Generating two-dimensional game maps with use of cellular automata

Michał Wolski

June 14, 2018

Contents

Chapter 1

Introduction

During recent years, presence of computer games in human lives has increased. The demand for games has shown that playing games, both as a medium of expression and a means for entertainment, is a desirable form of activity. However, as the demand for games rises [1] and computer games become increasingly complex, demand for game content must also rise – game 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 [Hen+13] show where the evidence for insufficiency of manual content creation may be found. In the study, authors point to work of Kelly and McCabe [KM07], Lefebvre and Neyret [LN03], Smelik et al. 2009 [Sme+09] and Iosup 2009 [Ios09] as sources which reveal game content production as a time-consuming and expensive endeavour.

Solving the inefficiency issue

Scientific surveys such as [Hen+13] and [Sme+09] show why investigating procedural generation 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 [Hen+13], [Tog+11] and [DC+11]

[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`

show what types of game content can be generated and are a good starting point for seeking methods of procedural generation.

Personal motivation

During two recent years, the author of this thesis took part in a small, after-hours independent game development project. Working with a group of friends, using Unreal Engine as a tool to develop a simple prototype of a game belonging to the *rogue-like* game genre. The project is still in development phase and finding a good method of map generation can potentially result in contribution of useful features.

1.1 Thesis structure

The overall structure of this thesis includes introduction followed by three chapters. The second 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. The chapter 3 describes performed experiments, design and implementation of a solution to the problem. Chapter 4 summarizes the findings and concludes the thesis, followed by chapter 5 which lists full source code of the developed solution.

1.2 Objectives

This work focuses on automated creation of 2-dimensional game maps using a cellular automata approach. We aim to do so by generating small map tiles, which can be later merged 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. Moreover, we could also allow for editing the tile before placing it in the map. An approach that integrates manual editing or parametrization of desired results with procedural generation techniques has been proposed before [Bid+10], [Sme+10], [Sme+11].

We focus on creation of maps for games, since literature shows map generation as an interesting area for experimentation, although personal motivation influenced the choice as well.

Beginning experimentation with flat maps on 2-dimensional plane avoids the complexity that may arise when dealing with higher dimensions.

We will investigate existing methods for procedural generation of game maps which resemble cave structures. Then, an approach that may be used for automated creation of such maps will be selected and examined with a focus on implementing a working map generator. Main points of focus for this project are as follows:

- research on procedural generation of maps
- selecting a promising approach to use
- designing a map generator program
- implementing the solution in a programming language of choice

TO DO: Objectives - is that all?

1.3 Thesis scope

TO DO: scope - what we will do, what we will not do. specific goals.

TO DO: scope - shortly: what could be done instead

1.4 Technology and tools

The following paragraphs summarize what tools were involved during the project of thesis preparation and performing the experiments.

1.4.1 Hardware

All experiments in this thesis have been performed using a laptop with an Intel x64 2.0 GHz multi-core processor, 16GB RAM and an *nVidia GeForce GTX 560M* graphics card.

1.4.2 Software

Development environment for the purposes of thesis experiments and writing has been set up under Windows 10 operating system with the following software installed:

- Visual Studio 2015 Community IDE
- CMake for Windows
- TeXstudio editor with MikTeX back-end
- Git version control system
- Notepad++
- UMLet open source modelling program
- TO DO: ...

Other configuration details include:

TO DO: environment variables, configuration specifics...

This thesis has been prepared with LaTeX system for document typesetting.

Programming languages

The program 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 VS2015 IDE.

Libraries

The implementation uses following libraries:

- 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/`
- TO DO: ...

1.4.3 Other tools

Design patterns

TO DO: list used design patters, if any. Singleton? Command? Factory?

1.5 Related work (?)

TO DO: think what could be included here

Chapter 2

Research on 2D map generation methods

2.1 Definitions

Before we start planning a solution to the problem of map generation, we must first define what we mean by maps. As stated in chapter 1, our context does not deal with projections of 3D objects onto a plane, like the fields of geography and cartography do [Sny93]. Our goal is simply to generate planar maps.

Map

what is a map?

Generation

what generation means?

TO DO: 2d map types?

2.2 Automation - reduction in development time and cost

TO DO: write about PCG in general, short

TO DO: PCG types of content

TO DO: PCG methods

TO DO: focus on maps

2.3 Existing solutions for 2D map generation

In scientific surveys on PCG methods, we find approaches to map generation employed in the past. As listed by Hendrikx et al. [Hen+13],

TO DO: list map procgen methods

TO DO: HOW it was done until now? options?

TO DO: ref survey with table of 2d dungeon gen

2.3.1 Cellular automata

A cellular automaton 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 electronic computers [Sar00]. One of the most complete sources on cellular automata is a book summarizing research on CA carried out by Stephen Wolfram since 1980s [Wol02], where a classification of cellular automata 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.

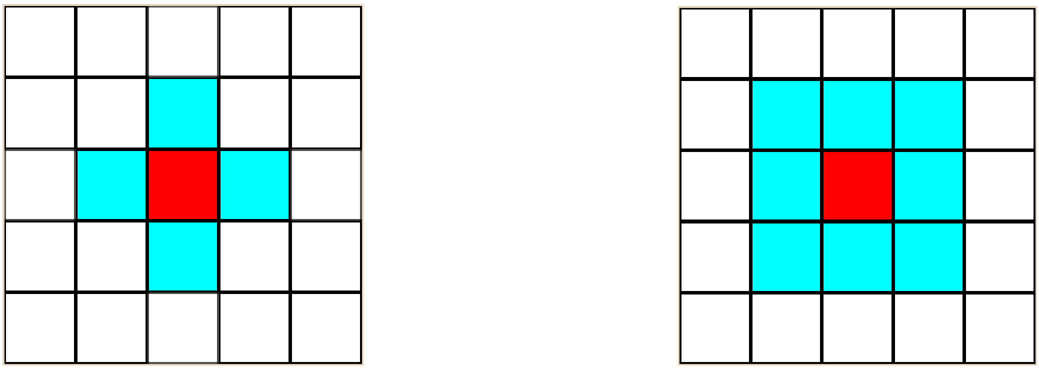
Cell A cell is simply one unit positioned in CA simulation space. Cells have state, which can be simple - for example, a binary digit, an integer - or more complicated - a real number with constraints, a complex number, or other.

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

Moore's neighbourhood Moore neighbourhood includes the cell and its immediate neighbours - one to the north, south, east and west of the cell, as shown in figure 2.1.

Von Neumann neighbourhood Von Neumann neighbourhood includes 8 closest neighbours of the cell - immediate and diagonal, as shown in figure 2.1.

Other types of neighborhood It is possible to imagine other types of cell neighbourhoods, possibly including more cell rings around a cell or only a selection of them arranged in a custom pattern. Those cases are beyond the scope of this thesis.



(a) Moore neighborhood *(b) Von Neumann neighborhood*

Figure 2.1: *Two basic types of cell neighborhood*

There is a...

