HAW
HAMBURG

BACHELORTHESIS
Laura Westfalen

Procedural Generation of Buildings with Wave Function Collapse and Marching Cubes

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Laura Westfalen

Procedural Generation of Buildings with Wave Function Collapse and Marching Cubes

Bachelor Thesis handed in as part of the final examination

Course of studies: Applied Computer Science
Department: Computer Science
Faculty: Engineering and Computer Science
Hamburg University of Applied Science

Supervisor: Prof. Dr. Philipp Jenke
Second supervisor: Prof. Dr. Stefan Sarstedt

Handed in on: November 27, 2020

Bachelorarbeit eingereicht im Rahmen Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Abgegeben am: 27. November 2020

Laura Westfalen

Thema der Arbeit

Procedural Generation of Buildings with Wave Function Collapse and Marching Cubes

Stichworte

Wave Function Collapse, Marching Cubes, Prozedurale Content Generierung, Prozedurale Gebäude Generierung, Textur Synthese, Computergrafik, Unity, C#

Kurzzusammenfassung

In dieser Arbeit wird eine Möglichkeit erarbeitet, mit den Algorithmen Wave Function Collapse und Marching Cubes prozedural Gebäude zu generieren. Dafür werden 3D-Modelle als Grundlage verwendet, welche mithilfe der Algorithmen angeordnet werden. Die Generierung wird von Nutzer*innen gesteuert, die bestimmen, auf welche Weise die Gebäude positioniert werden und welche Formen sie annehmen sollen.

Laura Westfalen

Title of the paper

Procedural Generation of Buildings with Wave Function Collapse and Marching Cubes

Keywords

Wave Function Collapse, Marching Cubes, Procedural Content Generation, Procedural Building Generation, Texture Synthesis, Computer Graphics, Unity, C#

Abstract

In the scope of this thesis, a solution for procedurally generating buildings with the algorithms Wave Function Collapse and Marching Cubes is developed. For this, 3D models are used as a basis, which are arranged with the help of the algorithms. The generation is controlled by users, who determine the positions and shapes of the buildings.

Table of Contents

1 Introduction 1

1.1 Motivation 1

1.2 Goals 2

1.3 Structure of the work 3

2 Basics and Related Work 4

2.1 Procedural Content Generation 4

2.1.1 Definition 4

2.1.2 Taxonomy of PCG 5

2.1.3 General PCG Techniques 6

2.1.4 Procedural Generation of Buildings 10

2.2 Texture Synthesis 15

2.2.1 Basics 15

2.2.2 Examples of Further Developments in Texture Synthesis 17

2.3 Wave Function Collapse 18

2.3.1 The WFC Algorithm 19

2.3.2 Examples of Further Developments of Wave Function Collapse 22

2.4 Marching Cubes 23

2.4.1 The Marching Cubes Algorithm 23

2.5 Combining Wave Function Collapse and Marching Cubes 27

3 Concept 29

3.1 Overview 29

3.2 Modifications to the Marching Cubes Algorithm 29
3.2.1 Basic Process 30
3.2.2 Cube Creation at Runtime 30
3.2.3 The Internal Grid 30
3.2.4 Node States Represent Adjacent Cells 31
3.2.5 Equivalence Classes 32
3.2.6 Note on the Utility Program 33
3.3 Modifications to the Wave Function Collapse Algorithm 35
3.3.1 The Wave Stores Decision Variables 35
3.3.2 Observe 36
3.3.3 Propagate 38
3.3.4 Optimization of WFC 42
3.4 Combining Wave Function Collapse and Marching Cubes 43
3.4.1 Basic Process 43
3.4.2 Integration of MC and WFC 44
3.4.3 Placing the Tiles 44
4 Implementation 45
4.1 Analysis 45
4.1.1 Stakeholders 45
4.1.2 User Stories and User Story Map 46
4.1.3 Use Cases 46
4.1.4 Overview of User Interactions 48
4.1.5 Quality Assurance 48
4.2 Software Design 50
4.2.1 Component Diagram 50
4.2.2 Class Diagram 51
4.2.3 Patterns 52
4.2.4 Sequence Diagrams 54
4.3 Tools 56
4.3.1 Agile Software Development 56
4.3.2 Technical Means 57
5 Evaluation 59

