Generating two-dimensional game maps with use of cellular automata

Michał Wolski

April 23, 2019

Table of contents

Chapter 1

Introduction

During recent years, presence of computer games in human lives has increased. The amount of time spent on playing games by the modern society has shown that games are desirable both as a means for entertainment and a medium of expression. However, as the interest in games rises [1] and computer games become increasingly complex, the need for game content must also rise. Elements such as believable maps, textures, sound and models (among other types of content) are a necessary resource for production of games.

Studies such as [1] show where the evidence for insufficiency of manual content creation may be found. In the study, authors point to work of Kelly and McCabe [2], Lefebvre and Neyret [3], Smelik et al. [4] and Iosup [5] as sources which reveal game content production as a time-consuming and expensive endeavour. Hence, it is logical to conclude that information contained in studies and statistics on the topic of game development suggest that most projects aimed at creating games or simulations could benefit from seeking new or more efficient automated means of content creation.

Solving the inefficiency issue

In order to provide a solution to the inefficiency of manual content production, formal methods have emerged and are commonly referred to as *procedural generation techniques*, defined by the literature as processes or methods of automatic content creation, through algorithmic mechanisms [6] [7].

Scientific surveys such as [1] and [4] show why investigating procedural gen-

[1] The Interactive Software Federation of Europe compiles and publishes statistics which include frequency of gaming in European countries and show that demand for games is on the rise. `https://www.isfe.eu/industry-facts/statistics`

eration is useful for the game industry, by providing examples of successful methods which can be used to generate content for games. Primary concerns which drive the interest in automated ways to create game content are the rising project costs and increasing development time.

In order to reduce the cost of game development, allow for greater replay value or provide a feeling of vastness to the game worlds that designers aim to create, procedural content generation techniques can provide an attractive solution to the problem of content creation. Surveys such as [1], [6] and [8] show what types of game content can be generated and are a good starting point for seeking methods of procedural generation.

1.1 Objectives

This thesis focuses on automated creation of 2-dimensional game maps using a cellular automata approach to generate small map tiles and merge them into a bigger map. Such approach allows for a degree of control to the map designer – who may want to decide which tiles will be merged and at which locations in the map they will be present. Integrating manual editing or parametrization of desired results with procedural generation techniques has been proposed before in the works concerning procedural generation techniques [9], [10], [11].

Beginning experimentation with flat maps on 2-dimensional plane avoids the complexity that may arise when dealing with higher dimensions, hence the main aim is to develop a solution to the problem of automating planar map creation for games.

Specifically, maps created by the generator should be planar and their layout must have characteristics of shapes that can be helpful in design of believable environments. These characteristics may include irregularity, asymmetry and imbalance. The goal is to create structures that are interesting because of their unpredictability, somewhat resembling the look of results produced by natural processes like erosion, dissolution, deformation, tectonic fragmentation or weathering.

1.2 Thesis scope

The research carried out to support this thesis was focused on discovery of generative techniques presented in current literature describing methods of content generation for games.

Work on preparing the solution has been divided into the following parts, described in the remainder of this thesis:

- research on procedural generation of planar maps and method selection
- design of a map generator program and its implementation in C++
- experiments to find a set of rules that generate satisfying maps
- extending the feature set of the generator

1.3 Thesis structure

This thesis includes introduction followed by three chapters. chapter 2 serves as a study on possible mechanisms that could be used for procedural generation and specifically, for creation of 2D maps for games. chapter 3 describes design and implementation of a solution to the problem along with performed experiments. Chapter chapter 4 summarizes the findings and concludes the thesis.

1.4 Experimental setup

The solution that allowed to carry out experiments in this thesis was implemented using the C++ programming language and compiled with MSVC++ 14.0 compiler, natively included in the Visual Studio 2015 Community Edition IDE. Other tools and libraries used in the project:

- Dear ImGui, by Omar Cornut - to easily build an Immediate Mode user interface. Project homepage: `https://github.com/ocornut/imgui`
- GLFW 3.2.1 library - to create an OpenGL context and have direct access to texture functions. Project homepage: `http://www.glfw.org/`

In order to satisfy requirements for using OpenGL API function calls, it is recommended to use a dedicated graphics processing unit and install updated drivers. For the purposes of development and experiments *nVidia GeForce GTX 560M* was used and drivers were updated to latest available versions.

This thesis has been prepared with LaTeX system for document typesetting, included diagrams were drawn with *UMLet* - an open source modelling program.

1.5 Abbreviations and acronyms

The following terms, abbreviations and acronyms have been used in the thesis.

CA Cellular Automaton (or Automata). A simulation consisting of cell objects.

PCG Procedural Content Generation. An automated process of creation.

GUI Graphical User Interface

ASCII American Standard Code for Information Interchange - a character encoding standard, used to represent characters in computers and information systems

RPG Role-Playing Game

Chapter 2

Research on 2D map generation methods

2.1 Maps and cartography

Historically, maps have been used by the human race since ancient times. The need for navigation in the world has been a driving force behind the evolution of maps. Starting with cave paintings and representations of stars on the sky, our kind had the need for capturing an abstract model of a territory, terrain shape, location of useful resources or some other aspect of surrounding environment in a useful way.

Making a model of the physical (or fictional) world with maps requires choice of the data types describing locations represented on the map. List of data types visualized with maps has been growing with the evolution of cartography and whenever new technologies have been introduced to the map making crafts. Some examples of data possible to represent on maps are:

- physical maps - terrain shape, elevation, forests, bodies of water, etc.
- political maps - borders around a territory, districts, states
- climate and weather maps - temperature, humidity, precipitation, wind currents
- geologic maps - terrain features, location of precious resources underground
- star maps - views of the distant cosmic objects measured by solid angles from a fixed point

- route maps - transport links, connections joining points on the modelled territory

Although early maps had the form of drawings or etchings on surface of solid materials, now there are other possibilities of representing the abstract model which a map aims to represent. The rise of digital maps and geographic information systems has opened new possibilities - maps have become dynamic entities, stored digitally, easily updated with new data and not limited to the boundaries of physical model. With digital maps, it is possible to show more than one layer of data, as chosen by the user, whereas physical maps are limited to the data and view scale chosen initially at the time when map was crafted. Despite the limitations, they still can serve well as a medium for storage of geographic information, requiring simpler processes during archival and conservation efforts [12].

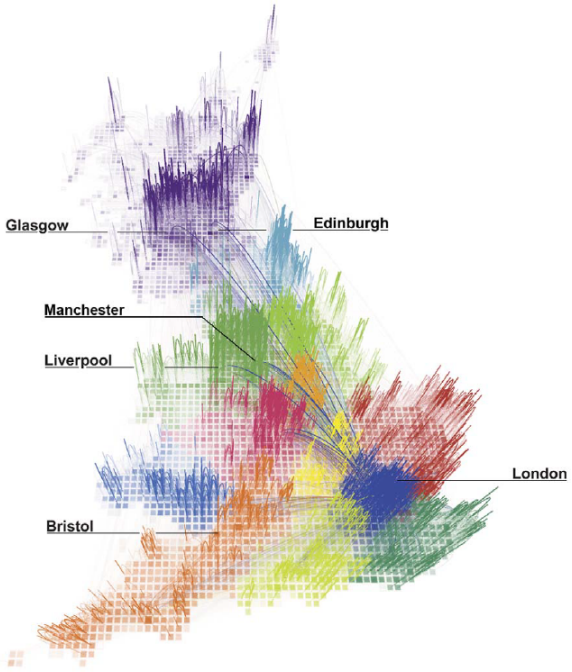


Figure 2.1: *Pure data map showing the geography of talk in Great Britain. Authors measured the total talk time via communication networks between areas in Britain and used the data to produce the visualzation. Source article: [13]*

Digital technology has also brought interesting methods to the art of crafting maps, allowing for new types of maps to be crafted, which brought previously undiscovered insights into the nature of represented territories. For example, with

data maps such as the one described by article [13] it is possible to draw more useful borders around regions, fitting actual human interaction groups as opposed to those defined by past governments, as shown in figure fig. 2.1. Maps like these are generated from vast data sets, collected and stored in databases - an approach that would not be possible without use of digital technology. Since maps can emerge out of pure data, the same approach can be used to create fictional maps, out of generated data.

The possibility of visualizing map layers which could not have been created and shown without digital data processing techniques and ease of experimentation with the information and algorithms used to create modern digital maps have also made it apparent that the source data for the layer itself do not necessarily have to measure some aspect of reality, but can be generated using mathematical methods. Such approach effectively allows for creation of fictional maps, representing imaginary territories. Moving on, the next section presents examples of fictional maps and their use in games, physical and digital alike.

