if(!neighbourhoodHash.isEmpty()){
136         return neighbourhoodHash;
137       }
138
139       // Else calculate hash
140       String ret = "";
141
142       for(int tempY = posY-1; tempY<=posY+1; tempY++){
143         for(int tempZ = posZ-1; tempZ<=posZ+1; tempZ++){
144           for(int tempX = posX-1; tempX<=posX+1; tempX++){
145             if(tempX >= 0 && tempY >= 0 && tempZ >= 0 && tempX < cellgrid.getGridWidth() &&
146                     tempY < cellgrid.getGridHeight() && tempZ < cellgrid.getGridDepth() ){
147               Integer c = cellgrid.getCellState(tempX, tempY, tempZ);
148               if(c != null){
149                 ret += c+"-";
150               } else {
151                 ret += "0-";
152               }
153             } else {
154               ret += "0-";
155             }
156           }
157         }
```

```
158        }
159
160        neighbourhoodHash = ret;
161        return ret;
162    }
163
164    public GL_Vector getPosition() {
165        return new GL_Vector(posX, posY, posZ);
166    }
167
168
169
170
171 }
```

Listing A.8: HashRule class, shows how the rule system works

```
1  package ca.rules;
2
3  import java.util.regex.Matcher;
4  import java.util.regex.Pattern;
5
6  public class HashRule extends Rule {
7
8    private String neighbourhood;
9    private Pattern pattern;
10
11   public HashRule(String neighbourhood, int nextState, String description) {
12     super(nextState, description);
13     this.neighbourhood = neighbourhood;
14
15     String replaced = neighbourhood.replace("x", "\\d+");
16     pattern = Pattern.compile(replaced);
17   }
18
19   @Override
20   public boolean checkCell(String hashFromCell) {
21
22     Matcher matcher = pattern.matcher(hashFromCell);
23     return matcher.matches();
24
25   }
26
27   public String getNeighbourhood(){
28     return neighbourhood;
29   }
30
31   @Override
32   public String toString() {
33     return "HashRule:␣"+getDescription();
34   }
35
36   public static boolean isInteger( char c )
37   {
38      try
39      {
40         Integer.parseInt( c + "" );
41         return true;
42      }
43      catch( NumberFormatException e )
44      {
45         return false;
46      }
47   }
48
49   public void setNeighbourhood(String neighbourhood){
50     this.neighbourhood = neighbourhood;
51   }
52
53 }
```

Appendix B

Rules

In the next sections the rules used in the final structure is listed.

B.1 Macro CA rules

Rules for growing floors, walls and windows.

Listing B.1: Function for finding a matching rule for cell

```
3–3–3–3–3–3–3–3–3–4–0–0–0–0–0–0–0–x–0–0–0–0–0–0–0–@5@New floor helper NW
0–0–6–0–0–4–0–0–0–3–2–2–3–2–2–0–0–0–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