5.1 Goals 59
5.2 Requirements 60
5.2.1 Maintainability and Extensibility 60
5.2.2 Usability 62
5.2.3 Properties of Generated Buildings 64
5.3 Possible Improvements 65
6 Conclusion and Perspective 67
6.1 Conclusion 67
6.2 Perspective 67
7 Bibliography 69
8 Appendix 73
8.1 A1: Code Churn 73
8.1.1 List of All Commits 73
8.1.2 List of First Commits 81
8.1.3 List of Latest Commits 85
8.2 A2: Basic Equivalence Classes 88

Table of Figures

Figure 1: Official screenshot of the city building game *Townscaper* [2]. 2
Figure 2: A plant generated with an L-System using re-writing rules with a growing number of iterations [7]. 7
Figure 3: The diamond-square subdivision as described by [12]. 8
Figure 4: The hybrid model for modelling a complex terrain with overhangs [14]. 9
Figure 5: Example of using a simple 2D shape rule in multiple iterations, based on: [23]. ... 11
Figure 6: A simple 2D example for a split grammar. The white boxes represent non-terminal shapes, and the colored boxes represent terminal shapes. The start shape gets split into 4 façades, which are further split in different elements [24]. 12
Figure 7: The final model resulting from the derivation of the split grammar shown in Figure 6 [24]. 12
Figure 8: A procedural model of Pompeii, generated with CGA Shape [25]. 13
Figure 9: View of a 3D model generated by the method of Merrell et al. [29]. 14
Figure 10: Example for a more stochastic texture on the left side and a more regular texture on the right side [33]. 16
Figure 11: The large building complex on the right side was generated with model synthesis using the smaller model on the left [30]. 17
Figure 12: A model consisting of four model pieces can be seen in (a), (b) shows an inconsistent model that was generated without rules, and (c) shows a consistent model [30]. 18
Figure 13: Simple example for an input image [4]. 19
Figure 14: All 2x2 patterns derived from the input image above, including rotation and reflection [4]. 19
Figure 15: 9 options how 2x2 patterns can be positioned together [4]. 20
Figure 16: One part of the index data structure, describing how other patterns can be positioned together with the pattern in the middle panel (0,0) [4]. 20
Figure 17: All 14 possibilities how a cube can be intersected by a surface [3]. 25
Figure 18: Different 8-bit node states determine the appropriate building tiles [41]. 28

Figure 19: A node and the corners of its 8 adjacent cells. The numbers represent the positions of the corresponding cell states within the 8-bit index of the node state. E.g. if the cells at position 2 and 8 are “inside”, the node state would be: 00001010. 31
Figure 20: The node state 01000000 corresponds to a 3D tile that fills one eights of a cell. Together with other tiles of neighboring nodes a complete building is created..........32
Figure 21: These node states (01000000 and 10000000) are equivalent and belong to the same equivalence class. Thus, the same 3D tile can be used for them with different rotations. 34
Figure 22: To give an example: The basic equivalence classes 3 and 7 comprise four respectively six sub-equivalence classes that are shown here as 3D tiles. Each sub-class comprises four different node states – the possible rotations around the Y-axis.........34
Figure 23: The red and green nodes are neighbors and do not correspond to the equivalence class 0. However, the green node is not a direct neighbor of the red node as they do not share a common cube. 39
Figure 24: The lower left front corner of a cube belongs to this node. Since the lower left front corner of a cube defines the cube’s position, the position of the cube matches the position of the node. 40
Figure 25: A simple user story map of the application. 46
Figure 26: The component diagram shows the two layers of the project, UI and Logic. 50
Figure 27: The colors in this class diagram mark the different components. 51
Figure 28: This sequence diagram gives an overview of the entire process of the application after one mouse click into the visual grid..54
Figure 29: The sequence diagram shows the basic process of the Wave Function Collapse function. The process is triggered by the public method *ApplyWFC*. 55
Figure 30: The final building models and refurbished surroundings in the game. 57
Figure 31: The two buildings would look the same if MC was used on its own. Through WFC they differ. 59
Figure 32: For a better usability, interactions with the UI are kept simple: Buildings can be generated on the island, colors can be selected, and the application can be ended. Users can also move and zoom the camera. 63
Figure 33: Buildings are set at the outer edges of the designated area in the game. 64