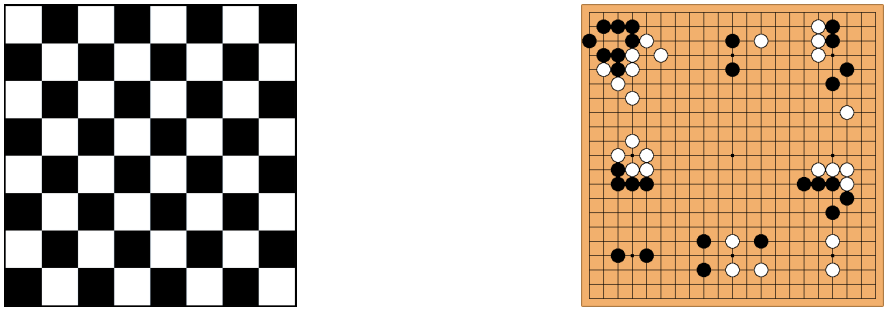
2.2 Maps in games

It is not clear what kind of game was the first one in history to use a map to represent the game world, however two notable examples may easily come into mind: Chess and Go, which are both widely known around the world.

Chess, a board tactical war-game developed before 6th century AD, uses black-white board as the map of its world. Although the environment represented by a chess board is very simple, it has some important features and rules. The map is composed of square cells, which are arranged on 8 by 8 grid, effectively creating a rigid boundary around the game world, which according to game rules - cannot be crossed. Each cell has 8 neighbours and can be occupied by only one game piece.

Another game, originating from ancient China, defines a similar, grid-based game world, effectively making a map of a uniform planar territory. Go is played on 19 by 19 board, however smaller board sizes are used as well. In Go, the goal is to capture more territory than the opponent, which is done by placing game pieces on line intersections, one piece per turn.

Another interesting example of a game world map is the multi-player strategy board game Risk, invented in 1957 by Albert Lamorisse, where the map represents a territory divided into regions, which must be captured by players in order to win. Risk game shows how a political world map with imagined region borders can be creatively used in a game, as a resource for the players to fight over. The game



(a) *Empty chess board* ***(b)*** *Go board during gameplay*

Figure 2.2: *Chess and Go both use planar boards divided into tiles by a square grid - simple maps to represent the environment in which game is played. Sources: Chessboard image - own work; Go board image -* `https://senseis.xmp.net/?LargeBoards`*, distributed under the terms of the Open Content License*

of Risk has been since published in many variations. Most of them share the same gameplay goal: to capture more territory than the opponents do, which is an example of how a game might use environments represented by maps as a limited resource for players to acquire.

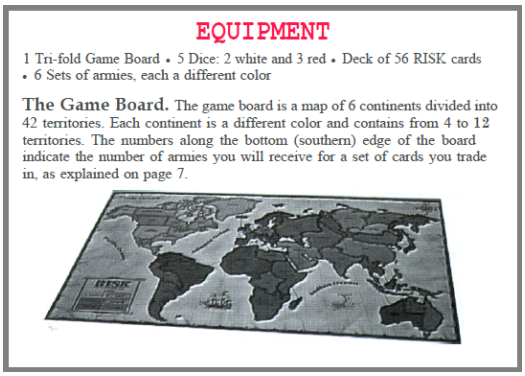


Figure 2.3: *Risk rule book fragment, containing a photo of the game board. The board shows a world map with fictional political borders, dividing the map into regions, which serve as a resource for players to capture. Image source: photo taken from original Risk rule book, copyright Hasbro 1993*

Modern board games have introduced many new ideas to the design of game boards. One example of such ideas is bringing modularity into the board design, composing pieces of the board similarly to how a jigsaw puzzle is composed of singular pieces. Such arrangement allows for greater value in replaying the game, since the game world can be different at each time the game is played.

An example of such game is Carcassonne, published in 2000, designed by Klaus-Jürgen Wrede in Germany. The game of Carcassonne involves an interesting mechanism: rather than having a fixed game board, cards with tiles are used to construct the board during gameplay. The game rules around tile placement can be thought of as an algorithm of procedural generation: only one tile can be placed during each game turn, adjacent to other tiles, forming a connection with features that tiles represent - roads must connect to roads, fields to other fields, and cities to cities. The rules ensure that the players will develop the game board as the game progresses, which leads to an interesting observation: each turn, the players are presented with new territory to consider in their decisions - and since the game requires players to deploy their resources onto the constructed game map in order to accumulate score and eventually win, these decisions may often become quite challenging with increasing complexity of the board layout. There is a web version of Carcassonne available at `https://concarneau.herokuapp.com/game`.

Figure 2.4: *Carcassonne game board during gameplay. Players place tiles on gameplay surface once per turn, making sure that each new tile is compatible to surrounding tiles. Image source:* `https://deerfieldlibrary.org/2016/01/carcassonne-a-modern-board-game-for-adults-teens/`

There are other board games which make use of generative mechanics to create or alter the board layout, before or during gameplay. Some of them rely on randomness, while others introduce rules by which the game world may be altered. An example of such rules is presented by Labirynth, a board game designed by Max Kobbert and first released by Ravensburger company in 1986. The game board in Labirynth represents a dungeon composed of tiles representing path elements: corners, straight paths, crossings. Each turn, the game board is altered by shifting an entire row or column of movable tiles in a direction chosen by player,

in effect changing the board layout.

Figure 2.5: *Labirynth game board. Even rows and columns are movable (marked with a yellow triangle on board edges). Image source:* `https://rulesofplay.co.uk/products/labyrinth`

While it is possible to find or develop interesting mechanics for board games, some more complex game rules and ideas are better implemented using computer simulation, where most of the mundane tasks which do not contribute to gameplay can be automated. Random number generation, board preparation and arrangement, checking player moves against game rules - all those activities are good candidates for automation. The other reason to simulate games may be to develop artificial intelligence algorithms which can simulate player behaviour at a chosen level of competitive play, effectively providing a way for beginners who want to learn the game they are interested in or for veterans who want to develop their skills further, as has been done for chess and other classic board games.

Among modern board games, there are many more examples which involve interesting mechanics, but their description lies beyond the scope of this thesis project. Next section investigates a few examples of computer games and simulations, where maps are used to construct some aspects of gameplay.

2.3 Interactivity and automation

The evolution of personal computers has allowed players to enjoy a new form of entertainment - video and computer games. Possibility of performing real-

time simulations on computers and development of computer graphics rendering techniques have created a new medium of expression in the form of computer software. At the time when early forms of interactive simulations were created, first computer games were also developed.

In the past decades, when the earliest computer games have been created, an industry focused on the craft of game development has emerged. The efforts of game designers and developers have lead to creation of multiple game genres and have driven the evolution of mechanics and challenges that modern games can now offer to players.

As stated in chapter 1, the context of this thesis does not deal with projections of 3D objects onto a plane, like the fields of geography and cartography do, as shown by works similar to [14]. The goal is to generate planar maps, which makes games that include them of particular interest for finding examples of working solutions to the problem.

Early examples involving procedurally created maps are Rogue and Nethack, dungeon crawling games developed in 1980s. Both examples generate a set of rooms with randomized dimensions, which are then connected to each other by a system of corridors, as shown in fig. 2.6.

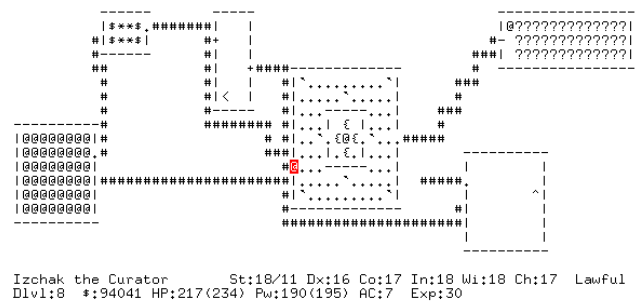


Figure 2.6: *An exemplary level, generated by Nethack game rules. Image source: Nethack project webpage,* `https://nethack.org/common/index.html`

Dungeon layouts generated by Rogue or Nethack could be described as suitable for representing indoor spaces, maps produced by connecting rooms are typically do not contain irregular shapes like landmass forms found in depictions of natural terrain. It is however, possible to use such mechanisms anyway. Generated levels found in Diablo series, Torchlight, Path of Exile and a few other similar action-RPG titles suggest that careful design of map components (textures, tiles, rooms) processed later by a generation mechanism can produce believable environments, regardless of what shapes are shown by a map of their layout.

2.4 Existing solutions for generating planar maps

There have been scientific surveys conducted on PCG (Procedural Content Generation) methods, which describe approaches to map generation employed in the past by successful game development projects. Contents of three such surveys are summarized in the following paragraphs.

Authors of the most general survey on PCG techniques [1] point to other works for deeper exploration into methods involving generative grammars, genetic algorithms and hybrid approaches for generating indoor spaces. For synthesizing outdoor maps, survey lists usage of image filtering, tiling, layering, fractals, Voronoi diagrams and cellular automata. In both cases, pseudo-random number generation and hybrid approaches are listed as tools which can be helpful in finding a promising solution.

