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.

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, reproduere 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 bygninger 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>Two-dimensional Cellular Automata</td>
<td>9</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Game of Life</td>
<td>11</td>
</tr>
<tr>
<td>2.7</td>
<td>Three-dimensional Cellular Automata and usage</td>
<td>11</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Stopping growth</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Genetic algorithms</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Selection</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Reproduction</td>
<td>13</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Crossover</td>
<td>13</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Mutation</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Fitness function</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>Usage</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>Criticisms</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>Searching</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Euclidean distance</td>
<td>17</td>
</tr>
<tr>
<td>4.2</td>
<td>A* search</td>
<td>17</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Concept</td>
<td>18</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Example</td>
<td>18</td>
</tr>
<tr>
<td>II</td>
<td>Technology</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>Technology</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Java</td>
<td>21</td>
</tr>
<tr>
<td>5.2</td>
<td>Renderer</td>
<td>22</td>
</tr>
<tr>
<td>5.2.1</td>
<td>LWJGL</td>
<td>22</td>
</tr>
<tr>
<td>5.2.2</td>
<td>GLApp</td>
<td>22</td>
</tr>
<tr>
<td>III</td>
<td>Results</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>The Cellular Automata</td>
<td>27</td>
</tr>
<tr>
<td>6.1</td>
<td>The simple CA</td>
<td>28</td>
</tr>
<tr>
<td>6.2</td>
<td>Making of the cell grid</td>
<td>29</td>
</tr>
<tr>
<td>6.3</td>
<td>Rulebook</td>
<td>29</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Hash Rules</td>
<td>29</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Don’t care states</td>
<td>30</td>
</tr>
<tr>
<td>6.4</td>
<td>Macro CA</td>
<td>30</td>
</tr>
<tr>
<td>6.5</td>
<td>Rule editor</td>
<td>31</td>
</tr>
<tr>
<td>6.6</td>
<td>Visualizer</td>
<td>32</td>
</tr>
<tr>
<td>6.7</td>
<td>Design choices</td>
<td>33</td>
</tr>
<tr>
<td>6.8</td>
<td>States</td>
<td>33</td>
</tr>
</tbody>
</table>
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
LIST OF FIGURES

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 years. 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.
CHAPTER 1. INTRODUCTION

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 grid 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].

\[
a_i^{(t)} = F[a_{i-r}^{(t-1)}, a_{i-r+1}^{(t-1)}, \ldots, a_i^{(t-1)}, \ldots, a_{i+r}^{(t-1)}]
\] (2.1)
In Equation 2.1 the arrays of $a_i^{(t-1)}$ and the neighborhood is sent to $F$, the rule function, determining the next state of $a_i^{(t)}$. $F$ 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

$$k^{k^{(2^r+1)}} = 2^{2^{2^1+1}} = 256$$  \hspace{1cm}  (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]](image)
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.

\[
k^{k^{(2* r + 1)^d}}
\]  

\(^1\)Vertical, Horizontal, Diagonal - The choice is yours
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 μακρόν 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.

\[ k^{k^{(2r+1)d}} = 2^{(2r+1)^2} = 1.4078079 \times 10^{154} \]  

(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.
CHAPTER 2. CELLULAR AUTOMATA

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 cell is either 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 7.2 and 7.3.
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 consist of multiple chromosomes, much like chromosomes in an individual’s DNA, controlling the behavior of the system.

All genomes are tested 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.

![Genetic Algorithms Diagram](image)

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 straightforward 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.

![ST5-33.142.7 antenna](image)

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.
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.

\[ d(p, q) = d(q, p) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2} \] (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,
- \(h(x)\) 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

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 $g(x) = \sqrt{10^2 + 10^2} = 14.142... \approx 14$. 

Figure 4.1: Example of A* search
Part II

Technology
5.1 Java

Java[28] is an 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 OS X 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\[^2\] 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.

![Image of First CA with "Game of Life", with the oscillator pattern "Beacon"](image)

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.