1 Introduction

During the evolution of computer games, they have become increasingly complex. Game worlds are becoming larger, environments more detailed, and the graphics more and more realistic. This not only leads to increasing demands on storage and performance capacities, but also to a higher demand for working hours that are required for the development. [1]

A possible solution, which can reduce the memory requirements for the clients as well as the working hours required for production, is Procedural Content Generation (PCG). Reducing memory requirements is less important these days. At the beginning, however, this was the decisive argument for PCG. Furthermore, PCG is also used for artistic purposes.

For the above-mentioned reasons, PCG has played an increasingly important role in computer games and other graphical applications. Not only textures, vegetation, and other game elements can be generated automatically but also entire levels, dungeons, street maps, and cities.
Another related field of research is Texture Synthesis. While PCG generates new content with the help of algorithms, Texture Synthesis uses existing textures to produce similar results.

1.1 Motivation

One field of research in Procedural Content Generation is the automatic generation of buildings. This area of PCG is also often used outside of gaming applications, e.g. in architecture software.

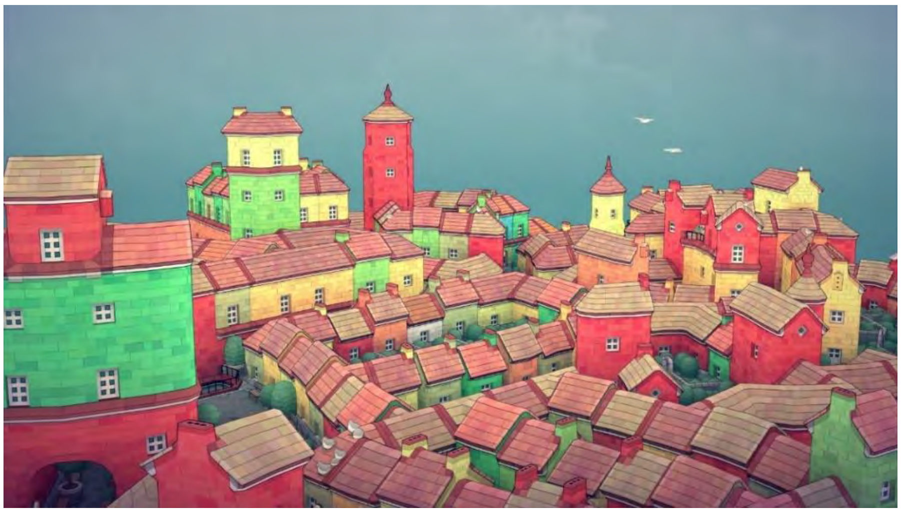


Figure 1: Official screenshot of the city building game *Townscaper* [2].

A recent game that uses building generation as its main element is *Townscaper* by Oskar Stålberg [2]. Stålberg combines methods from PCG with Texture Synthesis to let the users create versatile buildings by simply clicking on a 3-dimensional grid. More precisely, he combines the Wave Function Collapse (WFC) algorithm from the area of Texture Synthesis with the Marching Cubes (MC) algorithm that belongs to PCG.
The procedural building generation in *Townscaper* was an inspiration for this thesis. The application directly involves users in the generation process and has hardly any hurdles for users of all ages thanks to an intuitive usability.
Stålberg only made hints about his actual implementation, e.g. that he uses WFC and MC.
Therefore, this thesis examines how the algorithms WFC and MC can be combined to allow users to generate buildings in a 3-dimensional grid in a similarly simple way.

1.2 Goals

As part of this bachelor thesis, an application for procedural building generation is created, which is based on Wave Function Collapse and Marching Cubes. For this purpose, the two algorithms are modified and combined. Thus, an alternative use of the algorithms is presented. The buildings are generated in 3D on tile level, i.e. previously created 3D models are assembled to create coherent buildings.