Another general survey on PCG methods suitable for games [8] explores the subject of environmental content, listing and describing methods for generation of landscapes, continents, cityscapes, road networks and rivers. Terrain generated with methods presented in the survey has properties of naturally occurring land features with surface roughness resembling real environments. Authors list the usage of noise algorithms, L-systems, fractals and combinations of these methods along with modifications based on natural process simulations (e.g. erosion) to generate believable terrains. Described methods are categorized into *assisted* and *non-assisted* techniques, the former of which can be controlled by parametrization of inputs to achieve the desired outcomes and often require human support. Methods listed for generation of roads, rivers and cities involve more advanced and complicated techniques, such as the work of *Chen et al.* [15] on guiding the generation of road graphs with tensor fields, cited by the survey. Most of these methods are not suitable for this thesis project due to their complexity or are specifically designed for 3D environments.

Survey [16] focuses specifically on generation of dungeon levels, listing and describing how cellular automata, generative grammars, genetic algorithms and constraint-based methods are used to create maps containing features of indoor spaces.

The following sections shortly discuss the possibility of applying approaches found in described surveys to generation of planar maps with irregular shapes. Since the goal of this thesis is to build a generator which performs that process, the following sections focus on summarizing characteristics of cellular automata and generative grammars, as these approaches seem to be most promising for map generation purposes.

2.4.1 Cellular automata

A cellular automaton (CA) is a simulation in which every object in a mathematically defined space is being updated at every step of a simulation. Historically, cellular automata and their properties have been studied since the time of first computers [17]. One of the most complete sources on cellular automata is a book summarizing research carried out by Stephen Wolfram since 1980s [18], where a classification is shown along with examples for each kind of CA.

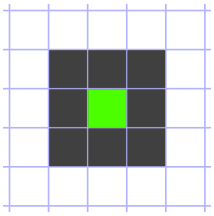
Specifically, 2-dimensional automata operate on a grid of cells with arbitrary discrete dimensions. Each cell in the grid has neighbours, which may be relevant to the simulation rules. Depending on the type of rules which are used by a particular CA, a different type of cell neighbourhood may be used. To present this concept concisely, a short list of definitions follows.

Cell Cells are units of state in CA simulation. Depending on CA type, they can be represented by simple values - a binary digit (0 or 1), an integer, a real or complex number with constraints, or other, more complicated value.

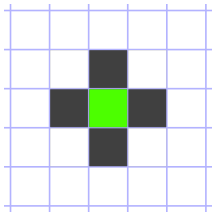
Cell neighborhood In a context of a 2D square grid of cells, neighbourhood is a collection of cells directly adjacent to the selected one.

Von Neumann neighbourhood Includes the cell and its immediate neighbours - one to the north, south, east and west of the cell, as shown in fig. 2.7.

Moore's neighbourhood Includes 8 closest neighbours of the cell - immediate and diagonal, as shown in fig. 2.7.



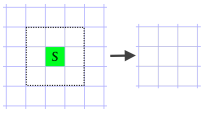
(a) *Moore neighborhood*



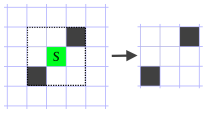
(b) *Von Neumann neighborhood*

Figure 2.7: *Two basic types of cell neighborhood. Source: own work*

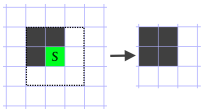
Every CA simulation uses rules which drive the process of cell evolution to its next stage. Typically, such rules define how board elements must be changed once a specific cell arrangement is recognized. The set of rules must cover all possible neighbourhood patterns of a cell so that the value of each cell in a future



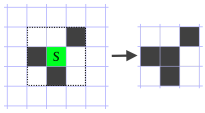
(a) *Rule 0 - cell is set to empty when no neighbours are present*



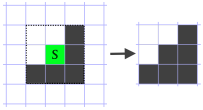
(b) *Rule 2 - cell is set to empty if there are less than three neighbours*



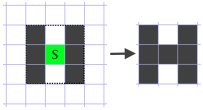
(c) *Rule 3 - cell is set to filled if it has three neighbours*



(d) *Rule 3a - same rule, behavior independent of neighbour positioning*



(e) *Rule 5 - cell is set to filled if more than 3 neighbours*



(f) *Rule 6 - cell is set to filled if more than 3 neighbours*

Figure 2.8: *A subset of rules for an unknown cellular automaton which uses Moore neighbourhood. Source: own work*

step is unambiguous. Exemplary rules for a 2D cellular automaton are presented in fig. 2.8, where included images do not cover all possible cases.

Observing the presented subset, it may be intuitive to see rules 3, 5 and 6 as alternate versions of the same rule that sets the cell state according to a simple test:

- if selected cell has less than 3 neighbours, it remains empty
- otherwise, it becomes filled

However, not knowing the full set of rules for this particular automaton, it cannot be concluded if those rules can be reduced in such way. For example, there might be rules for arrangements with 4, 7 or 8 neighbours which fill the cell instead of emptying it or rules that behave differently depending on neighbour positions. Without information about the rule set, one cannot prove which properties of the cell neighbourhood are measured during simulation, nor predict the results of automaton evolution. These properties may include the following:

- count of cells with state s in neighbourhood,
- state of selected cell,
- state of adjacent cells (values associated with each cell),

- positions of neighbours with state s

An example demonstrating such difficulty in reasoning about CA rules and their consequences is one of the classic cellular automata simulations, the Game of Life[1], invented by an American mathematician John Conway [19]. The simulation uses a square grid board, where cells can have one of two possible states: alive or dead. The rule set for Game of Life contains only three rules, based on counting alive neighbours of each cell:

- Alive cells with less than 2 alive neighbours become dead.
- Alive cells with 4 or more alive neighbours become dead.
- Dead cells with exactly 3 alive neighbours become alive.

Figure 2.9 shows an example of how these rules affect cells in a $5x5$ board during one simulation step. Cells outside the grid edges are assumed dead.

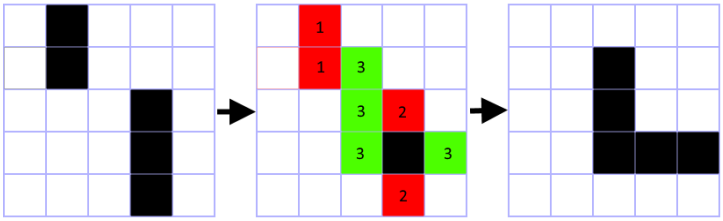


Figure 2.9: *Illustration of step transition in Game of Life. Alive cells which become dead in next turn are marked red, dead cells that become alive are marked green. Coloured cells were assigned a number equal to the count of their alive neighbours. Source: own work*

Despite being based on simple rules, Game of Life is able to generate complex visual patterns, such as those shown in fig. 2.10. While many of such cell arrangements have been found accidentally, there are organized efforts focused on finding and cataloguing them, such as the Life Wiki at `http://www.conwaylife.com/wiki/`.

Cellular automata have been used in map generation mechanisms for games, some of notable examples are Dwarf Fortress (2006), Minecraft (Mojang, 2011) and Ultima Ratio Regum (2011). Each of these games use CA-based algorithms to generate map layers, regional structures or other types of data used to describe the

[1] Some Game of Life simulation examples: one at MIT website `http://web.mit.edu/jb16/www/6170/gameoflife/gol.html` and another at `https://copy.sh/life/`

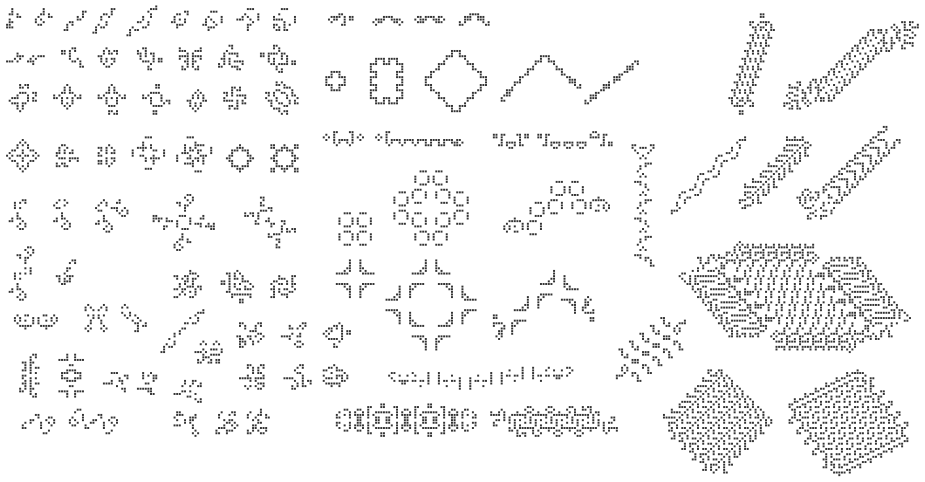


Figure 2.10: *A collection of patterns found in Game of Life, which oscillate with period of 3 simulation steps. Image source: adapted from* `http://copy.sh/life/?pattern=period3oscillators`

game world. Examples of maps found in Dwarf Fortress and Ultima Ratio Regum show that CA-based approach to map generation can indeed yield sufficient results. Dwarf Fortress starts the generative process with randomized fractals [20] to define various aspects of the data layers from which the final map is built, but also uses a 3D cellular automaton to simulate behaviour of fluids (water, lava). Minecraft uses fractal noise algorithms to generate an open world map along with other methods to determine the structure of terrain layers. Also, cellular automata are utilized to simulate fluid flow and possibly to propagate updates among certain world elements (blocks with state, simulated circuits).