TO DO:

Every CA simulation also consists of rules which drive the process of cell evolution to its next stage.

TO DO: ca basics - game of life

TO DO: using CA for simulations

TO DO: using CA for generation of content

2.3.2 Generative grammars

TO DO: What is it? Is relevant to maps? Can we use it? Why?

2.3.3 L-systems

TO DO: What is it? Is relevant to maps? Can we use it? Why?

2.3.4 ...

TO DO: What is it? Is relevant to maps? Can we use it? Why?

2.3.5 ...

TO DO: What is it? Is relevant to maps? Can we use it? Why?

2.4 Choosing a method of generation

In order to effectively judge the value that a working map generator may bring to a game development project, we need to consider what characteristics should be evaluated. First, a useful generator must be effective at map generation.

TO DO: how to measure effectiveness? time of map generation, map shape, desirable map features?

Another point to consider is how easy to use such generator can be. Game designers may ultimately decide to use manual methods of map creation if the method of map generation requires too much effort to include in their project.

TO DO: how to measure such ease of use? accessibility?

The third aspect of choice what a generation method could be used is to consider how much value it brings to the designer versus what development costs it can reduce.

TO DO: how to measure cost?

The following subsections describe how each of the mentioned aspects can influence the choice of a generation method.

2.4.1 Effectiveness

TO DO: study on generation time

TO DO: desired characteristics of generated content?

2.4.2 Accessibility

TO DO: study on what makes generation easy to include in game development projects

TO DO: integrating manual editing AND procgen

2.4.3 Cost

TO DO: examples of development costs - human resources, machine resources

TO DO: which of these costs can be reduced by PCG

2.5 Chosen approach: cellular automata for 2D map generation

One of possible proposed approaches is the work of L. Johnson, G. Yannakakis and J. Togelius from IT University of Copenhagen [JYT10].

Authors describe rules of a cellular automaton which are able to transform a tile filled initially with random distribution of cells into a tile which has interesting properties for a map designer.

TO DO: authors describe a process - 1 random image 2 apply CA steps as in article cave gen 3 merge tiles, result: maps!

TO DO: short paragraph on the choice of CA for game maps

TO DO: why we chose CA for mapgen?

TO DO: what are pros and cons of such choice?

Chapter 3

Generating and visualizing maps - proposed solution

TO DO: describe stages of the project

3.1 Analysis of requirements for a map generator

Having gathered the abstract constructs needed to build a CA map generator in chapter 2, we may proceed to state the requirements formally.

3.1.1 Functional requirements

First, we must define the desired functions which a useful map generator should provide to the user.

- user interface allowing playing with parameters
- rendering each generation step
- exporting generated maps
- TO DO:

3.1.2 Non-functional requirements

- allow changing parameters by user

- format of exported maps must be easy to understand and use
- TO DO:

3.1.3 Constraints

- Constraint: time of map tile generation must not exceed 10 seconds.
- TO DO:

3.2 Design

3.2.1 Data structures and persistence

TO DO: how do we store data?

TO DO: diagrams of cell, board

TO DO: exporting data from generator?

TO DO: how designers can get a complete map model?

3.2.2 Application logic

TO DO: how a generator will work

TO DO: behavior diagrams

3.2.3 User interface

TO DO: OpenGL immediate mode paradigm

TO DO: imgui immediate mode user interface library

TO DO: diagram of texture class, used by Component MapGen, uses OpenGL

TO DO: mention Bret Victor talks - why we choose Immediate Mode

3.3 Basic cellular automata simulations

Having chosen cellular automata as a method for generating maps, we need to have a clear idea about how to approach building a program that could simulate a

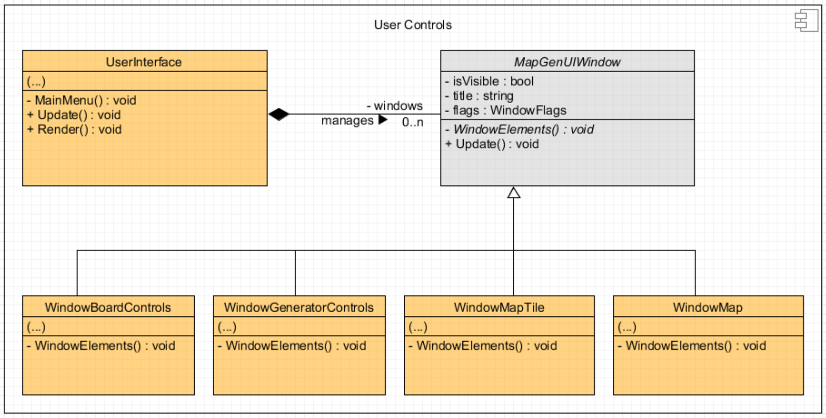


Figure 3.1: *Example model of classes to be used to construct user interface in the map generator program*

cellular automaton. One of the helpful resources on the topic of building cellular automata simulations is chapter 7 in Nature of Code, a book by Daniel Shiffman [Shi12], where we can find a short tutorial to build our first CA simulation. There, author describes elementary concepts needed to construct a basic CA, explains how to implement a working simulation and provides helpful exercises. The tutorial is quite useful as a guide, since examples presented in New Kind of Science [Wol02] are implemented in the Wolfram language and would require familiarity with it. As stated in Nature of Code [Shi12], a 2-dimensional CA would need the following key elements to be simulated:

- 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.

In order to represent the cells of an automaton, a primitive data type is sufficient. However, we could design a class which will act as a collection of cells and provide additional utility to the user. Figure 3.2 presents an example model of a class that would encapsulate a collection of cell states while also preserving information about the board on which those cells are placed.

We can also assign a number to each cell

TO DO: why?

as shown in table 3.1.

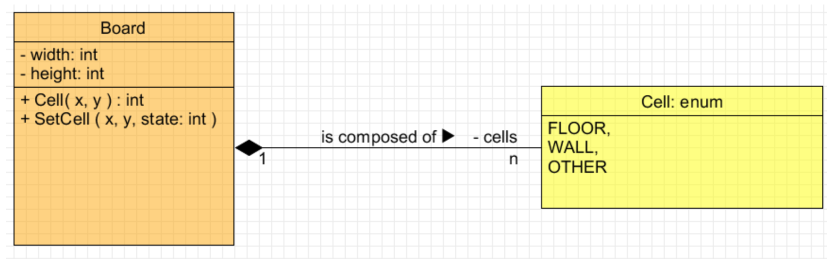


Figure 3.2: *A possible model of a Board Class which holds cell states in its block of memory and lets its user change their states*

0	1	2
7	S	3
6	5	4

Table 3.1: *Cell neighbours, numbered. S denotes selected cell. Cells marked with odd numbers are members of Moore neighborhood of selected cell and all numbered cells are members of Von Neumann neighbourhood of it.*

Such abstraction creates an easy to use interface for further development and is also sufficient to access the values of neighbors to the selected cell. However, in some CA simulations summing the values of cells in neighbourhood is a common operation, so we can include variations of it for convenience. Similarly, a method to translate cell states into texture points would be welcome, since we may possibly need a way to display the state of CA board on screen. Adding those elements to our abstraction yields a class presented on figure 3.3.