The MC algorithm is extended in such a way that it can be used for user-initiated content generation. The original MC algorithm uses existing 3D data, interprets its values at different positions and generates a 3-dimensional surface based on these values (see chapter 2.4, and the original paper [3]). In my modification of MC, users determine the course of the surface by selecting and deselecting cells in a 3D grid. The generated surface encloses the selected cells.

Wave Function Collapse originally generates textures that are based on an input texture. The patterns that occur in the input image are used and reorganized for the new output textures. However, especially the constraint solving of WFC allows the algorithm to be used for various purposes, including the generation of 3D content. [4]
My adapted WFC expands the possibilities of Marching Cubes. While MC finds exactly one solution for certain “enclosed” cells, my adapted WFC enables different possible solutions for the same combination of cells. More precisely, MC would always generate the same building for the same cell combination; together with WFC, different building façades appear. WFC examines the different possibilities for a particular cell combination and consequently decides on how exactly the cells are enclosed, using constraint solving.

Hence, instead of reassembling individual patterns from a texture, my modified WFC assembles individual building sections in such a way that they form the surface that MC has recognized.

1.3 Structure of the work

This thesis consists of the following 6 sections:

1. Introduction
2. Basics and Related Work
    - Basic methods of PCG and Texture Synthesis are presented as well as a selection of current papers on this, Marching Cubes, and Wave Function Collapse. In addition, the procedural generation of buildings is discussed in more detail. Eventually, the combination of WFC and MC is discussed.
3. Concept
    - The logical concept of the application that is implemented in the context of this thesis is presented.
4. Implementation
    - Software engineering aspects in terms of analysis, software design, and quality are discussed as well as tools and libraries.
5. Evaluation
    - The development of the application will be evaluated and examined regarding the goals, the requirements, and software quality.
6. Conclusion and Perspective
    - A conclusion will be drawn and possible perspectives for further developments are shown.

2 Basics and Related Work

2.1 Procedural Content Generation

This chapter overviews the topic Procedural Content Generation (PCG) and is intended to provide an understanding of the topic by comparing definitions and boundaries set up by different authors. A taxonomy of PCG will be discussed and this work will be classified according to it. Subsequently, related work and developments of different techniques in PCG are first presented in general and then focused on the generation of buildings.

2.1.1 Definition

Togelius et al. define PCG as follows: “[PCG is] the algorithmical creation of game content with limited or indirect user input” [5]. Shaker et al. add that “PCG refers to computer software that can create game content on its own, or together with one or many human players or designers” [1]. It should be noted that the above-mentioned authors focus on PCG in games, although many other areas of application are conceivable. While some authors assume PCG is exclusively used in games, Hendrikx et al. distinguish PCG and PCG-G (Procedural Content Generation for Games) [6]. Freiknecht et al. name a few examples for a broader use of PCG like simulations, urban planning and animation movies [7]. Compton et al. also point out that the understanding of PCG is often reduced to the field of games, although other fields like generative art, music, and design, are equally interesting research areas. Therefore, they use the term “generative methods” to clarify that they refer to all possible areas of application [8]. Nevertheless PCG-G and PCG are often used interchangeably and, in the following, PCG will be used as a generic term, including both PCG-G and PCG for other applications.

In this context, it is also important to define *content*. Shaker et al. summarize it as follows: “[C]ontent is most of what is contained in a game: levels, maps, game rules, textures, stories, items, quests, music, weapons, vehicles, characters, etc” [1]. Again, they focus on PCG-G, Freiknecht et al. give a broader definition and describe *content* as “digital assets for games, simulations or movies” [7]. In this thesis a broad understanding of *content* will be used, namely as digital assets for any kind of applications.

Authors draw different distinctions when it comes to the question of what can and cannot be counted among PCG.

On the one hand, according to Shaker et al. “a map editor for a strategy game that simply lets the user place and remove items, without taking any initiative or doing any generation on its own” cannot be PCG [1]. Togelius et al. also write that “if the human input to the content generator is part of a game, and the player directly intends to create content in the game, it is not procedural content generation”. Therefore, according to Togelius et al., the purpose of the application also determines whether something is PCG or not. Furthermore, they distinguish content creation which is completely led by users (not PCG) from content generation that may be initiated by users but cannot be controlled by them completely (PCG) [5].