However, in order to develop a map generator using CA, a rule set which achieves stable board states is needed. Game of Life simulations show quick, but impermanent changes in local areas of the board, while allowing for creation of occasional stable structures. They are however somewhat rare, as 2.9 shows, and may be easily broken by future simulation steps, which makes them insufficient to easily compose larger structures with randomized processes.

One of possible approaches to map generation which can create irregular shapes is the article written by L. Johnson, G. Yannakakis and J. Togelius from IT University of Copenhagen [21]. Authors describe rules of a cellular automaton which is able to transform a tile filled initially with random distribution of cells into a map tile with desired properties which can later be merged with other tiles into the game map. Authors define types of cells to indicate which areas of

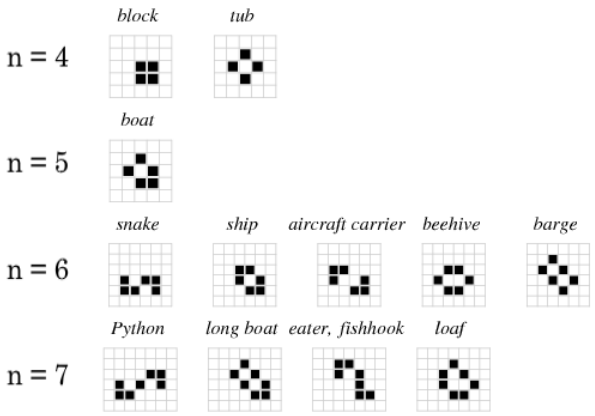


Figure 2.11: *A collection of small ($n < 8$, n - number of cells in a pattern) still life patterns from Game of Life. Image source: adapted from* `http://mathworld.wolfram.com/GameofLife.html`

the map are accessible (floor cells) and which are not (rock and wall cells). To achieve better control over the generation process, authors altered the following parameters:

- r - percentage of floor cells in initial state,
- n - count of CA generations (steps) to perform,
- T - neighbourhood threshold value which defines inaccessible cells,
- M - Moore neighbourhood size

As described in the article, results depend on automaton rules and parameters selected to transform the tile into derived states. Authors claim that it is possible to create structure for playable maps with this approach. The article contains images of generated tiles which show irregular, cave-like shapes and provides an illustration of a map constructed from them, with a roughly shaped area with a number of narrow connections between grander spaces.

2.4.2 Shape grammars

Generative grammars are systems of rules and transformations which show ability to synthesize constructs out of a predefined set of symbols. Grammar theory has

been developed in the field of linguistics where Noam Chomsky laid the foundations in 1950s. Development of formal language theory as a branch of applied mathematics that involves study of formal languages and grammars has led to creation of many grammar subtypes and variations [22]. They have been later adapted by computer science for use in syntax analysis algorithms, compilers, debuggers, parsers and other applications.

One of specific grammar subtypes are shape grammars, applied to a certain class of problems in architecture, decorative arts and industrial design. Research on grammars performed by G. Stiny and J. Gips [23], [24] and other scholars has been proven useful to computer scientists for development of interactive design frameworks [25] among other applications.

Shape grammars can be utilized as a formal foundation for PCG techniques and extended with probabilistic aspects [26] to generate content for games. A domain specific language has also been developed on the foundations of grammar theory [27] for such purposes with success, as presented by fig. 2.12 adapted from the cited work.

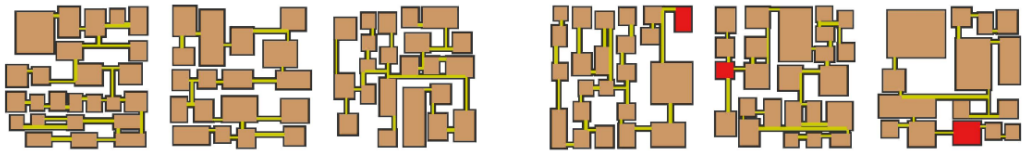


Figure 2.12: *Game levels generated with grammatical PCG. Image source: Work of Tiannan Chen, Stephen J. Guy on Grammatical Item Generation Language (GIGL) [27]*

As a system of formal rules for transformation of shapes into more complex constructs, shape grammars are a viable candidate for a method to generate game maps, limited only by shapes and transformation rules available to the generator.

2.5 Chosen method: cellular automata

Since this thesis is focused on creation of maps through procedural processes, it is worth noting why cellular automata have been chosen as the method to develop a solution. The following arguments show why CA might be a useful approach:

- as shown by [21], CA are sufficient for map generation
- 2-dimensional CA can be implemented in a similar way to some image processing algorithms (e.g. erosion, dilation)

- CA are able to model evolution of dynamical systems (e.g. fluids), which may be useful to create maps of terrain similar to natural landmass forms
- amount of repetition in shapes produced by grammars depends on how complex the set of transformations is [28], while CA can generate surprising, chaotic results with simple rules [29]

Although grammars are favourable for clarity of transformation rules they bring to the process of map generation, CA can be a viable alternative, providing irregularity and surprise to possible results, as proven by the work of A. Khalifa and J. Togelius [30] - which is why they have been chosen for experimentation in this thesis.

Chapter 3

Generating and visualizing maps - proposed solution

To describe the developed solution, this chapter consists of five sections: definition of required features, design of simple CA simulation and map generator models, followed by implementation in C++ and description of performed experiments.

3.1 Analysis of requirements for a map generator

The approach chosen in chapter 2 can be imagined as a process that consists of several steps. Starting with generation of a board with random cell states, which after several transformations performed by CA rules becomes structured with irregular, island-like features. The resulting board is then used as one of tiles to be merged into the larger map. Each of these steps must be automated while allowing the designer a degree of control over the generation process. Transformations of the board should be applied according to parameters set by the designer, which must be constrained to ranges that do not allow creation of obviously unusable results (e.g. board filled with cells of identical state). Usable ranges of parameters can be determined with experimentation.

To formalize what map generator must do, required features and behaviours can be now extracted from the paragraph above, as follows:

Functional requirements

The map generator program shall perform the following functions:

- prepare square tiles with random data for transformation,
- simulate a cellular automaton to generate irregular shapes in tiles,
- have a mechanism to merge tiles into a bigger map,
- show tile states graphically,
- have controls to allow tweaking CA parameters,

Operational requirements

The map generator program shall operate with the following qualities:

- real time drawing tile and map images, whenever their state changes
- responsive user controls, avoid long waiting times for computation result

3.2 Design and implementation

In chapter 3, cellular automata have been chosen as a method for generating maps, so it is helpful to find resources on the topic of building cellular automata simulations. One of them is chapter 7 in Nature of Code, a book by Daniel Shiffman [31], where one can find a short tutorial on building a CA simulation. Author describes elementary concepts needed to construct a basic CA, explains how to implement a working simulation and provides helpful exercises. Another source, a book by Stephen Wolfram [18] shows advanced research, exploring concepts and organizing knowledge on cellular automata.

3.2.1 Basic cellular automaton

The initial step required to implement a generator based on cellular automata is to create a framework for performing CA simulations. As stated in Nature of Code [31], a 2-dimensional CA needs the following key elements:

- Cell state - every cell has a state updated on each simulation step,
- Grid - a space on which cells are placed,
- Neighbourhood - each cell needs to know the state of its neighbours to update its state.

Data structures

In order to represent state of a cell, a primitive data type is sufficient. Class model in fig. 3.1 presents an abstraction that can encapsulate a collection of cell states, along with two operations: retrieving the state of a cell in position (x, y) from a grid and setting its state to a chosen value.

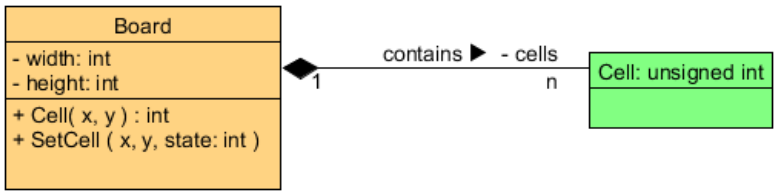


Figure 3.1: *Model of a Board class, which holds cell states in its block of memory and lets its user change their states. Diagram source: own work*

Operations

While the *Board* class is sufficient to represent the grid of cells and their states, the concept of a neighbourhood is missing. It can be captured and represented by creating a procedure that selects a cell, and treats its neighbours as if an identifier is assigned to each of them, as shown in table table 3.1.