![Figure 6.3: Neighborhood text string from a hash rule](image)

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

```python
def getNextState(x, y, z):
    neighborhoodString = getCellNeighborhoodString(x, y, z)
    if hashRules.has(neighborhoodString):
        return hashRules.getState(neighborhoodString)
    else:
        for rule in dontCareHashRules:
            if rule.matches(neighborhoodString):
                return rule.nextState
        # No matching rules, return current state
        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.

$$\text{total number of states}^{\text{number of don't cares}} \quad (6.1)$$

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.

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.
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 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

---

1See 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.
CHAPTER 6. THE CELLULAR AUTOMATA

(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 somewhat 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 Figure 7.2a.
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.
Figure 7.3: Growing a floor using 3 cell types over 11 generations
7.3. LEVELS

(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 itself 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.

![Inactive lights](image1.png) ![Active lights](image2.png)

**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.
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-
neighborhood 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.

---

\(^1\)By inside I mean cells concealed by other floor or wall cells
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](image)

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
CHAPTER 8. GENETIC ALGORITHMS

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

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


REFERENCES


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

```python
def power_lights_from_outlet():
    game.goToGeneration(28)  # Finished structure

    # Find y coordinate of outlet
    startY = findCellType(CellType.Outlet).y

    rulesToCheck = []

    for cell in cellsInSubgrid:
        if cell.y == startY:
            # skip existing inactive wires
            if cell.state != CellType.InactiveWire:
                rulesToCheck.add(c.neighborhoodString)

    # Best rules for each generation is added here
    chosenRules = []

    while not finished:
        generation++

        for rule in rulesToCheck:
            game.rulebook.add(rule)  # Add rule temporarily

            # Test rule, grow 30 more states
            game.goToGeneration(28 + 30)

            points.add(rule, fitnessFunction())

        game.rulebook.remove(rule)  # remove temporarily rule

        game.goToGeneration(28)  # Jump back before next test
```

59
APPENDIX A. SOURCE CODE

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

---

60 APPENDIX A. SOURCE CODE
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

```java
package genetic;