0–4–0–0–6–0–0–0–0–0–1–1–0–1–1–0–3–3–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
0–0–0–0–0–0–0–0–0–0–0–0–0–0–2–0–2–2–0–0–0–0–0–0–0–@2@New diagonally
0–0–0–0–6–0–0–4–0–0–0–0–0–0–0–0–2–0–0–0–0–0–0–4–0–@2@New floor 2 North
0–0–0–0–0–0–0–0–0–0–1–1–0–1–3–0–3–2–0–0–0–0–0–0–0–@3@New south edge west
0–4–0–0–6–0–0–0–0–0–1–0–0–0–0–0–0–0–0–0–0–0–0–0–0–@1@New floor 1 South
0–0–0–0–0–0–0–0–0–1–1–1–1–1–3–1–3–2–0–0–0–0–0–0–0–@3@New middle going west
0–6–0–0–0–0–0–0–0–0–1–1–0–1–1–0–3–3–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
0–0–0–0–0–0–0–6–0–0–0–0–0–0–0–0–2–0–0–0–0–0–0–0–0–@2@New floor 2 North cont
0–0–0–x–0–x–x–x–x–0–0–0–x–0–x–x–x–x–0–0–0–3–3–3–3–3–3–@4@New Walls north
0–0–0–0–0–0–0–0–0–0–0–0–1–0–0–0–0–0–0–0–0–0–0–0–0–@1@New floor 1 East cont
4–0–0–6–0–0–0–0–0–1–1–0–1–0–0–0–0–0–0–0–0–0–0–0–0–@1@New floor diagonally
0–4–0–0–5–5–0–0–0–0–4–0–0–0–0–0–0–0–x–0–0–0–0–0–0–0–@5@Diagonal beams NW
0–4–14–0–5–7–0–14–7–0–4–14–0–6–7–0–14–7–0–3–3–0–3–3–0–3–3–@14@Remove floor making supports west
0–0–0–0–0–0–0–6–0–3–3–0–2–2–0–2–2–0–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
0–0–0–x–0–0–x–x–0–0–0–0–x–0–0–x–x–0–0–0–0–3–3–0–3–3–0–@4@Corner north east
0–0–0–x–x–x–x–x–x–0–0–0–x–4–x–x–7–x–0–0–0–x–x–x–x–x–x–@14@Make windows north corner
0–6–0–0–0–0–0–0–0–0–1–0–0–0–0–0–0–0–0–0–0–0–0–0–0–@1@New floor 1 South cont
0–0–0–0–0–0–0–0–0–1–3–0–3–2–0–2–2–0–0–0–0–0–0–0–0–@3@New north edge east
3–3–3–3–3–3–3–3–3–0–0–0–0–0–0–0–4–0–0–0–0–0–0–0–x–@5@New floor helper SE
0–0–0–0–0–0–0–0–0–1–1–1–1–1–3–1–3–3–0–0–0–0–0–0–0–@3@New middle going west
0–0–0–0–4–0–0–6–0–0–0–0–0–1–3–0–3–3–0–0–0–0–0–0–0–@3@Make solid floor over glass edge
0–0–0–0–0–0–0–0–0–1–3–2–3–2–2–0–0–0–0–0–0–0–0–0–0–@3@New south edge east
0–0–0–0–4–0–0–6–0–0–0–0–0–4–0–0–0–0–0–0–0–0–x–0–0–0–0–@1@Start new floor NE
4–0–0–6–0–0–0–0–0–1–1–3–1–1–3–3–3–3–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
0–0–6–0–0–4–0–0–0–3–2–2–3–2–2–0–0–0–0–0–0–0–4–0–0–0–@3@Make solid floor over floor helper
4–0–0–6–0–0–0–0–0–1–1–3–1–1–3–3–3–3–4–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
0–0–0–0–0–6–0–0–4–3–3–3–3–2–2–3–2–2–0–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
4–0–0–6–0–0–0–0–0–1–1–3–1–1–3–1–1–3–0–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
0–0–0–0–6–0–0–4–0–3–3–0–2–2–0–2–2–0–0–0–0–0–0–0–4–0–@3@Make solid floor over floor helper
0–x–x–0–x–x–0–x–x–0–x–x–0–4–7–0–x–x–0–x–x–0–x–x–0–x–x–@14@Make windows west corner
0–3–3–0–3–3–0–3–3–0–4–0–0–0–0–0–0–0–x–0–0–0–0–0–0–0–@5@New floor helper NW
0–4–0–0–6–0–0–0–0–0–1–0–0–0–0–0–0–0–4–0–0–0–0–0–0–0–@1@New floor 1 South
```

35 0–0–0–4–0–0–6–0–0–0–0–0–1–1–3–1–1–3–0–0–0–4–0–0–0–0–0–@3@Make solid floor over floor helper