0	1	2	$(-1,-1)$	$(0,-1)$	$(1,-1)$
7	S	3	$(-1,0)$	$(0,0)$	$(1,0)$
6	5	4	$(-1,1)$	$(0,1)$	$(1,1)$

Table 3.1: *Table on left: Moore neighbourhood, with numbered cells. S denotes cell selected as a base for further operations. Table on right: Neighbour positions, relative to selected cell* $(0,0)$ *Source: own work*

In CA simulations used for map generation by [21], summing the values of cells in neighbourhood is a required operation. Hence, it is useful to include it into the *Board* abstraction as part of its interface, which yields a class presented on fig. 3.2 with added method for computing sum of adjacent cell values around a cell with (x, y) coordinates. The *nht* parameter can be used to control which cells are treated as neighbours.

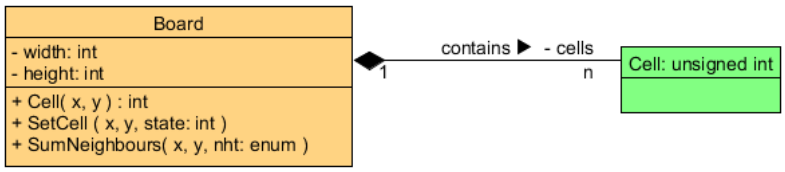


Figure 3.2: *Model of a Board class with a method to sum neighbour values. Diagram source: own work*

Implementation

The model presented above is implemented by the *Board* class. It allows to simulate some basic cellular automata based on rules which require a sum of values in cells adjacent to the selected cell to decide its future state. *Board* class methods operate as follows:

- *CellAt* can access cell at position (x, y) in the grid model and return its value, as shown on listing 3.1,
- *SetCellAt* works similarly to the previous method, accessing the cell at position (x, y) and writing a new value provided by *state* parameter, shown on listing 3.2,
- *SumNeighbours* computes a sum of cell values in a neighbourhood of type *nht* at position (x, y), as shown on listing 3.3

Listing 3.1: *Accessing cells*

```
1 CELL_t CellAt(unsigned int x, unsigned int y)
2  {
3    return cells.at(cellsX * (y%cellsY) + (x%cellsX));
4  }
```

Listing 3.2: *Setting cell states*

```
1 void SetCellAt(unsigned int x, unsigned int y, CELL_t newState)
2  {
3    cells.at(cellsX * (y%cellsY) + (x%cellsX)) = newState;
4    isBoardChanged = true;
5    return;
6  }
```

Listing 3.3: *Summing adjacent cell values*

```
unsigned int SumMooreNhd(unsigned int x, unsigned int y, int rad)
 {
   unsigned int sum = 0;
   for (int nx = −rad; nx <= rad; nx += 1) {
     for (int ny = −rad; ny <= rad; ny += 1) {
       sum += CellAt(x + nx, y + ny);
     }
   }
   return sum − CellAt(x + 0, y + 0);
 }
```

3.2.2 Simulating CA rules

As described in chapter 2, basic CA rules check the values of adjacent cells to a selected cell and determine its subsequent state. Separating the concept of a rule and its application from the cell grid structure simplifies modifying the CA rule set for further usage.

Data structures

Although applying rules to a grid of cells does not require a specialized data structure, a class to encapsulate procedures used to do so separated from *Board* class can be useful for later development. The *Ruleset* class presented in fig. 3.3 is used to produce a new board state by applying rules encoded in its internal functions, which are used by *EvolveState* method.

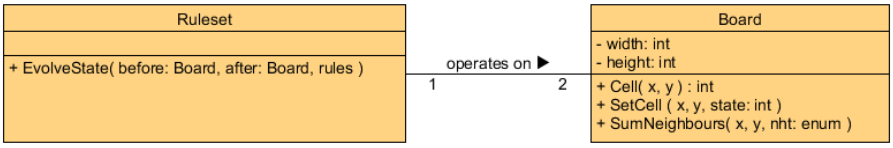


Figure 3.3: *Model of Ruleset class, which is meant to use a Board instance to produce a new state of the cell grid in another Board instance by rewriting its cell values. Source: own work*

Such abstraction allows to later add an internal data structure to hold separate rules for each possible neighbourhood pattern, if needed.

Operations

As shown in fig. 3.3, the operation which handles computation of a new cell grid state is named *EvolveState*(). It does so by calling one of its internal methods chosen by *rules* parameter. Listing 3.4 shows an implemented method which calls *Board* methods to perform the following steps:

1. sum the values of cells adjacent to cell (x, y),
2. set the future (x, y) cell state to 1 if the sum is less than 5,
3. set the future (x, y) cell state to 0 if the sum is more than 5.

Listing 3.4: *Applying rules to cell grid*

```
1  static void Rules_MapGen(Board *before, Board *after)
2   {
3     Board::isMarkingEnabled = true;
4     for (unsigned int x = 0; x < before->cellsX; x++)
5     {
6       for (unsigned int y = 0; y < before->cellsY; y++)
7       {
8         unsigned int sum = before->SumNeighbours(x, y, MOORE8);
9         if (sum < 5) after->SetCellAt(x, y, 1);
10        if (sum > 5) after->SetCellAt(x, y, 0);
11      }
12    }
13    return;
14  }
```

With basic CA operations organized this way, adding new rules requires implementing a new method in the *Ruleset* class. Choosing a new set of rules is done via passing a correct parameter value to *EvolveState* method, which can be done by binding the function call to user interface.

3.2.3 Presenting the board state

In order to visualize state of *Board*, all cell values need to be drawn as an image. To achieve this, OpenGL and ImGui libraries are used to create an environment in which it is possible to construct interface elements and render textures.

Data structures

Before the cells can be represented as an image, their states must be converted to colour values in one of the formats interpreted by OpenGL functions. In order to do so, a class *SimpleTexture2D* has been prepared, as shown in fig. 3.4.

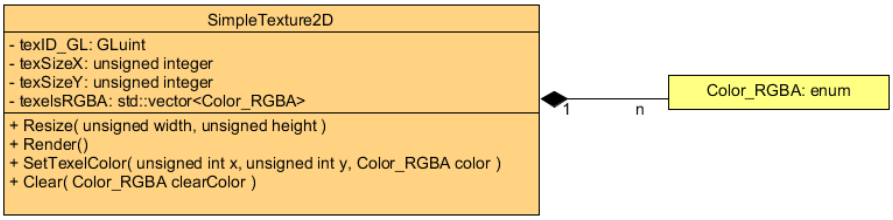


Figure 3.4: *Model of a SimpleTexture2D class, which contains a collection of ColorRGBA elements. Source: own work*

Colours are represented by a 32-bit unsigned integer, with 8 bits dedicated to each colour channel (Red, Green, Blue) and additional 8 bits for the opacity (Alpha) channel, defined by type *ColorRGBA* as shown in listing 3.5.

Listing 3.5: *Definition of ColourRGBA type*

```
1 typedef GLuint Color_RGBA;
2
3 const Color_RGBA color_BLACK = 0x000000CC;
4 const Color_RGBA color_WHITE = 0xFFFFFFCC;
5 const Color_RGBA color_RED = 0xFF0000CC;
6 const Color_RGBA color_GREEN = 0x00FF00CC;
7 const Color_RGBA color_BLUE = 0x0000FFCC;
```

In order to allow the program user a degree of control over the generation process, a rudimentary user interface has been prepared. Figure 3.5 shows the model of its classes. The class named *UserInterface* acts as an update and rendering controller for each of the windows represented by the other classes in the model, which can call functions bound to UI controls they contain and show a visual representation of a *Board* or *Map* by using the data encapsulated by *SimpleTexture2D*.

Operations

Methods provided by *SimpleTexture2D* class allow to:

- set individual colour values at specified position in the *texelsRGBA* collection, necessary for rendering the *Board* state

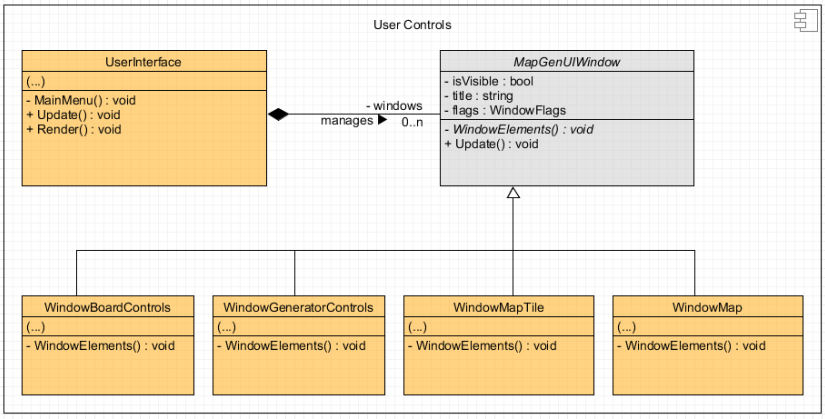


Figure 3.5: *UI class model in map generator program. Source: own work*

- change the collection size, to allow support for *Board*s of different sizes,
- set all colour values in the collection to a specified value,
- request rendering of the contained data by OpenGL API.