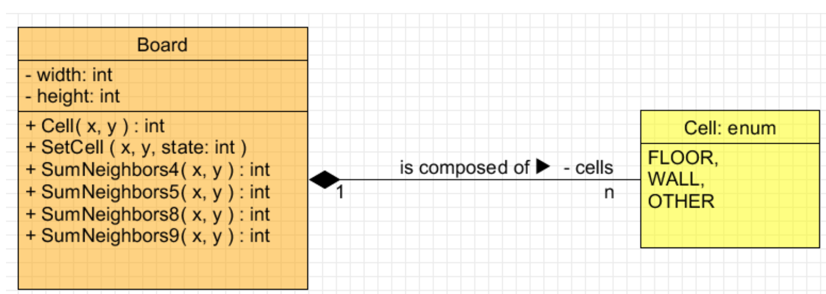


Figure 3.3: *Revised Board abstraction - added methods for neighbor sums and translation of cell states to texels*

TO DO: add neighbor methods to board2

TO DO: result of what it all does?

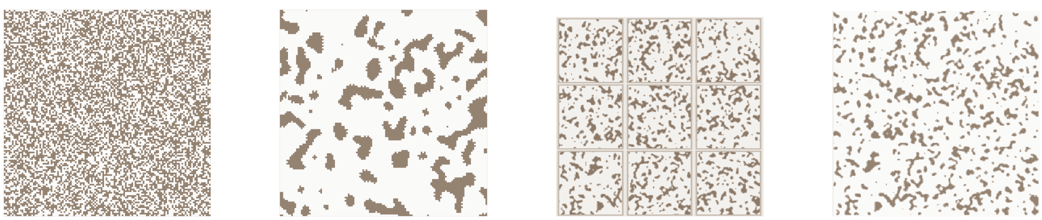
At this point, we could also observe a common property of cellular automata - whenever cell states need to change (the simulation moves to a later step), the state change is applied to every cell in the grid before simulation step ends [Wol84] and cells do not need to be updated in sequence if only all cells will be changed before the next step. Hence, cell state updates could be applied in parallel to reduce the time needed to compute the simulation step. One way to do so would be to apply the findings presented by Reno Fourie in his thesis about applying CUDA technology to reduce time to compute next state of the board in case of 2-dimensional cellular automata [Fou15].

TO DO: what else to include?

TO DO: more on CA, 2dim CA?

3.4 Generating maps with CA

Since the goal of this work is not to just implement a working cellular automaton simulation, we need to find a way to generate maps using CA simulation.



(a) *Step 1 - random noise* ***(b)*** *Step 2 - generated tile* ***(c)*** *Step 3 - tiles in a grid* ***(d)*** *Step 4 - complete map*

Figure 3.4: *Four stages of map construction*

3.5 Implementation

TO DO: describe how it all works now, with object diagrams

TO DO: refer to code itself

3.6 Tests

3.6.1 Performance test

3.7 Deployment (?)

Chapter 4

Conclusions

4.1 Evaluation of results

4.1.1 Effectiveness

4.1.2 Accessibility

4.1.3 Cost

4.2 Perspectives for usage

TO DO: map generator will be used in game project codenamed 'UW'

4.3 Further work

Chapter 5

Full source code

Listing 5.1: *main.cpp Source Code*

```
#include <iostream>

#include <GL\gl3w.h>
#include <GLFW\glfw3.h>

#include "UserInterface_MapGenerator.h"

GLFWwindow* window;

void glfw_error_callback (int error, const char* description)
{
  std::cerr << "GLFW Error " << error << ": " << description << std::endl;
}

bool glfwSetupWindow (unsigned int width, unsigned int height, const char* title)
{
  glfwSetErrorCallback (glfw_error_callback);
  if ( glfwInit () )
  {
    glfwWindowHint (GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint (GLFW_CONTEXT_VERSION_MINOR, 2);
    glfwWindowHint (GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint (GLFW_MAXIMIZED, GLFW_TRUE);
#if __APPLE__
    glfwWindowHint (GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif
    window = glfwCreateWindow (width, height, title, NULL, NULL);
    glfwMakeContextCurrent (window);
    glfwSwapInterval (1); // Enable vsync
    gl3wInit ();
```

```
  return true;
 }
 return false;
}

int main (int, char**)
{
 if ( !glfwSetupWindow (800, 600, "Cellular Automata Map Generator 152017") ) return 1;
 else
 {
  UserInterface_MapGenerator missionControls = UserInterface_MapGenerator (window);
  while ( !glfwWindowShouldClose (window) ) // Main loop
  {
   glfwPollEvents ();
   {
    missionControls.Update ();
    missionControls.Render ();
   }
   glfwSwapBuffers (window);
  }
 }
 glfwDestroyWindow (window);
 glfwTerminate ();
 return 0;
}
```