import glm.model.GL_Vector;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.Random;
import mechanics.CellGrid;
import mechanics.Game;
import ca.Cell;
import ca.CellStructures;
```
import ca.RuleBook;
import ca.rules.HashRule;

public class GeneticAlgorithms {
    private RuleBook rulebook;
    private SimpleDateFormat sdf;
    private Random rand;
    private ArrayList<CellOrder> finalOrder;

    public GeneticAlgorithms() {
        sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        rulebook = new RuleBook();
        rand = new Random();
        finalOrder = new ArrayList<CellOrder>();
        spreadTest();
    }

    public void spreadTest() {
        for (int i = 0; i < 100; i++) {
            System.out.println("Test "+i);
            spreadTest();
            System.out.println();
            System.out.println();
        }
    }

    private void spreadTest() {
        int startGeneration = 28;
        Game game = new Game(null, CellStructures.BUILDNING_START, 6, 5, 6);
        game.goForwardToGeneration(startGeneration);
        int startY = (int) GAInstance.findStartCell(game, 12).y;
        CellGrid subGrid = game.getCellGrid().getSubGrid();
        if (subGrid == null) {
            System.out.println("Subgrid Null!");
            return;
        }

        int maxX = 0, maxZ = 0, minX = subGrid/Grid.Width(), minZ = subGrid/Grid.Depth();
        if (subGrid == null) {
            System.out.println("Subgrid_Null!");
            return;
        }

        // Find max/min values for lights in X and Z direction
        int maxX = 0, maxZ = 0, minX = subGrid/Grid.Width(), minZ = subGrid/Grid.Depth();
        for (Cell c : game.getCellsInSubGrid()) {
            if (c.getStat() == 8) {
                if (c.getX() > maxX)
                    maxX = c.getX();
                else if (c.getX() < minX)
                    minX = c.getX();
                if (c.getZ() > maxZ)
                    maxZ = c.getZ();
                else if (c.getZ() < minZ)
                    minZ = c.getZ();
            }
        }

        // All lights start with inactive wire above, therfor all cells in state 3
        for (Cell c : game.getCellsInSubGrid()) {
            if (c.getStat() == 8) {
                if (c.getX() > maxX)
                    maxX = c.getX();
                else if (c.getX() < minX)
                    minX = c.getX();
                if (c.getZ() > maxZ)
                    maxZ = c.getZ();
                else if (c.getZ() < minZ)
                    minZ = c.getZ();
            }
        }

        // candidates for a HashRule. TEST THEM ALL!
        // measure them by a fitness function, and send the best through to the next generation
        HashMap<String, GL_Vector> hashesToCheck = new HashMap<String, GL_Vector>();
for (Cell c : game.getCellsInSubGrid()) {
if (c.getY() == startY) {
  if (c.getX() <= maxX && c.getX() >= minX && c.getZ() <= maxZ && c.getZ() >= minZ) {
    if (c.getState() != 10) hashesToCheck.put(c.getHash(), c.getPosition());
  }
}
System.out.println(hashesToCheck.size());
if (hashesToCheck.size() == 0) {
  System.out.println("NO CELLS");
  return;
}
boolean finished = false;
ArrayList<HashRule> chosenRules = new ArrayList<HashRule>();
HashMap<String, Integer> points = new HashMap<String, Integer>;
int generation = startGeneration;
long startTime = System.currentTimeMillis();
Integer highestPoints = 0;
String bestHash = "";
while (finished == false) {
  generation++;
  System.out.println("NEW GENERATION ===== " + generation);
  // For all possible hashes
  for (String hash : hashesToCheck.keySet()) {
    game = new Game(null, CellStructures.BUILDING_START, 6, 5, 6);
    // Jump to finished building
    game.goForwardToGeneration(startGeneration);
    // Add already chosen rules
    for (HashRule cs : chosenRules) {
      game.getSubRuleBook().addHashRule(cs, false);
    }
    System.out.println("active rules: " + game.getSubRuleBook().getHashRules().size());
    // Get hash and add it to rulebook
    HashRule hashRule = new HashRule(hash, 10, "GA generated in generation " + startGeneration);
    game.getSubRuleBook().addHashRule(hashRule, false);
    // Try the rule for 30 generations
    for (int i = 0; i < 30; i++) {
      int inactiveLights = game.getStateCount(8);
      int activeLights = game.getStateCount(9);
      double percent = activeLights / (double)(activeLights + inactiveLights);
      System.out.println("test: +i+ Percentage completed: " + percent);
      game.step();
    }
    // Test how successful the rule was, save result in HashMap with position
    int inactiveLights = game.getStateCount(8);
    int activeLights = game.getStateCount(9);
    int wireLength = game.getStateCount(10) + game.getStateCount(11);
    double percent = activeLights / (double)(activeLights + inactiveLights);
    System.out.println("G: " + generation + " hash: " + hash);
    System.out.println("Lights: " + activeLights + " wireLength: " + wireLength);
    System.out.println("Percent completed: " + percent);
  }
  System.out.println("G: " + generation + " hash: " + hash);
  System.out.println("Lights: " + activeLights + " wireLength: " + wireLength);
  System.out.println("Percent completed: " + percent);
  // If finished, stop
if ((int) percent == 1) finished = true;
if (finished) System.out.println("FINISHED");

// Calculate score
int score = activeLights * 10 - wireLength;
System.out.println("Points: "+score);

// Save score
points.put(hash, score);

// remove the rule from test game
game.getSubRuleBook().removeHashRule(hashRule, false);

// Always start from the finished building
// does not work :( make new game each time then ...
//game.revertToGeneration(startGeneration);

// Find cell with highest points
for (String hash : hashesToCheck.keySet()) {
    if (points.get(hash) > highestPoints)
        highestPoints = points.get(hash);
    bestHash = hash;
    if (points.get(hash) == highestPoints && rand.nextBoolean())
        highestPoints = points.get(hash);
    bestHash = hash;
}

// Clear points
points.clear();

// Make rule for best cell, and remove it from checking in the future
HashRule bestRule = new HashRule(bestHash, 10,
    "GA made in generation " + generation + " with score " + highestPoints);
chosenRules.add(bestRule);
hashesToCheck.remove(bestHash);
game.getSubRuleBook().addHashRule(bestRule, false);
rulebook.addHashRule(bestRule, false);

rulebook.saveRules("geneticAlgorithms/" + highestPoints + "+sdf.format(new Date())+"gen" + generation + ".txt");
System.out.println("Solution found after "+
    (System.currentTimeMillis() - startTime) + " ms");

private void climbTest() {
    GAInstance[] instances = new GAInstance[CellOrder.values().length];
    rulebook = new RuleBook();
    int startGeneration = 58;
    boolean isFinished = false;

    for (int i = 0; i < instances.length; i++)
        instances[i] = new GAInstance(i, startGeneration, 12, rulebook, 6, 13, 6);

    for (int gen = 0; gen < 150; gen++)
        System.out.println("New generation: "+gen);

    // Try all directions
    for (int i = 0; i < instances.length; i++)

A.2. JAVA CODE

```java
// Find best direction
StateInformation bestState = instances[0].state;
int bestIndex = 0;

for(int i = 1; i < instances.length; i++){
    if(instances[i].state.points > bestState.points)
        bestState = instances[i].state;
    bestIndex = i;
}