Such set of operations allows to achieve the goal of simple 2-dimensional image rendering, which can be used to visualize the state of all cells contained in one of the *Board* class instances.

Drawing cell states in tile generation and map grid windows is achieved by the *Board* class method, $DrawCellsToTexture()$, described previously. Additionally, *Map* provides operations for drawing its state depending on the selected view mode - $DrawMap()$ and $DrawTileAt()$.

3.2.4 Generating and merging tiles into maps

Generating tiles for map construction can be performed with methods implemented by $TileGenerator$ and $Ruleset$ classes. Internal mechanics of the $TileGenerator$ class allow it to initialize cells in the first generation to randomly selected states and then to use $Ruleset$ class to generate future states of the board. $Ruleset$ provides a set of rules which describe the possible transformations of a cell depending on the state of its neighbourhood, as described in previous sections. Tiles prepared by $TileGenerator$ can be transferred into the grid of tiles represented by the *Map* class.

Data structures

The map construction process is simulated by *TileGenerator* and *Map* classes. An ordered collection of *Board* states is contained in both of them, but for different purposes. *TileGenereator* uses the collection as a history of *Board* state transformations. *Map* uses its collection of *Board* instances as a tile grid representation. Figure 3.6 shows how these classes relate to each other.

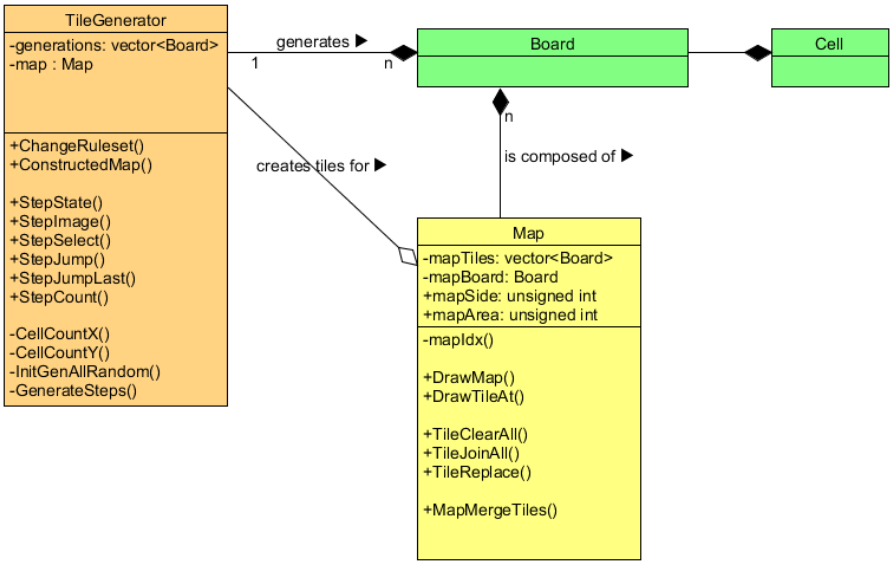


Figure 3.6: *A model of interactions between TileGenerator, Board and Map classes. Source: own work*

The *Map* class represents its state in two ways - as a collection of *Board* states and as a single *Board* instance with dimensions equal to tile width and height multiplied by their count along the Map borders - for example, a square map composed of 25 tiles with 64 by 128 size would have a width of 320 and a height of 640. Such dual representation allows the user to switch between two views onto the map - a tile grid view and a complete map view.

Generating tiles

Initializing the first *Board* contained in *TileGenerator* class with random states enables the generator to use it as a base for further transformations, while also ensuring that each generated tile will be different, which is implemented by *InitGenAllRandom* method contained in *TileGenerator*.

Listing 3.6: *Tile generation rules*

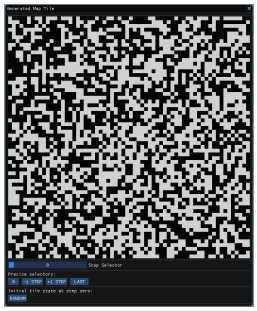
```
static void Rules_MapGen1(Board *before, Board *after)
{
  Board::isMarkingEnabled = true;
  for (unsigned int x = 0; x < before->cellsX; x++)
  {
    for (unsigned int y = 0; y < before->cellsY; y++)
    {
      unsigned int sum = before->SumMooreNhd(x, y, 1);
      if (sum < 5) after->SetCellAt(x, y, 1);
      if (sum > 5) after->SetCellAt(x, y, 0);
    }
  }
  return;
}
static void Rules_MapGen2(Board *before, Board *after)
{
  Board::isMarkingEnabled = true;
  for (unsigned int x = 0; x < before->cellsX; x++)
  {
    for (unsigned int y = 0; y < before->cellsY; y++)
    {
      unsigned int sum = before->SumMooreNhd(x, y, 2);
      if (sum > 12) after->SetCellAt(x, y, 1);
      else after->SetCellAt(x, y, before->CellAt(x, y));
    }
  }
  return;
```

After the initial state of the *Board* is prepared, $TileGenerator$ can then use $GenerateSteps()$ method to create *Board* states following the initial one by applying operations defined in and selected by current $Ruleset$. Listing 3.6 shows functions which transform cells, of which the first one applies following rules:

- every cell that has more than 5 neighbours will be set to *wall* state,
- every cell that has less than 5 neighbours will be set to $floor$ state.

Additional sets of rules should be defined similarly, by adding more methods to the $Ruleset$ class.

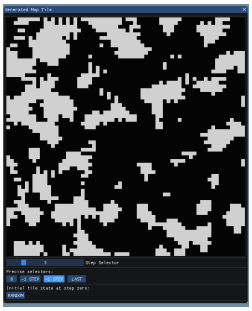
Generated tiles are saved in an ordered collection, to allow viewing the history of generation steps. One possible result of such process is presented by section 3.2.4. The rules used to generate these tiles show an interesting property - after a few initial steps, values of most cells begin to alternate between $floor$ and *wall* states.



(a) *Step 1 - random noise*

(b) *Step 2*

(c) *Step 3*



(d) *Step 4*

(e) *Step 5*

(f) *Step 6 - partially balanced tile, rough edges*

(g) *Step 7*

(h) *Step 8 - balanced tile, smooth edges*

(i) *Step 9 - further transformation*

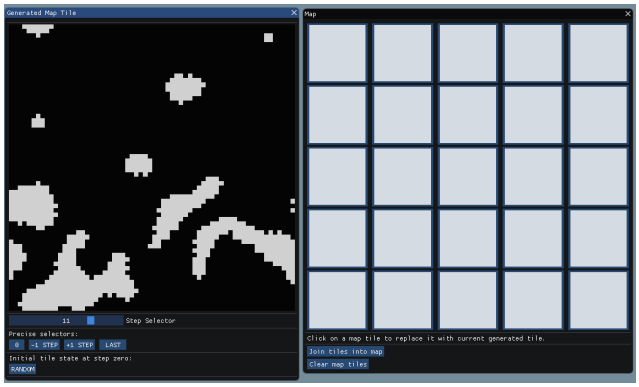
Figure 3.7: *Stages of tile generation. Source: own work*

Such emergent condition can be thought of as a *balancedstate* of a tile, since rules seemingly cannot further modify the tile after a sufficient number of transformations. It is however possible that map designers may decide to use a tile before it becomes *balanced*. Furthermore, exploring different sets of rules for tile generation may reveal better rules for creation of useful tiles. Amount of time required to find them can be reduced by providing a user interface for rule definition and management.

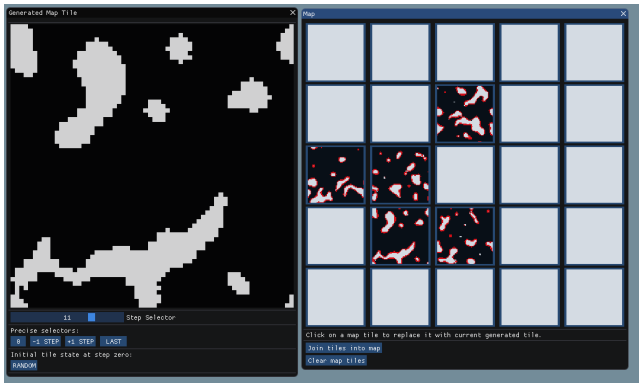
Merging tiles

The process of merging generated tiles into a map consists of several steps and begins with the decision to transfer a tile from the generator to map construction window.