Hence, the generation of buildings by only using the Marching Cubes algorithm would not be considered PCG by neither of the above-mentioned authors. The combination of Marching Cubes (MC) and Wave Function Collapse (WFC) for building generation, however, might still be seen as PCG since the WFC algorithm introduces a non-deterministic element. Still, Togelius et al. might disagree if the main purpose of the application was the generation of buildings.

On the other hand, Freiknecht et al. take a completely different view of the limits of the subject. They see this topic much more comprehensively and call all kinds of modeling and designing mechanisms PCG that go beyond “[moving] single vertices”, e.g. the character editor in the game “The Sims” that “offer[s] a set of parameters to edit single characteristics of a human to e.g., adapt the waist, size or the interocular distance.” The authors call this “procedural content generation with a strong user interaction” [7]. According to this definition, even building generation with only Marching Cubes would count as PCG. This thesis will follow the latter understanding of the limits of PCG.

2.1.2 Taxonomy of PCG

Shaker et al. [1] depict a taxonomy of PCG based on the work of Togelius et al. [9]. They distinguish the following approaches:

- Online vs. offline

Online PCG is the generation of content during runtime, e.g. the generation of endless landscapes in a game, whereas offline PCG refers to content generation during the development of an application or game.

- Necessary vs. optional

Necessary generated content is required for the usage of an application or game and should always be correct. Optional content can be discarded or exchanged by users.

- Degree and dimensions of control

The generation of content can be controllable to different extents. On the one side, Shaker et al. and Togelius et al. name the usage of a random seed and, on the other side, they mention multiple parameters that can be used to create and control the generation of content.

- Generic vs. adaptive

Generic PCG does not take the users' behavior into account. Adaptive PCG, however, is personalized and adapts to the user.

- Stochastic vs. deterministic

Deterministic PCG always creates the same content with the same input parameters and thus allows the recreation of content. Stochastic PCG is not predictable to this extent.

- Constructive vs. generate-and-test

Constructive PCG generates content entirely, whereas the generate-and-test version generates and tests individual parts repeatedly.

- Automatic generation vs. mixed authorship

Mixed authorship PCG includes the aid of users and/or designers within the generation process. Automatic generation allows only limited input respectively mainly the adjustment of parameters by the developers.

The software project presented with this work can therefore be classified as follows:

The application uses *online* PCG as the generation happens during runtime. The generation is *necessary* as the generated content is required for the use of the application. However, this category may not be applicable in our case since the actual purpose of the application is to generate content and for this reason the question of the necessity does not arise. The *degree and dimensions of control* are relatively low but of course the users have some control. They decide where content (in our case buildings) should be generated. Furthermore, users can decide about the colors, the program is using for its generation process. All other decisions are made by the program. Furthermore, the application uses *generic* PCG – the users' behavior does not influence how the program works, i.e. the program is not adaptive. Using the WFC algorithm gives the application *stochastic* behavior. The results of the generation are not fully predictable and are random to some extent. Besides, the program uses *constructive* PCG as every input results in a complete building (that can of course be expanded). Also, this program uses *mixed authorship*. The users' influence is limited but the generation does not happen automatically either.

2.1.3 General PCG Techniques

In the following, important PCG techniques are presented that are meant to generate different kinds of content like landscapes, terrain, or plants.

Plant Generation with L-Systems

One way to generate content is the use of grammars. An early approach to grammars and string rewriting was published by Noam Chomsky in the 1950s. And in 1968 Aristid Lindenmayer formulated a grammar called Lindenmayer-Systems (or L-Systems) to describe the structure of plants. In the field of content generation, L-Systems became particularly important. Chomsky's formal grammars differ from L-Systems in that his grammars productions are applied sequentially, while the replacement of symbols in the L-Systems takes place simultaneously. The mapping of L-systems to graphics is called "Turtle Model" [7] [10].
A simple version of the L-Systems defines a certain number of symbols that describe how a line on a 2-dimensional grid shall be drawn, e.g. the symbol "F" means to move one step in the current direction on the grid and draw a line, the symbol "+" means to turn right by a certain degree and so on. In addition, L-Systems define re-writing rules that determine how simple elements are replaced by more complex ones (E.g. "F" can be replaced by "F+F"). It is possible to apply the replacement rules as often as desired and therefore create pictures with different levels of detail. Furthermore, it is possible to use a stack and go back to a certain point in the drawing [7] [10].