// Favorise sequence
if(instances[bestIndex].state.points == bestState.points && rand.nextBoolean())
    bestState = instances[bestIndex].state;

// Make a rule of the best direction
rulebook.addHashRule(instances[bestIndex].getRule(), false);
rulebook.saveRules("geneticAlgorithms/p" + bestState.points + "," + sdf.format(new Date()) + "+gen" + gen + ".txt");

if(instances[bestIndex].getRule() != null)
    System.out.println("Chose:", CellOrder.values()[bestIndex] + "+
instances[bestIndex].state.position")
    System.out.println();
finalOrder.add(CellOrder.values()[bestIndex]);
```

```java
// Copy best state to all and run one step
for(int k = 0; k < instances.length; k++){
    instances[k].setState(new StateInformation(bestState));
    instances[k].step();
    if(instances[k].isFinished()){
        rulebook.saveRules("geneticAlgorithms/F" + bestState.points + "," + sdf.format(new Date()) + "+bestIndex+" + gen + ".txt");
        try {
            // Create file
            FileWriter fstream = new FileWriter("geneticAlgorithms/p" + bestState.points + "," + sdf.format(new Date()) + "," + bestIndex + "," + gen + ".txt");
            BufferedWriter out = new BufferedWriter(fstream);
            for(CellOrder co : finalOrder){
                out.write(co + "n");
            }
            out.close();
        } catch (Exception e){ // Catch exception if any
            System.err.println("Error:", e.getMessage());
        }
        isFinished = true;
    }
    if(isFinished) break;
}
```

```java
public static void main(String[] args) {
    new GeneticAlgorithms();
}
```
Listing A.3: GA instance crawling around like a snake

```java
package genetic;
import mechanics.Game;
import org.lwjgl.util.vector.Vector3f;
import ca.Cell;
import ca.CellStructures;
import ca.RuleBook;
import ca.rules.HashRule;

public class GAInstance {
    public CellOrder lastDirection;
    public int id;
    private Game game;

    public GAInstance(int id, int generation, int cellTypeToFind, RuleBook rulebook,
                      int gridWidth, int gridHeight, int gridDepth) {
        this.id = id;
        state = new StateInformation();
        this.generation = generation;
        this.cellTypeToFind = cellTypeToFind;
        game = new Game(null, CellStructures.BUILDING_START, gridWidth, gridHeight, gridDepth);
        game.goForwardToGeneration(generation);
        state.position = findStartCell(game, 11);
    }

    public static Vector3f findStartCell(Game game, int startCellState) {
        Vector3f start = new Vector3f();
        for(Cell c : game.getCellsInSubGrid()){
            if(c.getState() == startCellState){
                start.x = c.getX();
                start.y = c.getY();
                start.z = c.getZ();
            }
        }
        System.out.println("Start:
```
```java
    state.position.y++;  
    break;
  case DOWN:
    state.position.y--;  
    break;
  case NORTH:
    state.position.z--;  
    break;
  case SOUTH:
    state.position.z++;  
    break;
  case EAST:
    state.position.x++;  
    break;
  case WEST:
    state.position.x--;  
    break;
}
  }

  // Checks if current state.position is legal
  private boolean legalPosition(int x, int y, int z) {  
    if (x < 0 || x >= game.getCellGrid().getSubGrid().getGridWidth() ||
        y < 0 || y >= game.getCellGrid().getSubGrid().getGridHeight() ||
        z < 0 || z >= game.getCellGrid().getSubGrid().getGridDepth()) {
      return false;
    }

    String hash = game.getCellGrid().getSubGrid().getCellHash(x, y, z);  
    String[] hashArray = hash.split("-");

    // Check if cell in state.positions is in a state legal to take over
    boolean foundState = false;
    for (int state : legalTakeoverStates) {  
      if (Integer.parseInt(hashArray[hashArray.length / 2]) == state) {
        foundState = true;
      }
    }

    if (!foundState) {
      return false;
    }

    String replacedHash = hash;
    for (int state : legalNeighborStates) {  
      replacedHash = replacedHash.replace(state + "-", "");
    }

    if (replacedHash.length() > 0) {
      return false;
    }

    if (hash.contains("10-") || hash.contains("11-")) {
      return true;
    }

    return false;
  }

  private boolean legalPosition() {  
    return legalPosition((int) state.position.x, (int) state.position.y, (int) state.position.z);
  }