Listing 5.2: *Board.h Source Code*

```
#pragma once

#include <vector>
#include "TextureAtlas.h"

enum CELL_t
{
 // TODO: we may need to make a class Cell{} perhaps, but for now, just enum
 CELL_FLOOR = 0,
 CELL_WALL = 1,
 CELL_OTHER = 2
};

class Board
{
private:
 std::vector<CELL_t> cells;
 bool isBoardChanged = false;
 bool isCellMarked( unsigned x, unsigned y )
```

```
  {
    if ( !isBoardChanged ) return false;
    int s = Neighbors4_Sum( x, y );
    return ( s == 2 || s == 3 ) && isMarkingEnabled;
  }

public:
  static float ui_boardDisplayScale;
  static bool isMarkingEnabled;
  unsigned int cellsX;
  unsigned int cellsY;

  Board( unsigned int width, unsigned int height )
  {
    cellsX = width;
    cellsY = height;
    Clear( CELL_FLOOR );
  }
  ~Board() = default;

  // BOARD STATE
  CELL_t CellAt( unsigned int x, unsigned int y )
  {
    return cells.at( cellsX * ( y%cellsY ) + ( x%cellsX ) );
  }
  unsigned int Neighbors4_Sum( unsigned int x, unsigned int y )
  {
    unsigned int sum = 0;
    //sum += CellAt( x - 1, y - 1 );
    sum += CellAt( x - 1, y + 0 );
    //sum += CellAt( x - 1, y + 1 );
    sum += CellAt( x + 0, y - 1 );
    //sum += CellAt( x + 0, y + 0 );
    sum += CellAt( x + 0, y + 1 );
    //sum += CellAt( x + 1, y - 1 );
    sum += CellAt( x + 1, y + 0 );
    //sum += CellAt( x + 1, y + 1 );
    return sum;
  }
  unsigned int Neighbors5_Sum( unsigned int x, unsigned int y )
  {
    unsigned int sum = 0;
    //sum += CellAt( x - 1, y - 1 );
    sum += CellAt( x - 1, y + 0 );
    //sum += CellAt( x - 1, y + 1 );
    sum += CellAt( x + 0, y - 1 );
    sum += CellAt( x + 0, y + 0 );
```

```
  sum += CellAt( x + 0, y + 1 );
  //sum += CellAt( x + 1, y - 1 );
  sum += CellAt( x + 1, y + 0 );
  //sum += CellAt( x + 1, y + 1 );
  return sum;
}
unsigned int Neighbors8_Sum( unsigned int x, unsigned int y )
{
  unsigned int sum = 0;
  sum += CellAt( x - 1, y - 1 );
  sum += CellAt( x - 1, y + 0 );
  sum += CellAt( x - 1, y + 1 );
  sum += CellAt( x + 0, y - 1 );
  //sum += CellAt( x + 0, y + 0 );
  sum += CellAt( x + 0, y + 1 );
  sum += CellAt( x + 1, y - 1 );
  sum += CellAt( x + 1, y + 0 );
  sum += CellAt( x + 1, y + 1 );
  return sum;
}
unsigned int Neighbors9_Sum( unsigned int x, unsigned int y )
{
  unsigned int sum = 0;
  sum += CellAt( x - 1, y - 1 );
  sum += CellAt( x - 1, y + 0 );
  sum += CellAt( x - 1, y + 1 );
  sum += CellAt( x + 0, y - 1 );
  sum += CellAt( x + 0, y + 0 );
  sum += CellAt( x + 0, y + 1 );
  sum += CellAt( x + 1, y - 1 );
  sum += CellAt( x + 1, y + 0 );
  sum += CellAt( x + 1, y + 1 );
  return sum;
}

// BOARD MODIFY
void SetCellAt( unsigned int x, unsigned int y, CELL_t newState )
{
  cells.at( cellsX * ( y%cellsY ) + ( x%cellsX ) ) = newState;
  isBoardChanged = true;
  return;
}
void ReplaceWith( const Board* boardToCopy )
{
  cells.erase( cells.begin(), cells.end() );
  cells.assign( boardToCopy->cells.begin(), boardToCopy->cells.end() );
  isBoardChanged = true;
```

```
}
void Clear( CELL_t with )
{
  cells.assign( ( cellsX*cellsY ), with );
  isBoardChanged = true;
}

// BOARD DRAWING
float DisplayScaleX()
{
  return ui_boardDisplayScale * cellsX;
}
float DisplayScaleY()
{
  return ui_boardDisplayScale * cellsY;
}
void  DrawCellsToTexture( unsigned texIdx, bool forceDraw = false )
{
  if ( isBoardChanged || forceDraw )
  {
    for ( unsigned int x = 0; x < cellsX; x++ )
    {
      for ( unsigned int y = 0; y < cellsY; y++ )
      {
        switch ( CellAt( x, y ) )
        {
        case CELL_FLOOR: SimpleTexture2D::Texture( texIdx )->SetTexelColor( x, y, color_BLACK ); break;
        case CELL_WALL:  SimpleTexture2D::Texture( texIdx )->SetTexelColor( x, y, color_GREEN ); break;
        case CELL_OTHER: SimpleTexture2D::Texture( texIdx )->SetTexelColor( x, y, color_WHITE ); break;
        default:         SimpleTexture2D::Texture( texIdx )->SetTexelColor( x, y, color_BLUE ); break;
        }
        if ( isCellMarked( x, y ) ) SimpleTexture2D::Texture( texIdx )->SetTexelColor( x, y, color_RED );
      }
    }
  }
  isBoardChanged = false;
}
};

float Board::ui_boardDisplayScale = 4.0f;
bool Board::isMarkingEnabled = true;
```

Listing 5.3: Map.h Source Code

```
#pragma once
#include <vector>
#include "Board.h"


class Map
{
private:
  Board* mapBoard;
  std::vector<Board> mapTiles;
  unsigned mapIdx( unsigned x, unsigned y )
  {
    if ( x < mapSide && y < mapSide )
    {
      return mapSide * x + y;
    }
    else throw;
  }
public:
  static float ui_mapDisplayScale;
  unsigned mapSide;
  unsigned mapArea;
  Map( unsigned boardsizeX, unsigned boardsizeY, unsigned mapN = 2 )
  {
    mapSide = ( 2 * mapN + 1 );
    mapArea = mapSide*mapSide;
    for ( unsigned i = 1; i <= mapArea; i++ ) SimpleTexture2D::Texture( i )->Resize(
        boardsizeX, boardsizeY );
    mapTiles.assign( mapArea, Board( boardsizeX, boardsizeY ) );
    mapBoard = new Board( mapSide*boardsizeX, mapSide*boardsizeY );
    SimpleTexture2D::Texture( mapArea + 1 )->Resize( mapSide*boardsizeX, mapSide*
        boardsizeY );
  }
  ~Map()
  {
    if (mapBoard) delete mapBoard;
    mapTiles.clear();
  }

  //// MAP DRAWING
  float DisplayScaleX_tiles()
  {
    return ( ui_mapDisplayScale / mapSide ) * mapTiles.at( 0 ).cellsX;
  }
  float DisplayScaleY_tiles()
```

```
{
  return ( ui_mapDisplayScale / mapSide ) * mapTiles.at( 0 ).cellsY;
}
float DisplayScaleX_map()
{
  return ( ui_mapDisplayScale / mapSide ) * mapBoard->cellsX;
}
float DisplayScaleY_map()
{
  return ( ui_mapDisplayScale / mapSide ) * mapBoard->cellsY;
}
void* DrawTileAt( unsigned x, unsigned y )
{
  mapTiles.at( mapIdx( x, y ) ).DrawCellsToTexture( mapIdx( x, y ) + 1 );
  return SimpleTexture2D::Texture( mapIdx( x, y ) + 1 )->Render();
}
void* DrawMap()
{
  mapBoard->DrawCellsToTexture( mapArea + 1 );
  return SimpleTexture2D::Texture( mapArea + 1 )->Render();
}

//// MAP BUILDING
void TileReplace( unsigned x, unsigned y, Board* tile )
{
  mapTiles.at( mapIdx( x, y ) ).ReplaceWith( tile );
}
void TileJoinAll()
{
  unsigned ct_x = mapTiles.at( 0 ).cellsX;
  unsigned ct_y = mapTiles.at( 0 ).cellsY;

  for ( unsigned x = 0; x < mapSide*ct_x; x++ )
  {
    for ( unsigned y = 0; y < mapSide*ct_y; y++ )
    {
      mapBoard->SetCellAt( x, y, mapTiles.at( mapIdx( y / ct_y, x / ct_x ) ).CellAt( x % ct_x,
      y % ct_y ) );
    }
  }
}
void TileClearAll()
{
  mapBoard->Clear( CELL_FLOOR );
}
void MapMergeTiles()
{
```

```
    Board* tempBoard = new Board( mapBoard->cellsX, mapBoard->cellsY );
    Rules::EvolveState( mapBoard, tempBoard );
    {
      // TODO: run algorithm to merge tile edges
    }
    delete mapBoard;
    mapBoard = tempBoard;
  }
};

float Map::ui_mapDisplayScale = 6.0f;
```