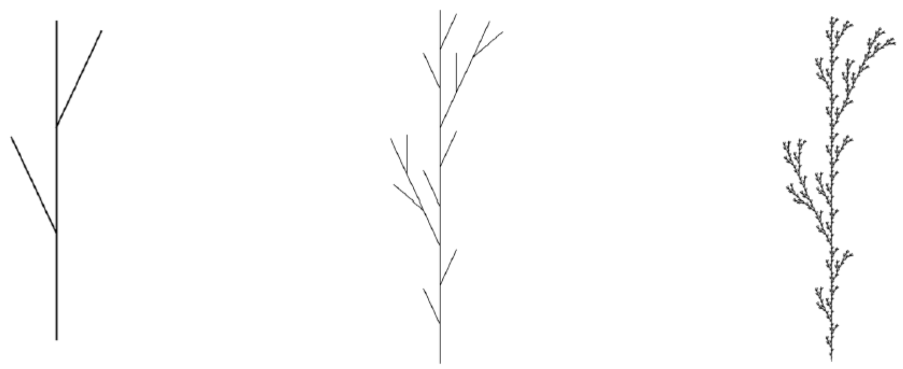


Figure 2: A plant generated with an L-System using re-writing rules with a growing number of iterations [7].

Extensions of the Lindenmayer-Systems include methods for the creation of 3-dimensional plants like the "cellwork L-Systems" [10]. Other methods for visualizing natural structures are the Mandelbrot set, the Koch snowflake, or the Pythagoras tree [7].

Terrain Generation

A terrain or landscape can be formed by height maps that define the current height for every field on a 2-dimensional grid [7]. Among the first algorithms for height map creation are subdivision-based methods, where a map is iteratively subdivided with a certain degree of randomness [11]. One approach, described by Gavin Miller in different variations, is mid-point displacement [12]. Mid-point displacement iteratively uses points of corners of a triangle or in a diamond shape and assigns a (more or less) random height to these points based on the average of its neighbors. A simple mid-point displacement method can be used in 2D where points on a line are iteratively selected and elevated. The extended diamond-square algorithm can be used for 3D terrains: With each step a 2-dimensional square field is divided by a point that is set in its center. The height of this point is made up of the mean height of the 4 surrounding points plus a random value.

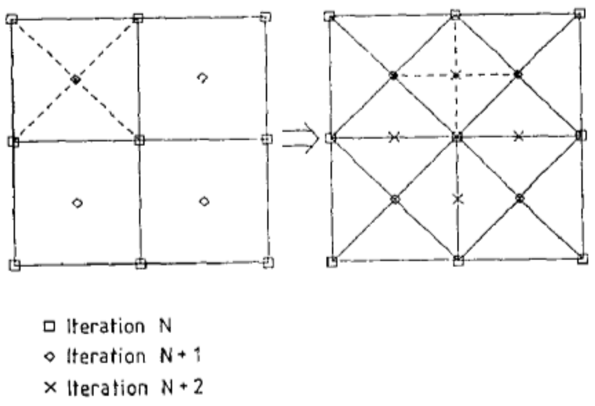


Figure 3: The diamond-square subdivision as described by [12].

Techniques to procedurally generate height maps often include the use of fractal noise (like Perlin Noise) to make the height shifts look more realistic. Ken Perlin introduced an approach for “an interactive synthesizer for designing highly realistic Computer Generated Imagery” [13]. His algorithm allows the creation of controlled stochastic effects and was e.g. used for generating clouds, wood, or water. In comparison to earlier techniques Perlin’s method proved to be faster and easier [13]. Applied on height maps, the generated results became more realistic and mountainous [11].

However, height maps do not allow the creation of caves or overhangs. This can be achieved by voxel terrains or layered terrains [7]. Peytavie et al. present an approach to modelling complex terrains using layers from different materials to create arches, overhangs, and caves [14]. Their “hybrid model” uses a discrete volumetric data structure for the different materials as well as an implicit surface model for the sculpting of the surface of the terrain. The discrete data structure bases on a 2-dimensional grid of material layers and the surface model is defined as a convolution surface.