36 x–x–x–x–x–x–0–0–0–x–7–x–x–4–x–0–0–0–x–x–x–x–x–x–0–0–0–@14@Make windows south corner
37 3–3–0–3–3–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–@4@New from bottom south east
38 0–0–0–0–0–0–0–0–0–0–0–0–1–1–3–1–3–2–0–0–0–0–0–0–0–0–0–@3@New north edge west
39 0–0–0–0–0–0–0–0–0–0–0–0–1–0–0–0–2–0–0–0–0–0–0–0–0–0–0–@3@New middle north edge
40 0–0–0–0–4–0–0–6–0–0–0–0–0–1–3–0–3–3–0–0–0–0–4–0–0–0–0–@3@Make solid floor over glass edge
41 x–x–x–x–x–x–x–x–x–x–x–x–x–0–x–x–x–x–3–3–3–3–3–3–3–3–3–@7@New room
42 0–0–0–0–3–3–0–3–3–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–@4@New from bottom north west
43 0–0–0–0–0–6–0–0–4–3–2–2–3–2–2–3–2–2–0–0–0–0–0–0–0–0–4–@3@Make solid floor over floor helper
44 x–x–x–x–0–x–0–0–0–x–x–x–x–0–x–0–0–0–3–3–3–3–3–3–0–0–0–@4@New Walls south
45 0–0–0–0–6–0–0–4–0–3–3–0–2–2–0–2–2–0–0–0–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
46 x–x–0–x–x–0–x–x–0–x–x–0–7–4–0–x–x–0–x–x–0–x–x–0–x–x–0–@14@Make windows east corner
47 0–x–x–0–0–x–0–0–0–0–x–x–0–0–x–0–0–0–0–3–3–0–3–3–0–0–0–@4@Corner south west
48 0–0–6–0–0–4–0–0–0–0–0–0–0–2–0–0–0–0–0–0–0–0–0–0–0–0–0–@2@New floor 2 West
49 0–0–0–0–0–0–0–0–0–3–3–2–3–2–2–2–2–2–0–0–0–0–0–0–0–0–0–@3@New middle going east
50 0–0–0–0–0–0–0–0–6–0–0–0–0–0–2–0–2–2–0–0–0–0–0–0–0–0–0–@2@New floor diagonally over floor helper
51 7–14–0–7–5–0–14–4–0–7–14–0–7–6–0–14–4–0–3–3–0–3–3–0–3–3–0–@14@Remove floor making supports east
52 0–0–0–5–5–0–0–4–0–0–0–0–0–0–0–0–4–0–0–0–0–0–0–0–0–x–0–@5@Diagonal beams SE
53 0–0–0–0–0–0–0–0–0–1–1–0–1–0–2–0–2–2–0–0–0–0–0–0–0–0–0–@3@New middle
54 0–0–0–4–0–0–6–0–0–0–0–0–1–0–0–0–0–0–0–0–0–0–0–0–0–0–0–@1@New floor 1 East
55 0–0–0–0–0–0–0–0–0–0–0–0–0–2–0–0–0–0–0–0–0–0–0–0–0–0–0–@2@New floor 2 West cont
56 0–0–0–0–0–0–0–0–0–0–1–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–@1@New down
57 0–0–0–0–0–0–0–0–6–3–3–3–3–2–2–3–2–2–0–0–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
58 x–x–0–x–0–0–x–x–0–x–x–x–x–0–x–x–x–x–3–3–0–3–3–0–3–3–0–@4@New Walls east
59 x–x–x–x–x–x–x–x–x–x–x–x–x–0–x–x–x–x–x–x–x–4–x–x–x–x–@4@Downwards **continue**
60 0–0–6–0–0–4–0–0–0–0–0–0–0–2–0–0–0–0–0–0–0–0–4–0–0–0–@2@New floor 2 West
61 0–0–0–0–0–6–0–0–4–0–0–0–0–0–2–0–2–2–0–0–0–0–0–0–0–0–0–@2@New floor diagonally
62 0–0–0–0–6–0–0–4–0–0–0–0–0–0–0–2–0–0–0–0–0–0–0–0–0–0–0–@2@New floor 2 North
63 6–0–0–0–0–0–0–0–0–1–1–3–1–1–3–3–3–3–0–0–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
64 0–0–0–4–0–0–6–0–0–0–0–0–1–0–0–0–0–0–0–0–4–0–0–0–0–0–@1@New floor 1 East
65 4–0–0–6–0–0–0–0–0–1–1–3–1–1–3–1–1–3–4–0–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
66 0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–2–0–0–0–0–0–0–0–0–0–0–0–@2@New up
67 x–6–x–0–4–0–x–0–x–x–0–x–0–4–0–x–0–x–x–0–x–0–4–0–x–0–x–@1@Grow upwards start W