Listing 5.4: *Ruleset.h Source Code*

```
#pragma once

#include "Board.h"

enum Ruleset
{
  RULES_GAMEOFLIFE = 0,
  RULES_MAPGEN = 1
};
class Rules
{
public:
  static void EvolveState( Board * before, Board * after, Ruleset r = RULES_MAPGEN )
  {
    switch ( r )
    {
    case RULES_GAMEOFLIFE: Rules_GameOfLife( before, after ); break;
    case RULES_MAPGEN:     Rules_MapGen( before, after ); break;
    default: break;
    }
  }
private:
  static void Rules_GameOfLife( Board *before, Board *after )
  {
    Board::isMarkingEnabled = false;
    for ( unsigned int x = 0; x < before->cellsX; x++ )
    {
      for ( unsigned int y = 0; y < before->cellsY; y++ )
      {
        switch ( before->Neighbors8_Sum( x, y ) )
        {
        case 2:  after->SetCellAt( x, y, before->CellAt( x, y ) ); break;
        case 3:  after->SetCellAt( x, y, CELL_WALL );             break;
```

```
        default: after->SetCellAt( x, y, CELL_FLOOR );          break;
        }
      }
    }
    return;
  }
  static void Rules_MapGen( 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->Neighbors8_Sum( x, y );
        if ( sum < 5 ) after->SetCellAt( x, y, CELL_WALL );
        if ( sum > 5 ) after->SetCellAt( x, y, CELL_FLOOR );
      }
    }
    return;
  }
  // TODO: with ruleset separated from automaton, maybe we could try rules where cellstate is dependent on states in the past, not just the previous one
};
```

Listing 5.5: *TextureAtlas.h Source Code*

```
#pragma once
#include <vector>
#include <GLFW\glfw3.h>

typedef GLuint Color_RGBA;

const Color_RGBA color_BLACK = 0x000000CC;
const Color_RGBA color_WHITE = 0xFFFFFFCC;
const Color_RGBA color_RED = 0xFF0000CC;
const Color_RGBA color_GREEN = 0x00FF00CC;
const Color_RGBA color_BLUE = 0x0000FFCC;

class SimpleTexture2D
{
private:
  static std::vector<SimpleTexture2D*> textures;

  GLuint texID_GL;
  unsigned int texSizeX;
  unsigned int texSizeY;
  std::vector<Color_RGBA> texelsRGBA;
```

```
  SimpleTexture2D( unsigned width, unsigned height )
  {
   texSizeX = width;
   texSizeY = height;
   Clear( color_BLUE );
   glGenTextures( 1, &texID_GL );
   glBindTexture( GL_TEXTURE_2D, texID_GL );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
     GL_CLAMP_TO_BORDER );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
     GL_CLAMP_TO_BORDER );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
   glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, texSizeX, texSizeY, 0, GL_RGBA,
     GL_UNSIGNED_INT_8_8_8_8, texelsRGBA.data() );
   glGenerateMipmap( GL_TEXTURE_2D );
  }

public:
  static SimpleTexture2D* Texture( unsigned i )
  {
   if ( !( i < textures.size() ) )
    textures.emplace( textures.begin() + i, new SimpleTexture2D( 2, 2 ) );
   return textures.at( i );
  }
  ~SimpleTexture2D()
  {
   glDeleteTextures( 1, &texID_GL );
  }
  void Resize( unsigned width, unsigned height)
  {
   texSizeX = width;
   texSizeY = height;
   Clear( color_BLUE );
   glDeleteTextures( 1, &texID_GL );
   glGenTextures( 1, &texID_GL );
   glBindTexture( GL_TEXTURE_2D, texID_GL );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
     GL_CLAMP_TO_BORDER );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
     GL_CLAMP_TO_BORDER );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
   glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
   glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, texSizeX, texSizeY, 0, GL_RGBA,
     GL_UNSIGNED_INT_8_8_8_8, texelsRGBA.data() );
   glGenerateMipmap( GL_TEXTURE_2D );
```

```
    return;
  }
  void* Render()
  {
    glBindTexture( GL_TEXTURE_2D, texID_GL );
    glTexSubImage2D( GL_TEXTURE_2D, 0, 0, 0, texSizeX, texSizeY, GL_RGBA,
      GL_UNSIGNED_INT_8_8_8_8, texelsRGBA.data() );
    return reinterpret_cast<void*>( texID_GL );
  }
  void SetTexelColor( unsigned int x, unsigned int y, Color_RGBA color )
  {
    if ( ( x < texSizeX ) && ( y < texSizeY ) ) texelsRGBA.at( texSizeX * y + x ) = color;
    return;
  }
  void Clear( Color_RGBA clearColor )
  {
    texelsRGBA.assign( texSizeX * texSizeY, clearColor );
  }

};

std::vector<SimpleTexture2D*> SimpleTexture2D::textures = std::vector<SimpleTexture2D
    *>();
```