Copying a tile state from the *TileGenerator* collection to the *Map* grid requires user interaction. Any tile in the window showing current *Map* state can be clicked by the user, which invokes the *TileReplace*() method of the *Map* class, effectively copying the tile state visible in the tile generation window to the map grid.



(a) *Initial state - blank map, no tile was placed*



(b) *Partially filled map - after copying 4 prepared tiles*

Figure 3.8: *Tile and map windows used to construct maps from generated tiles. Source: own work*

Once the map designer decides that the tile prepared in the generation window is a sufficient candidate to be used in a map, its state can be copied to the map grid by clicking a square slot in map window. Such action overwrites all cell states

in selected space with those contained in the prepared tile. Figure 3.8 shows two situations, where map designer starts with a blank map and proceeds to fill its grid with some of the tiles that were generated.

To complete the map construction process, designers can then switch the map representation mode, which is done by function bound to button labelled as `Join tiles into map`. Doing so also reveals other buttons provided by *Map* window, which can be used to make further adjustments. Switching the mode effectively copies the cells from each tile formerly placed in the grid into a bigger *Board* and renders its visualization instead of the tile grid.

3.3 Experiments in generating maps with CA

After implementing the map generator, it is possible to carry out experiments to find the optimal parameter ranges for map generation.

The first experiment carried out using the generator was to build a 5x5 map out of tiles generated with following number of CA steps:

- outer ring (edge) - 2 steps,
- inner ring tiles - 5 steps,
- central tile - 11 steps.

Rules used to generate tiles for the map were identical to rules used before to generate tiles in section 3.2.4. Cells with more than 5 adjacent walls are filled and those with less than 5 are emptied.

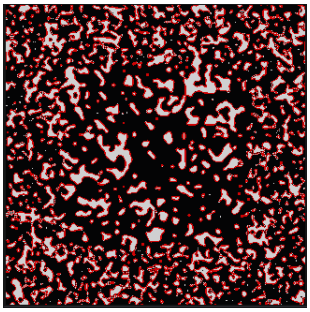
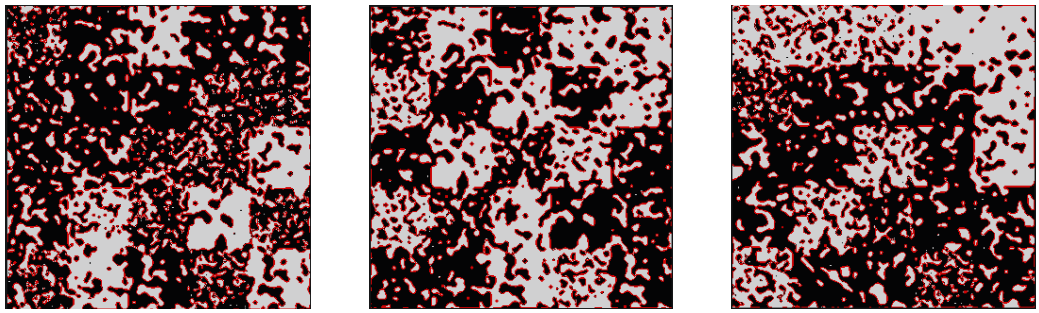


Figure 3.9: *Map constructed from a 5x5 grid of generated tiles. Source: own work*

The resulting map shown in fig. 3.9 has following properties:

- impassable areas are encountered more frequently as distance from central tile increases,
- impassable areas have greater sizes closer to central tile,
- there are no obstacles connecting those areas to form maze-like structures.

Other experimental maps, shown in fig. 3.10 illustrates that controlled placement of tiles produces imbalanced and asymmetric forms. However, a problematic property emerges when tiles placed in the map are packed with wall cells. Edges of those tiles produce straight lines which are not modified at any step of the performed transformations. A possible cause is that the active rule set did not modify cells which have exactly 5 adjacent walls.



(a) ***(b)*** ***(c)***

Figure 3.10: *Other experimental maps: 5x5 grid, tile placement controlled by user. Source: own work*

Another experiment, shown in fig. 3.11, uses rules where the radius of Moore neighbourhood is increased ($r = 2$) and walls are placed whenever there are more than 12 adjacent walls to the central cell. In this case, islands produced by transformation are not completely solid, which may be a useful feature to map designers.

Comparing fig. 3.11 with fig. 3.12 shows how removal or addition of one rule may affect shapes found in generated tiles. In closing: images in this section illustrate the utility of employing CA for generating map assets, however finding the right rules and parameters may require additional trials.

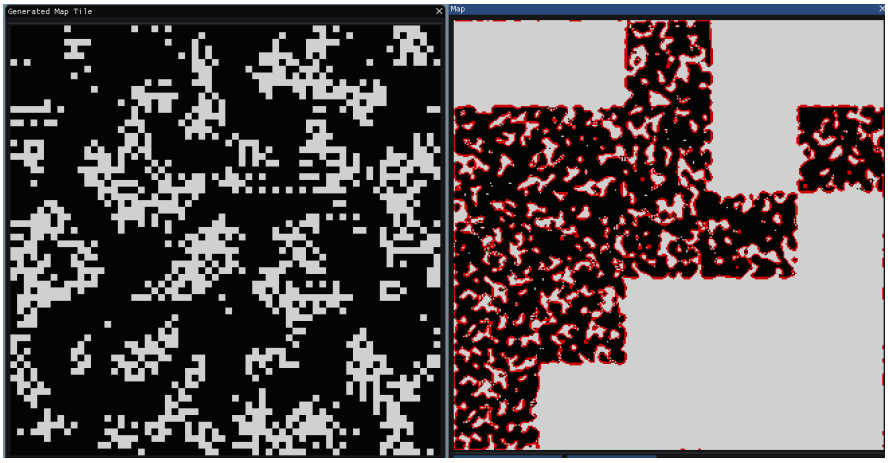


Figure 3.11: *Left: one of generated tiles with only one transformation rule: if $n > 12$, set cell to wall state. Right: construction of experimental map on 5x5 grid. Source: own work*

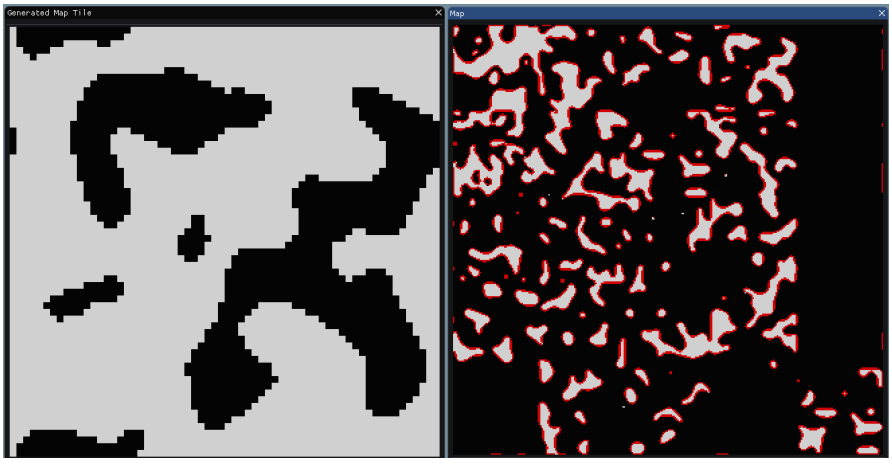


Figure 3.12: *Another experiment with map construction. Two rules applied with neighbourhood of $r = 2$, placement of walls for $n > 12$, floor for $n < 12$. Source: own work*

Chapter 4

Conclusions

The aim of this thesis was to provide a software solution which uses cellular automata for constructing maps. Although the goal of creating a program for map construction has been achieved, questions stand:

- Are cellular automata useful for generation of interesting, surprising and unique maps?
- What kind of improvements might be necessary to improve the generator and its mechanisms to satisfy map designers?
- Is the developed solution sufficient for use in production projects?

4.1 Results

Observing the results presented in figures produced by implemented generator reveal that cellular automata can be used for generation of irregular, surprising shapes. Tiles generated by simple rules from random noise have unique layouts, most of them are useful for constructing a map. However, whenever tiles are packed with impassable cells, merging them together produces unnatural, straight edges. A better method for merging tiles is required to eliminate those effects.

A degree of control achieved by introducing graphical user interface to the solution has provided a visual representation of how each tile evolves and an ability to select tiles with desired shapes for the map being constructed. Widgets of the interface provide control over some of generation parameters, which is useful for experimentation. Including new widgets for controlling other possible parameters and defining rules for the automaton would be vital for exploring what other kinds of shapes can be generated with cellular automata.