  public int getPoints(int failedPoints) {  
    int positionPoints = 0;
    int inactiveWireLength = game.getStateCount(10);
    int activeWireLength = game.getStateCount(11);
    int extraPoints = 0;
    double x=0, y=0, z=0;
```
Cell lowestCell = null;

// distance to all inactive lights
for(Cell c : game.getCellsInSubGrid()){
  if(c.getState() == cellTypeToFind) {
    if(lowestCell == null){
      lowestCell = c;
    } else {
      if(llowestCell.getY() > c.getY()){
        lowestCell = c;
      }
    }
  }
}

// Implement A* search here

float originalX = state.position.x;
float originalY = state.position.y;
float originalZ = state.position.z;
moveDirection(direction);

boolean wasLegal = false;
int failedPoints = 0;

if(legalPosition()) {
  String hash = game.getCellGrid().getSubGrid().getCellHash(
      (int)state.position.x,
      (int)state.position.y,
      (int)state.position.z
    );
  HashRule newRule = new HashRule(hash, 10, "Generation:", + 
      state.generation+",Direction:", + direction);
  newestRule = newRule;
  wasLegal = true;
} else {
  // reset, not valid direction
  state.position.x = originalX;
  state.position.y = originalY;
  state.position.z = originalZ;
  System.out.println("Not legal", +direction);
  newestRule = null;
  failedPoints = 10000;
}
```java
state.points = getPoints(failedPoints);
System.out.println("Points:");
// state.points = getPoints(wasLegal, direction, failedPoints);
return wasLegal;
}

public void step() {
game.step();
}

public void setState(StateInformation state) {
this.state = state;
}

public HashRule getRule() {
return newestRule;
}

public Game getGame() {
return game;
}

public boolean isFinished() {
return isFinished;
}
```

Listing A.5: Enum for valid directions

```java
package genetic;

public enum CellOrder {
UP,
DOWN,
LEFT,
RIGHT
}
```

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

```java
package genetic;
import java.util.ArrayList;
import org.lwjgl.util.vector.Vector3f;
import ca.rules.HashRule;

public class StateInformation {
public int points;
public int generation;
public Vector3f position;

public StateInformation(StateInformation state) {
points = state.points;
generation = state.generation;
position = new Vector3f(state.position.x, state.position.y, state.position.z);
}

public StateInformation() {
points = 0;
generation = 0;
position = new Vector3f();
}
```

```java
package genetic;

public enum CellOrder {
UP,
DOWN,
LEFT,
RIGHT
}
```
Listing A.6: CellGrid class containing all cells and operations done on them

```java
package mechanics;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import ca.Cell;
import ca.RuleBook;
import ca.gui.Visualizer;

public class CellGrid {

    private CellGrid subGrid;
    private Integer[][][] cellGrid;
    private Map<String, Cell> activeCells;
    private RuleBook rulebook;
    private int gridWidth = 0;
    private int gridHeight = 0;
    private int gridDepth = 0;
    private boolean isSubgrid = false;
    private int subCellsPerCell = 1;

    public CellGrid(RuleBook rulebook, int gridWidth, int gridHeight, int gridDepth, int subCellsPerCell, boolean isSubgrid) {
        cellGrid = new Integer[gridWidth][gridHeight][gridDepth];
        activeCells = new HashMap<String, Cell>();
        this.rulebook = rulebook;
        setGridWidth(gridWidth);
        setGridHeight(gridHeight);
        setGridDepth(gridDepth);
        this.isSubgrid = isSubgrid;
        this.subCellsPerCell = subCellsPerCell;
        if(isSubgrid) {
            // Place out start block for the active electrical system
            // setCell(gridWidth-1, 1, gridDepth/2, 11);
        }
    }

    public void createSubGrid(Visualizer visualizer, int subCellsPerCell, int gridWidth, int gridHeight, int gridDepth) {
        RuleBook subRules = new RuleBook(visualizer, "subrules.txt");
        subRules.loadRules();
        subGrid = new CellGrid(subRules, gridWidth*subCellsPerCell+1, gridHeight*subCellsPerCell, gridDepth*subCellsPerCell, subCellsPerCell, true);
    }