Listing 5.6: *TileGenerator.h Source Code*

```
#pragma once
#include <vector>
#include <random>
#include <chrono> // systime as random seed

#include "Ruleset.h"
#include "Board.h"
#include "TextureAtlas.h"
#include "Map.h"

enum BoardInit_t
{
  CLEAR_RANDOM,
  CLEAR_CHESS,
  CLEAR_XYMOD,
  TEST_GLIDER
};

class TileGenerator
{
public:
```

```
static int ui_boardSize[2];
static int ui_stepCount;
static int ui_stepSelected;
static float ui_stepProgress;
static Ruleset currentRules;

static TileGenerator* State()
{
  if ( !single_instance )
    single_instance = new TileGenerator();
  return single_instance;
}
static TileGenerator* Reset()
{
  delete single_instance;
  return single_instance = new TileGenerator();
}
~TileGenerator()
{
  delete map;
}

// STEP SELECTORS
void StepSelect( unsigned step )
{
  ui_stepSelected = step % StepCount();
}
void StepJump( unsigned int offset )
{
  StepSelect( ui_stepSelected + offset );
}
void StepJumpLast()
{
  StepSelect( StepCount() - 1 );
}

// STEP USAGE
unsigned int StepCount()
{
  return generations.size();
}
Board* SelectedStep()
{
  return ui_stepSelected < generations.size() ?
    &generations.at( ui_stepSelected ) :
    &generations.at( 0 );
}
```

```
void* SelectedStepImage()
{
  if (ui_stepSelected < generations.size() )
  {
    generations.at( ui_stepSelected ).DrawCellsToTexture( 0, true );
  }
  return SimpleTexture2D::Texture( 0 )->Render();
}

// STEP GENERATION
void  RegenerateStepsFrom( BoardInit_t initialBoard )
{
  switch ( initialBoard )
  {
  case CLEAR_RANDOM:  InitGenAllRandom();  break;
  case CLEAR_CHESS:   InitGenClearChess(); break;
  case CLEAR_XYMOD:   InitGenClearXYMOD(); break;
  case TEST_GLIDER:   InitGenTestGlider(); break;
  }
  GenerateSteps();
}
void ChangeRuleset( Ruleset r )
{
  currentRules = r;
  GenerateSteps();
}

// MAP USAGE
Map* ConstructedMap()
{
  return map;
}

private:
 static TileGenerator *single_instance;
 std::vector<Board> generations;
 Map* map;

 TileGenerator()
 {
  SimpleTexture2D::Texture( 0 )->Resize( ui_boardSize[0], ui_boardSize[1] );
  generations.assign( ui_stepCount, Board( ui_boardSize[0], ui_boardSize[1] ) );
  currentRules = RULES_MAPGEN;
  map = new Map( ui_boardSize[0], ui_boardSize[1] );
 }

 unsigned int CellCountX()
```

```
{
  return generations.at( 0 ).cellsX;
}
unsigned int CellCountY()
{
  return generations.at( 0 ).cellsY;
}

void BoardClear()
{
  BoardClear( CELL_FLOOR );
}
void BoardClear( CELL_t state )
{
  generations.clear();
  generations.assign( ui_stepCount, Board( ui_boardSize[0], ui_boardSize[1] ) );
}

void InitGenAllRandom()
{
  BoardClear();
  unsigned seed = unsigned( std::chrono::system_clock::now().time_since_epoch().count() )
     ;
  std::mt19937 randomizer( seed );
  std::uniform_int_distribution<int> distribution( 1, 100 );
  distribution.reset();
  CELL_t c = CELL_OTHER;
  for ( unsigned int x = 0; x < CellCountX(); x++ )
  {
    for ( unsigned int y = 0; y < CellCountY(); y++ )
    {
      switch ( distribution( randomizer ) % 2 )
      {
      case 0: c = CELL_WALL; break;
      case 1: c = CELL_FLOOR; break;
        //case 2: c = CELL_OTHER; break;
      default:break;
      }
      generations.at( 0 ).SetCellAt( x, y, c );
    }
  }
  return;
}
void InitGenClearChess()
{
  BoardClear();
  for ( unsigned int x = 0; x < CellCountX(); x++ )
```

```
  {
   for ( unsigned int y = 0; y < CellCountY(); y++ )
   {
    if ( ( x % 2 ) ^ ( y % 2 ) )
     generations.at( 0 ).SetCellAt( x, y, CELL_WALL );
    else
     generations.at( 0 ).SetCellAt( x, y, CELL_FLOOR );
   }
  }
}
void InitGenClearXYMOD()
{
  BoardClear();
  for ( unsigned int x = 0; x < CellCountX(); x++ )
  {
   for ( unsigned int y = 0; y < CellCountY(); y++ )
   {
    if ( ( ( x * 17 ) % ( 1 + y * 8 ) ) % 3 )
     generations.at( 0 ).SetCellAt( x, y, CELL_WALL );
    else
     generations.at( 0 ).SetCellAt( x, y, CELL_FLOOR );
   }
  }
}
void InitGenTestGlider()
{
  BoardClear();
  {
   unsigned int a = generations.at( 0 ).cellsX / 2;
   unsigned int b = generations.at( 0 ).cellsY / 2;
   generations.at( 0 ).SetCellAt( a - 1, b + 0, CELL_WALL );
   generations.at( 0 ).SetCellAt( a + 0, b + 1, CELL_WALL );
   generations.at( 0 ).SetCellAt( a + 1, b - 1, CELL_WALL );
   generations.at( 0 ).SetCellAt( a + 1, b + 0, CELL_WALL );
   generations.at( 0 ).SetCellAt( a + 1, b + 1, CELL_WALL );
  }
}

void GenerateSteps()
{
  // TODO: we could try to implement this function in lazy evaluation manner.
  // ui_stepProgress = 0.f;
  // ui_stepProgress = step / generations.size();
  for ( unsigned int step = 1; step < generations.size(); step++ )
  {
   Rules::EvolveState( &generations.at( step - 1 ), &generations.at( step ), currentRules );
  }
```

```
  }

};

int TileGenerator::ui_boardSize[2] = { 128, 128 };
int TileGenerator::ui_stepCount = 10;
int TileGenerator::ui_stepSelected = 0;
float TileGenerator::ui_stepProgress = 0.f;
Ruleset TileGenerator::currentRules = RULES_MAPGEN;

TileGenerator* TileGenerator::single_instance = 0;
```

Listing 5.7: *UserInterface_MapGenerator.h Source Code*

```
#pragma once
#include <string>
#include <list>
#include <GLFW\glfw3.h>

#include "TileGenerator.h"
#include "Window_Base.h"

class UserInterface_MapGenerator
{
public:
  UserInterface_MapGenerator( GLFWwindow* window )
  {
    glfwFocusWindow( system_window = window );
    // Setup ImGui binding
    ImGui::CreateContext();
    ImGui_ImplGlfwGL3_Init( system_window, true );
    ImGui::StyleColorsDark();
    // Setup ui elements
    UserInterfaceWindows.push_back( new WindowBoardControls( 10.f, 10.f, &clear_color ) );
    UserInterfaceWindows.push_back( new WindowGeneratorControls( 20.f, 200.f ) );
    UserInterfaceWindows.push_back( new WindowBoardImage( 300.f, 150.f ) );
    UserInterfaceWindows.push_back( new WindowMapTileGrid( 500.f, 200.f ) );
    // Initialize automaton with default data
    TileGenerator::State()->RegenerateStepsFrom( CLEAR_RANDOM );
  }
  ~UserInterface_MapGenerator()
  {
    // Cleanup
    UserInterfaceWindows.clear();
    ImGui_ImplGlfwGL3_Shutdown();
    ImGui::DestroyContext();
  }
```

```
  void Update()
  {
   ImGui_ImplGlfwGL3_NewFrame();
   {
    // UI updates:
    MainMenu();
    for ( Window_Base *w : UserInterfaceWindows ) w->Update();
    if ( isImguiDemoVisible )
    {
     ImGui::SetNextWindowPos( ImVec2( 10, 150 ), ImGuiCond_FirstUseEver );
     ImGui::ShowDemoWindow( &isImguiDemoVisible ); // Imgui demo for reference to
     ImGui examples
    }
    if ( isImguiMetricsVisible )
    {
     ImGui::ShowMetricsWindow( &isImguiMetricsVisible );
    }
   }
   if ( isProgramTerminated ) glfwSetWindowShouldClose( system_window, GLFW_TRUE );
  }
  void Render()
  {
   // Rendering
   int display_w, display_h;
   glfwGetFramebufferSize( system_window, &display_w, &display_h );
   glViewport( 0, 0, display_w, display_h );
   glClearColor( clear_color.x, clear_color.y, clear_color.z, clear_color.w );
   glClear( GL_COLOR_BUFFER_BIT );
   // Render gui
   ImGui::Render();
   ImGui_ImplGlfwGL3_RenderDrawData( ImGui::GetDrawData() );
  }

private:
  bool isProgramTerminated = false;
  bool isImguiDemoVisible = false;
  bool isImguiMetricsVisible = false;
  GLFWwindow* system_window;
  ImVec4 clear_color = ImVec4( 0.45f, 0.55f, 0.60f, 1.00f );

  std::list<Window_Base*> UserInterfaceWindows;

  void MainMenu()
  {
   if ( ImGui::BeginMainMenuBar() )
   {
    if ( ImGui::BeginMenu( "System:" ) )
```

```
      {
        ImGui::MenuItem( "New map...", "CTRL+N", false, false );// Disabled item
        ImGui::MenuItem( "Quit", "ALT+F4", &isProgramTerminated );
        ImGui::EndMenu();
      }
      if ( ImGui::BeginMenu( "Editing:" ) )
      {
        ImGui::MenuItem( "Undo", "CTRL+Z", false, false );  // Disabled item
        ImGui::MenuItem( "Redo", "CTRL+Y", false, false );  // Disabled item
        ImGui::EndMenu();
      }
      if ( ImGui::BeginMenu( "View:" ) )
      {
        for ( Window_Base *w : UserInterfaceWindows )
        {
          ImGui::MenuItem( w->menutitle, NULL, &w->isVisible, &w->isVisible );
        }
        ImGui::Separator();
        ImGui::MenuItem( "ImGui Demo Window", NULL, &isImguiDemoVisible, &
       isImguiDemoVisible );
        ImGui::MenuItem( "ImGui Metrics Window", NULL, &isImguiMetricsVisible, &
       isImguiMetricsVisible );
        ImGui::EndMenu();
      }
      if ( ImGui::BeginMenu( "About:" ) )
      {
        ImGui::MenuItem( "Author", NULL, false, false ); // Disabled item
        ImGui::MenuItem( "Used libraries", NULL, false, false );// Disabled item
        ImGui::EndMenu();
      }
      ImGui::EndMainMenuBar();
    }
  }
};
```