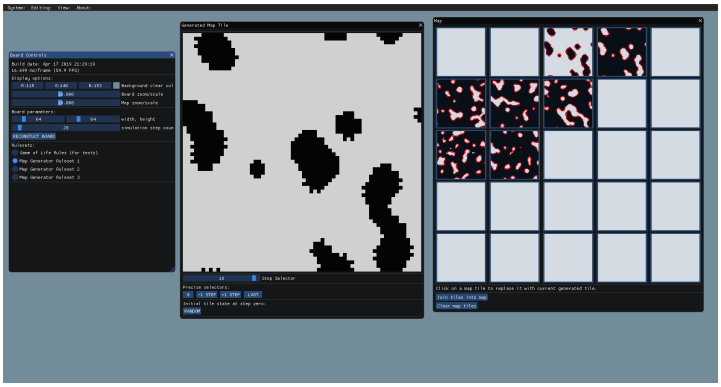


Figure 4.1: *The interface of developed Map Generator program. Source: own work*

The developed map generator tool shown in fig. 4.1 was sufficient for experimentation with generated tiles. Its production value is still limited without an easy to use interface for automation generative processes and asset export mechanisms. Adding them to the solution carries a further development cost which involves redesign of the program model and source code refactoring to add features proposed in the next section.

4.2 Future extensions for production environments

However, in order to adapt it in an environment where generated maps will be used as game assets, the following extensions are provided for consideration:

Tile editing and custom rule sets

A potentially useful feature would be manual editing of tiles before they are placed in the map, which would allow for an even greater degree of control over tile contents. Further expansion could be to provide the designer with tools to make their own rules for tile generation, while also allowing them to define types of single cells composing the tile.

Computation cost of tile generation

Since rules required for tile generation are applied to every cell in the tile grid identically, cells do not need to be updated in sequence. A possible improvement would be implementing cell state updates to be applied in parallel, which may reduce the time needed to compute a simulation step. One way to do so would be

to apply the findings presented by Reno Fourie in his thesis about applying CUDA technology in 2-dimensional cellular automata simulations [32].

Exporting results of generation

Although exporting maps is not necessary for experimentation and testing, a production version of the program should have such functionality. A map generator without a mechanism for saving the work done by a designer would certainly not be a useful tool. Such program needs a way to export generated maps to a file format that later can be used by a game engine of choice, possibly with procedures written by other programmers.

Automation

Since the goal of including procedural generation techniques into game production is often motivated by reducing the human effort needed to create assets, including tools for automating the process would be a welcome addition. Level designers would benefit from having a generator that eliminates the need for performing repetitive tasks, while still providing map layouts ready to be filled with other game assets and objects.

Cell states

Developed solution relies on a 2D cellular automaton with cells containing a binary state, which are sufficient for creation and interpretation of a map defining only which areas are passable and which are not. Game designers may however require additional map features to be generated. At that point, future experiments may involve other types of cell state like integers or non-discrete numbers might reveal other interesting results, especially with an interface for rule definition.

4.3 Acknowledgements

The author of this thesis would like to express gratitude with special thanks to those who have encouraged him to continue the project, for their continued support and kindness.

Bibliography

[1] Mark Hendrikx et al. “Procedural content generation for games: A survey”. In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9.1 (2013), p. 1.

[2] George Kelly and Hugh McCabe. “Citygen: An interactive system for procedural city generation”. In: *Fifth International Conference on Game Design and Technology*. 2007, pp. 8–16.

[3] Sylvain Lefebvre and Fabrice Neyret. “Pattern based procedural textures”. In: *Proceedings of the 2003 symposium on Interactive 3D graphics*. ACM. 2003, pp. 203–212.

[4] Ruben M Smelik et al. “A survey of procedural methods for terrain modelling”. In: *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*. 2009, pp. 25–34.

[5] Alexandru Iosup. “Poggi: Puzzle-based online games on grid infrastructures”. In: *European Conference on Parallel Processing*. Springer. 2009, pp. 390–403.

[6] Julian Togelius et al. “Search-based procedural content generation: A taxonomy and survey”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (2011), pp. 172–186.

[7] Georgios N Yannakakis and Julian Togelius. “Experience-driven procedural content generation”. In: *Affective Computing and Intelligent Interaction (ACII), 2015 International Conference on*. IEEE. 2015, pp. 519–525.

[8] Daniel Michelon De Carli et al. “A survey of procedural content generation techniques suitable to game development”. In: *Games and Digital Entertainment (SBGAMES), 2011 Brazilian Symposium on*. IEEE. 2011, pp. 26–35.

[9] Rafael Bidarra et al. “Integrating semantics and procedural generation: key enabling factors for declarative modeling of virtual worlds”. In: *Proceedings of the FOCUS K3D Conference on Semantic 3D Media and Content, Sophia Antipolis-Méditerranée, France*. 2010.

[10] Ruben Smelik et al. “Integrating procedural generation and manual editing of virtual worlds”. In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM. 2010, p. 2.

[11] Ruben Michaël Smelik et al. “A declarative approach to procedural modeling of virtual worlds”. In: *Computers & Graphics* 35.2 (2011), pp. 352–363.

[12] L. Bagrow. *History of Cartography*. Taylor & Francis, 2017. ISBN: 9781351515580.

[13] Carlo Ratti et al. “Redrawing the Map of Great Britain from a Network of Human Interactions”. In: *PLOS ONE* 5.12 (Dec. 2010), pp. 1–6. DOI: 10.1371/journal.pone.0014248. URL: https://doi.org/10.1371/journal.pone.0014248.

[14] J.P. Snyder. *Flattening the Earth: Two Thousand Years of Map Projections*. University of Chicago Press, 1993. ISBN: 9780226767468. URL: https://books.google.pl/books?id=MuyjQgAACAAJ.

[15] Guoning Chen et al. “Interactive Procedural Street Modeling”. In: *ACM Trans. Graph.* 27.3 (Aug. 2008), 103:1–103:10. ISSN: 0730-0301. DOI: 10.1145/1360612.1360702. URL: http://doi.acm.org/10.1145/1360612.1360702.

[16] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. “Procedural generation of dungeons”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.1 (2014), pp. 78–89.

[17] Palash Sarkar. “A Brief History of Cellular Automata”. In: *ACM Comput. Surv.* 32.1 (Mar. 2000), pp. 80–107. ISSN: 0360-0300. DOI: 10.1145/349194.349202. URL: http://doi.acm.org/10.1145/349194.349202.

[18] Stephen Wolfram. *A new kind of science*. Vol. 5. Wolfram media Champaign, 2002.

[19] John Conway. “The game of life”. In: *Scientific American* 223.4 (1970), p. 4.

[20] Tarn Adams. “Simulation principles from Dwarf Fortress”. In: *Game AI Pro* 2 (2015), p. 549.

[21] Lawrence Johnson, Georgios N Yannakakis, and Julian Togelius. “Cellular automata for real-time generation of infinite cave levels”. In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM. 2010, p. 10.

[22] M. A. Harrison. *Introduction to Formal Language Theory*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1978. ISBN: 0201029553.

[23] George Stiny. “Introduction to Shape and Shape Grammars”. In: 1980.

[24] George Stiny and James Gips. “Shape Grammars and the Generative Specification of Painting and Sculpture.” In: *IFIP Congress (2)*. Vol. 2. 3. 1971.

[25] Minh Dang et al. “Interactive Design of Probability Density Functions for Shape Grammars”. In: *ACM Trans. Graph.* 34.6 (Oct. 2015), 206:1–206:13. ISSN: 0730-0301. DOI: 10.1145/2816795.2818069. URL: http://doi.acm.org/10.1145/2816795.2818069.

[26] Francesco Sportelli, Giuseppe Toto, and Gennaro Vessio. “A Probabilistic Grammar for Procedural Content Generation”. In: July 2014. DOI: 10.13140/2.1.3820.4163.

[27] Tiannan Chen and Stephen J. Guy. “GIGL: A Domain Specific Language for Procedural Content Generation with Grammatical Representations”. In: *AIIDE*. 2018.

[28] Noam Chomsky. “On certain formal properties of grammars”. In: *Information and Control* 2.2 (1959), pp. 137 –167. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(59)90362-6. URL: http://www.sciencedirect.com/science/article/pii/S0019995859903626.

[29] Stephen Wolfram. “Cellular automata as models of complexity”. In: *Nature* 311.5985 (1984), p. 419.

[30] Ahmed Khalifa and Julian Togelius. “Marahel : A Language for Constructive Level Generation”. In: 2017.

[31] Daniel Shiffman. *The Nature of Code: Simulating Natural Systems with Processing*. Daniel Shiffman, 2012.

[32] Ryno Fourie. “A parallel cellular automaton simulation framework using CUDA”. PhD thesis. Stellenbosch: Stellenbosch University, 2015.

List of Figures

List of Tables

Attachments

1. TO DO: include thesis defence documents
2. TO DO: attach 1
3. TO DO: attach 2
4. TO DO: attach 3