68 0–6–0–0–4–0–0–0–0–0–0–0–4–0–0–0–0–0–0–0–x–0–0–0–0–@2@Start new floor SW
69 0–0–0–0–0–0–0–0–0–1–3–2–3–2–2–2–2–2–0–0–0–0–0–0–0–0–0–@3@New middle going east
70 3–3–0–3–3–0–3–3–0–0–0–0–0–0–0–0–4–0–0–0–0–0–0–0–0–x–0–@5@New floor helper SE
71 0–0–0–0–0–0–0–0–0–2–2–2–2–H–2–0–0–0–0–0–0–0–0–0–0–0–0–@2@Stay alive lower corner
72 0–0–0–0–0–6–0–0–4–3–3–3–3–2–2–3–2–2–0–0–0–0–0–0–0–0–4–@3@Make solid floor over floor helper
73 0–4–0–0–6–0–0–0–0–0–1–1–0–1–1–0–3–3–0–4–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
74 0–6–0–0–4–0–0–0–0–3–3–0–3–2–0–0–0–0–0–0–0–4–0–0–0–0–0–@3@Make solid floor over glass edge
75 0–6–0–0–4–0–0–0–0–3–3–0–3–2–0–0–0–0–0–0–0–0–0–0–0–0–0–@3@Make solid floor over glass edge
76 0–0–0–0–0–0–0–0–0–0–1–0–0–0–2–0–0–0–0–0–0–0–0–0–0–0–0–@3@New middle south edge
77 3–3–3–3–3–3–3–3–3–x–x–x–x–5–x–x–x–x–x–x–x–x–x–x–x–x–x–@7@New room from floor helper
78 x–x–x–x–4–x–x–x–x–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–@4@Long wall grouth
79 0–0–0–0–0–6–0–0–4–0–0–0–0–0–2–0–2–2–0–0–0–0–0–0–0–0–4–@2@New floor diagonally
80 x–x–x–x–x–x–x–x–x–x–x–x–x–0–x–x–x–x–x–x–x–x–7–x–x–x–x–@7@New room downwards
81 0–0–0–0–0–0–0–0–0–0–0–0–1–1–3–1–3–3–0–0–0–0–0–0–0–0–0–@3@New north edge west
82 x–x–x–x–x–x–x–x–x–x–x–x–x–5–x–x–x–x–x–x–x–x–14–x–x–x–x–@14@Remove support generic
83 0–4–0–0–5–0–0–0–0–0–4–0–0–0–0–0–0–0–0–x–0–0–0–0–0–0–0–@6@Before new floor NW
84 0–0–0–0–5–0–0–4–0–0–0–0–0–0–0–0–4–0–0–0–0–0–0–0–0–x–0–@6@Before new floor SE
85 0–0–0–4–0–0–6–0–0–0–0–0–1–1–3–1–1–3–0–0–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
86 0–0–0–0–0–0–0–0–0–3–3–2–3–2–2–0–0–0–0–0–0–0–0–0–0–0–0–@3@New south edge
87 0–0–0–0–0–0–0–0–0–1–1–0–1–0–0–0–0–0–0–0–0–0–0–0–0–0–0–@1@New diagonally
88 6–0–0–0–0–0–0–0–0–1–1–0–1–0–0–0–0–0–0–0–0–0–0–0–0–0–0–@1@New floor diagonally over floor helper
89 4–0–0–6–0–0–0–0–0–1–1–0–1–0–0–0–0–0–4–0–0–0–0–0–0–0–0–@1@New floor diagonally
90 0–6–0–0–4–0–0–0–0–0–0–0–0–4–0–0–0–0–0–0–0–0–4–0–0–0–0–@2@New floor with lots of glass
91 0–0–0–0–0–0–0–0–0–3–3–0–3–2–0–0–0–0–0–0–0–0–0–0–0–0–0–@3@Finish south east
92 0–0–0–0–0–0–0–0–0–3–3–0–3–2–0–2–2–0–0–0–0–0–0–0–0–0–0–@3@New north edge east
93 0–0–0–0–0–6–0–0–4–3–2–2–3–2–2–3–2–2–0–0–0–0–0–0–0–0–0–@3@Make solid floor over floor helper
94 0–0–0–0–0–0–0–0–0–0–0–0–0–1–3–0–3–3–0–0–0–0–0–0–0–0–0–@3@Finish north west
95 0–x–x–0–0–x–0–x–x–0–x–x–0–0–x–0–x–x–0–3–3–0–3–3–0–3–3–@4@New Walls west
96 0–0–0–0–0–0–0–0–0–0–1–1–0–1–3–0–3–3–0–0–0–0–0–0–0–0–0–@3@New south edge west
97 0–4–0–0–5–5–0–0–0–0–4–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–0–@5@Diagonal beams NW

B.2 Micro CA rules

Rules for adding details to windows, growing lights and some of the electrical wiring.