Listing 5.8: *Window_Base.h Source Code*

```
#pragma once

#include "imgui\imgui.h"
#include "imgui\imgui_impl_glfw_gl3.h"

#include "TileGenerator.h"
#include "SimpleTexture.h"

class Window_Base
{
```

```
public:
  bool isVisible = true;
  char* menutitle = "<...>";

  Window_Base() {}
  ~Window_Base() = default;
  void Update()
  {
    if ( isVisible )
    {
      ImGui::SetNextWindowPos( ImVec2( x, y ), ImGuiCond_FirstUseEver );
      if ( ImGui::Begin( title, &isVisible, flags ) )
      {
        WindowElements();
        ImGui::End();
      }
    }
  }

protected:
  float x, y;
  char* title = "<...>";
  ImGuiWindowFlags flags;
  virtual void WindowElements() {}
};

#include "WindowBoardControls.h"
#include "WindowBoardImage.h"
#include "WindowGeneratorControls.h"
#include "WindowMapTileGrid.h"
```

Listing 5.9: *WindowBoardControls.h Source Code*

```
#pragma once
#include "Window_Base.h"
class WindowBoardControls : public Window_Base
{
public:

  WindowBoardControls( float initialPositionX, float initialPositionY, ImVec4 *bgColor )
  {
    x = initialPositionX;
    y = initialPositionY;
    title = "Board Controls";
    menutitle = "Show Window: Board Controls";
    flags = ImGuiWindowFlags_NoCollapse;
    ccPtr = bgColor;
```

```
}
const char *build_str = "Build date: " __DATE__ " " __TIME__;
ImVec4 *ccPtr;
void WindowElements()
{
  {
    ImGui::Text( build_str );
    ImGui::Text( "%.3f ms/frame (%.1f FPS)", 1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate );
  }
  ImGui::Separator();
  {
    ImGui::Text( "Display options: " );
    ImGui::ColorEdit3( "Background clear color", reinterpret_cast<float*>( ccPtr ) );
    ImGui::SliderFloat( "Board zoom/scale", &Board::ui_boardDisplayScale, 2.f, 20.f );
    ImGui::SliderFloat( "Map zoom/scale", &Map::ui_mapDisplayScale, 2.f, 20.f );
  }
  ImGui::Separator();
  {
    ImGui::Text( "Board parameters:" );
    ImGui::SliderInt2( "width, height", TileGenerator::ui_boardSize, 16, 256 );
    ImGui::SliderInt( "simulation step count", &TileGenerator::ui_stepCount, 10, 200 );
    if ( ImGui::Button( "RECONSTUCT BOARD" ) ) { TileGenerator::Reset(); }
  }
  ImGui::Separator();
  {
    ImGui::Text( "Board initializers:" );
    if ( ImGui::Button( "init : random     " ) ) { TileGenerator::State()->RegenerateStepsFrom( CLEAR_RANDOM ); }
    if ( ImGui::Button( "init : chessboard" ) ) { TileGenerator::State()->RegenerateStepsFrom( CLEAR_CHESS ); }
    if ( ImGui::Button( "init : modxyboard" ) ) { TileGenerator::State()->RegenerateStepsFrom( CLEAR_XYMOD ); }
    if ( ImGui::Button( "init : glidertest" ) ) { TileGenerator::State()->RegenerateStepsFrom( TEST_GLIDER ); }

    ImGui::TextWrapped( "Note: these functions generate all board states at once. Calling them may take some time to finish, depending on board size and step count." );
  }
  ImGui::Separator();
  {
    ImGui::Text( "Cell Types:" );
    // TODO: stats of cell types in board. needs better cell implementation
    // for each celltype
    std::string cellStats; // + type name + count cells, etc

    ImGui::Text( "CELLTYPE1 : %% on board " );
```

```
      ImGui::Text( "CELLTYPE2 : %% on board " );
      ImGui::Text( "CELLTYPE3 : %% on board " );
    }
    ImGui::Separator();
    {
      ImGui::Text( "Other options: " );
      // TODO : (future work) enable/disable pixel editing with mouse
      ImGui::Text( "..." );
    }
    ImGui::Separator();
  }
};
```