    public void populateNeighbourCells(int x, int y, int z) { 
        Integer c = getCellState(x, y, z);
        if (c == null || c == 0) return;
        for(int localX=x-1; localX<=x+1; localX++)
```

70

APPENDIX A. SOURCE CODE
for (int localY = y - 1; localY <= y + 1; localY++)
    for (int localZ = z - 1; localZ <= z + 1; localZ++)
        if (localX >= 0 && localY >= 0 && localZ >= 0 && localX < getGridWidth() &&
            localY < getGridHeight() && localZ < getGridDepth())
            Integer cell = getCellState(localX, localY, localZ);
        if (cell == null)
            setCell(localX, localY, localZ, 0, true);
    }
}

public Integer getCellState(int x, int y, int z)
if (Game.DEBUG) System.out.println("getCell("+x+","+y+","+z+");
if (x >= 0 && y >= 0 && z >= 0 && x < getGridWidth() &&
    y < getGridHeight() && z < getGridDepth())
    return cellGrid[x][y][z];
return null;

public void step()
if (Game.DEBUG) System.out.println("Number of cells: "+activeCells.size());
String[] keys = new String[activeCells.size()];
activeCells.keySet().toArray(keys);
for (int index = 0; index < keys.length; index++)
    Cell c = activeCells.get(keys[index]);
    populateNeighbourCells(c.getX(), c.getY(), c.getZ());
}
for (Cell c : activeCells.values())
    // Update to next state
    c.updateNextState();
for (Cell c : activeCells.values())
    // Update to next state
    c.updateState();
    cellGrid[c.getX()][c.getY()][c.getZ()] = c.getState();
    if (subGrid != null && isSubgrid == false && c.hasChangedState())
        subGrid.setCellCluster(c.getX(), c.getY(), c.getZ(), c.getState());
}

// Last, update subgrid
if (subGrid != null)
    subGrid.step();

// Updates a cluster of cells (change of state in grid updates corresponding sub-cells)
private void setCellCluster(int x, int y, int z, int state)
if (isSubgrid == false) return;
x = x * subCellsPerCell;
y = y * subCellsPerCell;
z = z * subCellsPerCell;
if (Game.DEBUG) System.out.println("Setting subCellsPerCell+subcells "+x+" y:"+y+" z:"+z);
if (x >= 0 && y >= 0 && z >= 0 && x < getGridWidth() &&
    y < getGridHeight() && z < getGridDepth())
    for (int localX = x; localX < x + subCellsPerCell; localX++)
        for (int localY = y; localY < y + subCellsPerCell; localY++)
            for (int localZ = z; localZ < z + subCellsPerCell; localZ++)
                if (localX >= 0 && localY >= 0 && localZ >= 0 &&
                    localX < getGridWidth() && localY < getGridHeight() &&
                    localZ < getGridDepth()))
localX < getGridWidth() &&
localY < getGridHeight() && localZ < getGridDepth() } {
    setCell(localX, localY, localZ, state);
}

private String planeAsString(int yValue) {
    String plane = "";
    for (int tempZ = 0; tempZ < getGridDepth(); tempZ++){
        for (int tempX = 0; tempX < getGridWidth(); tempX++){
            Integer c = getCellState(tempX, yValue, tempZ);
            if (c != null) {
                plane += c + "X";
            } else {
                plane += "X";
            }
        }
    }
    return plane;
}

public void printOutPlane(int yValue) {
    System.out.println(planeAsString(yValue));
}

public Collection<Cell> getActiveCells() {
    return activeCells.values();
}

public CellGrid getSubGrid() {
    return subGrid;
}

public boolean setCell(int x, int y, int z, Integer state, boolean updateSubgrid) {
    if (x >= 0 && y >= 0 && z >= 0 && x < getGridWidth() && y < getGridHeight() && z < getGridDepth()) {
        cellGrid[x][y][z] = state;
        if (state != null) {
            activeCells.put(x++, "y", z++, new Cell(this, rulebook, x, y, z, state));
        if (updateSubgrid && getSubGrid() != null) {
            getSubGrid().setCellCluster(x, y, z, state);
        return true;
    } else {
        return false;
    }
}

public boolean setCell(int x, int y, int z, Integer state) {
    return setCell(x, y, z, state, false);
}

public void clearCells() {
    for (Cell c : getActiveCells()) {
        setCell(c.getX(), c.getY(), c.getZ(), null, false);
    }
    getActiveCells().clear();
    if (subGrid != null) {
        subGrid.clearCells();
    }
}

public RuleBook getRuleBook() {
A.2. JAVA CODE

    205         return rulebook;
    206     }
    207
    208     public boolean isSubgrid () {
    209         return isSubgrid;
    210     }
    211
    212     public void setGridWidth(int gridWidth) {
    213         this.gridWidth = gridWidth;
    214     }
    215
    216     public int getGridWidth () {
    217         return gridWidth;
    218     }
    219
    220     public void setGridHeight(int gridHeight) {
    221         this.gridHeight = gridHeight;