Listing B.2: Function for finding a matching rule for cell

```
1  x–x–x–x–x–x–x–x–x–x–x–x–11–10–x–x–x–x–x–x–x–x–x–x–x–x–x–@11@West wire active
2  x–x–x–x–x–x–x–x–x–x–14–x–x–14–x–x–7–x–x–x–x–x–x–x–x–x–x–@0@Remove inner glass south
3  x–x–x–x–x–x–x–x–x–x–x–x–x–10–x–x–x–x–x–x–x–x–11–x–x–x–x–@11@Above wire active
4  0–0–0–x–x–x–x–x–x–0–0–0–x–14–x–x–14–x–0–0–0–x–x–x–x–x–x–@0@Remove outer glass north
5  x–x–x–x–x–x–x–x–x–x–x–x–7–14–14–x–x–x–x–x–x–x–x–x–x–x–x–@0@Remove inner glass east
6  x–x–x–x–x–x–x–x–x–x–x–x–x–12–x–x–10–x–x–x–x–x–x–x–x–x–x–@10@Power outlet for floor too floor
7  x–x–x–x–x–x–x–x–x–x–x–x–14–14–7–x–x–x–x–x–x–x–x–x–x–x–x–@0@Remove inner glass west
8  x–x–x–x–x–x–x–x–x–x–x–x–x–10–11–x–x–x–x–x–x–x–x–x–x–x–x–@11@East wire active
9  x–x–x–x–11–x–x–x–x–x–x–x–x–10–x–x–x–x–x–x–x–x–x–x–x–x–x–@11@Below wire active
10 3–0–0–3–0–0–3–0–0–3–0–0–11–11–0–3–0–0–3–0–0–3–0–0–3–0–0–@0@Remove start wire
11 x–x–0–x–x–0–x–x–0–x–x–0–14–14–0–x–x–0–x–x–0–x–x–0–x–x–0–@0@Remove outer glass east
12 x–x–x–x–x–x–x–x–x–x–x–x–x–10–x–x–11–x–x–x–x–x–x–x–x–x–x–@11@South wire active
13 x–x–x–x–x–x–x–x–x–x–7–x–x–14–x–x–14–x–x–x–x–x–x–x–x–x–x–@0@Remove inner glass south
14 x–x–x–x–x–x–x–x–x–x–x–x–x–12–10–x–x–x–x–x–x–x–x–x–x–x–x–@10@Power outlet for floor too floor
15 x–x–x–x–x–x–x–x–x–x–x–x–x–3–x–x–12–x–3–3–3–3–3–3–3–3–3–@10@Connect power outlet with lights
16 x–x–x–x–x–x–0–0–0–x–14–x–x–14–x–0–0–0–x–x–x–x–x–x–0–0–0–@0@Remove outer glass south
17 0–x–x–0–x–x–0–x–x–0–x–x–0–14–14–0–x–x–0–x–x–0–x–x–0–x–x–@0@Remove outer glass west
18 7–7–7–7–7–7–7–7–7–3–3–3–3–3–3–3–3–3–x–x–x–x–x–x–x–x–x–@8@Make lights
19 x–x–x–x–x–x–x–x–x–x–x–x–x–8–x–x–x–x–x–x–x–x–11–x–x–x–x–@9@Make lights active when active wire above
20 x–x–x–x–x–x–x–x–x–x–x–x–10–12–x–x–x–x–x–x–x–x–x–x–x–x–x–@10@Power outlet for floor too floor
21 3–3–0–3–3–0–3–3–0–3–3–0–3–3–11–3–3–0–3–3–0–3–3–0–3–3–0–@11@Active wire entering floor
22 8–3–3–3–3–3–3–3–3–10–3–3–3–3–3–3–3–3–3–3–3–3–3–3–3–3–3–@12@Make outlet in south east corner
23 x–x–x–x–x–x–x–x–x–x–11–x–x–10–x–x–x–x–x–x–x–x–x–x–x–x–x–@11@North wire active
24 x–x–x–x–8–x–x–x–x–x–x–x–x–3–x–x–x–x–3–3–3–3–3–3–3–3–3–@10@Active wire entering floor
```

B.3 Rules produced by genetic algorithm

Read more in Section 8.6.

Listing B.3: Function for finding a matching rule for cell

```
1  3–3–3–8–3–8–8–3–8–3–3–3–10–3–10–10–3–10–3–3–3–3–3–3–3–3–3–@10@GA made in generation 28 with p261
2  3–8–3–3–3–3–8–8–3–3–10–3–3–3–3–10–10–3–3–3–3–3–3–3–3–3–3–@10@GA made in generation 29 with p583
3  8–8–8–3–3–3–8–8–8–10–10–10–3–3–3–10–10–10–3–3–3–3–3–3–3–3–3–@10@GA made in generation 30 with p537
4  8–3–3–8–3–8–8–3–8–10–3–3–10–3–10–10–3–10–3–3–3–3–3–3–3–3–3–@10@GA made in generation 31 with p576
5  3–8–8–3–3–3–3–8–3–3–10–10–3–3–3–3–10–3–3–3–3–3–3–3–3–3–3–@10@GA made in generation 32 with p585
```