Listing 5.10: WindowBoardImage.h Source Code

```
#pragma once
#include "Window_Base.h"
class WindowBoardImage : public Window_Base
{
public:
  WindowBoardImage( float initialPositionX, float initialPositionY )
  {
    x = initialPositionX;
    y = initialPositionY;
    title = "Generated Map Tile";
    menutitle = "Show Window: Generated Map Tile";
    flags = ImGuiWindowFlags_NoCollapse | ImGuiWindowFlags_AlwaysAutoResize;
  }
  void WindowElements()
  {
    // ImGui::ProgressBar( TileGenerator::ui_stepProgress, ImVec2( 0.0f, 0.0f ) );
    ImGui::Separator();
    ImGui::Image(
      TileGenerator::State()->SelectedStepImage(),
      ImVec2(
        TileGenerator::State()->SelectedStep()->DisplayScaleX(),
        TileGenerator::State()->SelectedStep()->DisplayScaleY() )
    );
    ImGui::Separator();
    {
      if ( ImGui::SliderInt( "Step Selector", &TileGenerator::ui_stepSelected, 0, TileGenerator::
        State()->StepCount()-1 ) ) {}
    }
    ImGui::Separator();
    {
      ImGui::Text( "Precise selectors:" );
      if ( ImGui::Button( "   0    " ) ) { TileGenerator::State()->StepSelect( 0 ); } ImGui::
```

```
        SameLine();
      if ( ImGui::Button( "<<< 3 STEP" ) ) { TileGenerator::State()->StepJump( -3 ); }  ImGui::
        SameLine();
      if ( ImGui::Button( "<<< 1 STEP" ) ) { TileGenerator::State()->StepJump( -1 ); }  ImGui::
        SameLine();
      if ( ImGui::Button( "1 STEP >>>" ) ) { TileGenerator::State()->StepJump( 1 ); }  ImGui::
        SameLine();
      if ( ImGui::Button( "3 STEP >>>" ) ) { TileGenerator::State()->StepJump( 3 ); }  ImGui::
        SameLine();
      if ( ImGui::Button( "   END    " ) ) { TileGenerator::State()->StepJumpLast(); }
    }
  }
};
```

Listing 5.11: *WindowGeneratorControls.h Source Code*

```
#pragma once
#include "Window_Base.h"
class WindowGeneratorControls : public Window_Base
{
public:
  WindowGeneratorControls( float initialPositionX, float initialPositionY )
  {
    x = initialPositionX;
    y = initialPositionY;
    title = "Generator Controls";
    menutitle = "Show Window: Generator Controls";
    flags = ImGuiWindowFlags_NoCollapse;
  }
  void WindowElements()
  {
    {
      static int ruleChoice = int(TileGenerator::currentRules);
      ImGui::Text( "Rulesets:" );
      if ( ImGui::RadioButton( "Game of Life Rules (for tests)", &ruleChoice, 0 ) )
      {
        TileGenerator::State()->ChangeRuleset( RULES_GAMEOFLIFE );
      }
      if ( ImGui::RadioButton( "Map Generator Rules", &ruleChoice, 1 ) )
      {
        TileGenerator::State()->ChangeRuleset( RULES_MAPGEN );
      }
    }
    ImGui::Separator();
    {
      ImGui::Text( "Rules:" );
      ImGui::Text( "R1 : neighbors : condition : new cell" );
```

```
      ImGui::Text( "R2 : neighbors : condition : new cell" );
      ImGui::Text( "R3 : neighbors : condition : new cell" );
    }
    ImGui::Separator();
  }
};
```

Listing 5.12: *WindowMapTileGrid.h Source Code*

```
#pragma once

#include "Window_Base.h"
#include "SimpleTexture.h"
#include "Map.h"

class WindowMapTileGrid : public Window_Base
{
public:
  WindowMapTileGrid( float initialPositionX, float initialPositionY )
  {
    x = initialPositionX;
    y = initialPositionY;
    title = "Map";
    menutitle = "Show Window: Map";
    flags = ImGuiWindowFlags_NoCollapse | ImGuiWindowFlags_AlwaysAutoResize;
  }
  ~WindowMapTileGrid()
  {
  }

  void WindowElements()
  {
    static bool mapMode = false;
    if ( !mapMode )
    {
      ShowMapTiles( TileGenerator::State()->ConstructedMap() );
      ImGui::Separator();
      ImGui::TextWrapped( "Click on a map tile to replace it with current generated tile." );
      ImGui::Separator();
      if ( ImGui::Button( "Join tiles into map" ) )
      {
        mapMode = true;
        TileGenerator::State()->ConstructedMap()->TileJoinAll();
      }
      if ( ImGui::Button( "Clear map tiles" ) )
      {
        TileGenerator::State()->ConstructedMap()->TileClearAll();
```

```
      }
    }
    if ( mapMode )
    {
      ShowMap( TileGenerator::State()->ConstructedMap() );
      ImGui::Separator();
      if ( ImGui::Button( "Run one CA step on Map" ) )
      {
        TileGenerator::State()->ConstructedMap()->MapMergeTiles();
      }
      ImGui::SameLine();
      if ( ImGui::Button( "Go back to editing" ) )
      {
        mapMode = false;
      }
      if ( ImGui::Button( "Export map to file <?>" ) )
      {
        // TODO: export Map to image on disk
      }

    }
  }
private:
  void ShowMapTiles( Map* m )
  {
    for ( int x = 0; x < m->mapSide; x++ )
    {
      for ( int y = 0; y < m->mapSide; y++ )
      {
        if ( ImGui::ImageButton(
          m->DrawTileAt( x, y ),
          ImVec2( m->DisplayScaleX_tiles(), m->DisplayScaleY_tiles() )
        ) )
        {
          m->TileReplace( x, y, TileGenerator::State()->SelectedStep() );
        }
        if ( y != m->mapSide - 1 )ImGui::SameLine();
      }
    }
  }
  void ShowMap( Map* m )
  {
    ImGui::Image(
      m->DrawMap(),
      ImVec2( m->DisplayScaleX_map(), m->DisplayScaleY_map() )
    );
  }
```

```
};
```

Bibliography

[Bid+10] 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.

[DC+11] 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.

[Fou15] Ryno Fourie. "A parallel cellular automaton simulation framework using CUDA". PhD thesis. Stellenbosch: Stellenbosch University, 2015.

[Hen+13] 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.

[Ios09] Alexandru Iosup. "Poggi: Puzzle-based online games on grid infrastructures". In: *European Conference on Parallel Processing*. Springer. 2009, pp. 390–403.

[JYT10] 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.

[KM07] 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.

[LN03] Sylvain Lefebvre and Fabrice Neyret. "Pattern based procedural textures". In: *Proceedings of the 2003 symposium on Interactive 3D graphics*. ACM. 2003, pp. 203–212.

[Sar00] 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.

[Shi12] Daniel Shiffman. *The Nature of Code: Simulating Natural Systems with Processing*. Daniel Shiffman, 2012.

[Sme+09] 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.

[Sme+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.

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

[Sny93] 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.

[Tog+11] 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.

[Wol02] Stephen Wolfram. *A new kind of science*. Vol. 5. Wolfram media Champaign, 2002.

[Wol84] Stephen Wolfram. “Cellular automata as models of complexity”. In: *Nature* 311.5985 (1984), p. 419.

List of Figures

List of Tables

List of abbreviations and acronyms

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

CA Cellular Automaton. A simulation consisting of cell objects.

PCG Procedural Content Generation. An automated process of creation.

TO DO: (?)

Attachments

1. List of To Do Notes
2. TO DO: include thesis defence documents
3. TO DO: ?
4. TO DO: ?
5. TO DO: ?

list of todos - remove this before submitting thesis

Todo list