    222     }
    223
    224     public int getGridHeight () {
    225         return gridHeight;
    226     }
    227
    228     public void setGridDepth(int gridDepth) {
    229         this.gridDepth = gridDepth;
    230     }
    231
    232     public int getGridDepth () {
    233         return gridDepth;
    234     }
    235
    236     public String getCellHash(int x, int y, int z) {
    237         return activeCells.get(x+,y+,z).getHash();
    238     }
    239
    240     public void revert(int stopGeneration) {
    241         for(Cell c : getActiveCells()){
    242             setCell(c.getX(), c.getY(), c.getZ(), c.revertStateToGeneration(stopGeneration));
    243             c.updateState();
    244         }
    245         if(subGrid != null && isSubgrid == false && c.hasChangedState()){
    246             subGrid.setCellCluster(c.getX(), c.getY(), c.getZ(), c.getState());
    247         }
    248         if(!isSubgrid()) subGrid.revert(stopGeneration);
    249     }
    250
    251  
    252 
  
Listing A.7: Cell class, showing information stored about each cell

    1 package ca;
    2 import glmmodel.GL_Vector;
    3 import java.util.ArrayList;
    4 import mechanics.CellGrid;
    5 import mechanics.Game;
    6 public class Cell {
    7     private int posX;
    8     private int posY;
    9     private int posZ;
private int currentState;
private int nextState;
private RuleBook rulebook;
private CellGrid cellgrid;
private boolean hasChangedState = true;
private int currentGeneration = 0;
private ArrayList<Integer> stateHistory;
private String neighbourhoodHash = "";

public Cell(CellGrid cellgrid, RuleBook rulebook, int posX, int posY, int posZ, int state) {
    this.currentState = state;
    setX(posX);
    setY(posY);
    setZ(posZ);
    this.rulebook = rulebook;
    this.nextState = state;
    this.cellgrid = cellgrid;
    stateHistory = new ArrayList<Integer>();
}

public Cell(CellGrid cellgrid, RuleBook rulebook, int posX, int posY, int posZ) {
    this(cellgrid, rulebook, posX, posY, posZ, 0);
}

/**
 * Sets the next state as current state
 */
public synchronized void updateState() {
    stateHistory.set(currentGeneration, currentState);
    currentState = nextState;
    neighbourhoodHash = "";
    currentGeneration++;}

public int getX() {
    return posX;
}

public int getY() {
    return posY;
}

public boolean hasChangedState() {
    return hasChangedState;
}

public int getZ() {
    return posZ;
}

private void setX(int posX) {
    this.posX = posX;
}

private void setY(int posY) {
    this.posY = posY;
}

private void setZ(int posZ) {
    this.posZ = posZ;
}

@Override
public String toString() {
    return "Cell(" + getX() + "," + getY() + "," + getZ() + ") State: " + getState();
}

public int getState() {
    return currentState;
}

public void updateNextState() {
    // If state in history, use it
    if (currentGeneration < stateHistory.size()) {
        System.out.println("Gets history from gen + currentGeneration");
        nextState = stateHistory.get(currentGeneration);
    } else {
        // else calculate new
        nextState = rulebook.getNextState(cellgrid, posX, posY, posZ);
        stateHistory.add(nextState);
    }

    hasChangedState = (currentState != nextState);

    if (Game.DEBUG) System.out.println("Next state: + state + x: +posX + y: +posY + z: +posZ");
}

public void setState(int state) {
    currentState = state;
}

public Integer revertStateToGeneration(int generation) {
    if (generation >= currentGeneration) return currentState;

    currentGeneration = generation;
    // Reverting more than possible, revert max
    if (currentGeneration < generations) {
        generations = currentGeneration;
    }

    int stateToRevertTo = currentGeneration - generations;
    nextState = stateHistory.get(generation);

    return nextState;
}

public String getHash() {
    // If not changed, return
    if (!neighbourhoodHash.isEmpty()){
        return neighbourhoodHash;
    }

    // Else calculate hash
    String ret = "";

    for(int tempY = posY-1; tempY<=posY+1; tempY++){
        for(int tempZ = posZ-1; tempZ<=posZ+1; tempZ++){
            if (tempX == posX-1 & tempX <= 0 & tempZ >= 0 & tempZ < cellgrid.getCellState(tempX, tempY, tempZ)) {
                if (cellgrid.getCellState(tempX, tempY, tempZ) != null){
                    ret += cellgrid.getCellState(tempX, tempY, tempZ);
                } else {
                    ret += "0-";
                }
            } else {
                ret += "0-";
            }
        }
    }

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

```java
package ca.rules;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class HashRule extends Rule {

private String neighbourhood;
private Pattern pattern;

public HashRule(String neighbourhood, int nextState, String description) {
    super(nextState, description);
    this.neighbourhood = neighbourhood;
    String replaced = neighbourhood.replace("x", "\\d+" );
    pattern = Pattern.compile(replaced);
}

@Override
public boolean checkCell(String hashFromCell) {
    Matcher matcher = pattern.matcher(hashFromCell);
    return matcher.matches();
}

public String getNeighbourhood() {
    return neighbourhood;
}

@Override
public String toString() {
    return "HashRule:\" + getDescription();
}

public static boolean isInteger(char c) {
    try {
        Integer.parseInt( c + "" );
        return true;
    } catch (NumberFormatException e) {
        return false;
    }
}

public void setNeighbourhood(String neighbourhood) {
    this.neighbourhood = neighbourhood;
}
}
```
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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>New floor helper NW</td>
</tr>
<tr>
<td>2</td>
<td>Make solid floor over floor helper</td>
</tr>
<tr>
<td>3</td>
<td>New south edge east</td>
</tr>
<tr>
<td>4</td>
<td>Make solid floor over floor helper</td>
</tr>
<tr>
<td>5</td>
<td>New middle going west</td>
</tr>
<tr>
<td>6</td>
<td>New floor helper 2 North</td>
</tr>
<tr>
<td>7</td>
<td>New floor 1 South</td>
</tr>
<tr>
<td>8</td>
<td>New solid floor over floor helper</td>
</tr>
<tr>
<td>9</td>
<td>New floor 2 North cont</td>
</tr>
<tr>
<td>10</td>
<td>New Walls north</td>
</tr>
<tr>
<td>11</td>
<td>New floor 1 East cont</td>
</tr>
<tr>
<td>12</td>
<td>New floor diagonally</td>
</tr>
<tr>
<td>13</td>
<td>Remove floor making supports west</td>
</tr>
<tr>
<td>14</td>
<td>Make solid floor over floor helper</td>
</tr>
<tr>
<td>15</td>
<td>New floor 1 South cont</td>
</tr>
<tr>
<td>16</td>
<td>New north east</td>
</tr>
<tr>
<td>17</td>
<td>Make windows north corner</td>
</tr>
<tr>
<td>18</td>
<td>New south east</td>
</tr>
<tr>
<td>19</td>
<td>New north east</td>
</tr>
<tr>
<td>20</td>
<td>New south west</td>
</tr>
<tr>
<td>21</td>
<td>New middle going west</td>
</tr>
<tr>
<td>22</td>
<td>Make solid floor over glass edge</td>
</tr>
<tr>
<td>23</td>
<td>Start new floor NE</td>
</tr>
<tr>
<td>24</td>
<td>New solid floor over floor helper</td>
</tr>
<tr>
<td>25</td>
<td>New solid floor over floor helper</td>
</tr>
<tr>
<td>26</td>
<td>New solid floor over floor helper</td>
</tr>
<tr>
<td>27</td>
<td>Make solid floor over floor helper</td>
</tr>
<tr>
<td>28</td>
<td>Make solid floor over floor helper</td>
</tr>
<tr>
<td>29</td>
<td>Make solid floor over floor helper</td>
</tr>
<tr>
<td>30</td>
<td>Make solid floor over floor helper</td>
</tr>
<tr>
<td>31</td>
<td>Make solid floor over floor helper</td>
</tr>
<tr>
<td>32</td>
<td>Make windows west corner</td>
</tr>
<tr>
<td>33</td>
<td>New floor helper NW</td>
</tr>
<tr>
<td>34</td>
<td>New floor 1 South</td>
</tr>
</tbody>
</table>
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-011@West wire active
2  x-x-x-x-x-x-x-x-x-x-14-x-x-x-x-x-x-x-x-x-x-x-x-x-010@Remove inner glass south
3  x-x-x-x-x-x-x-x-x-x-10-x-x-x-x-x-x-x-x-x-x-x-x-011@Above wire active
4  0-0-0-x-x-x-x-x-0-0-x-14-x-x-14-x-0-0-0-x-x-x-x-x-x-000@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-x-x-x-000@Remove inner glass east
6  x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-