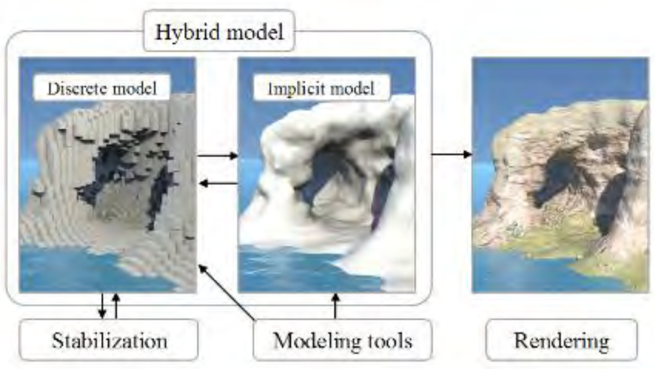


Figure 4: The hybrid model for modelling a complex terrain with overhangs [14].

Dungeon Generation

One approach to procedural dungeon creation is binary space partitioning. This algorithm divides a 2D or 3D space into cells by applying the same algorithm recursively on any new cell. One cell is divided into two sub-cells, respectively. In this way, a hierarchical tree structure is created [1].
Another common approach to dungeon creation is cellular automata. This method uses a grid of cells (in any dimension) where each cell holds a reference to a set of neighbors and to an initial state. Rules determine how the state of a cell and its neighbors is transformed into another state. After several transitions different patterns can be formed. Possible allowed states could be “wall” and “path” so that a dungeon can be created [15].
Grammars can also play a role in the creation of dungeons, e.g. graph grammars and shape grammars (the latter topic will be examined in chapter 2.1.4). Graph grammars generate topological descriptions as a graph where nodes represent rooms and edges adjacencies [15].
Further methods that can be used for dungeon creation are genetic algorithms and constraint-based approaches [15].

PCG with Constraints

Another way to procedurally generate content is to use constraints. (The Wave Function Collapse algorithm partly also belongs to this category and will be discussed chapter 2.3)

Horswill and Foged present an approach where they use constraint propagation for creating enemies and objects in games. They use “path constraints” that regard the possible paths a player can take and that can place objects effectively. E.g. they allow the system to balance the number of enemies according to the player’s level or to ensure that “keys are not hidden behind the doors they are intended to unlock, and so on” [16].

Another possible constraint-based method for PCG is Answer Set Programming (ASP), “a form of logic programming targeted at modeling combinatorial search and optimization problems” [4].

Smith and Mateas used ASP in the field of PCG for an explicit description of the design space. Thus, the space of content is declared, and a domain-independent solver uses this space to generate the desired content. Their approach also allows new generators for different game content domains to be quickly defined [17].
Neufeld et al. used ASP for procedural level generation. They developed an automatic way of evolving level generators for 2D games. A game gets transformed in a set of rules that are described in ASP. In this way, multiple levels can be generated that will be evaluated afterwards [18].

The use of ASP in PCG already shows similarities to the WFC algorithm (see 2.3). ASP in PCG aims to model the space of content beforehand to subsequently let a constraint solver use these models to create new content [4]. This means, the solver arranges the existing models to generate new content by using constraints to check how the models can be connected to one another. This is also the basis of what WFC does.

Further PCG Methods

There are many other methods for automatically generating content and of course only a few can be mentioned here.
One important aspect that is often mentioned in this subject area (and that should also be mentioned briefly here) is the customization of procedurally generated content to the player. Yannakakis and Togelius introduce a framework for experience driven PCG that take the user’s experiences into account to generate optimized content. E.g. they created personalized levels for the game Super Mario. While players were playing, the game recorded different metrics, e.g. the frequency of jumping and running but also the time users were moving as opposed to the time they spent standing still as a measurement of the “fun” the users had. This data is fed to a neural network and used to create personalized levels with variable degrees of difficulty [19].

2.1.4 Procedural Generation of Buildings

The following chapter focuses on various approaches to procedural generation of buildings and is intended to put this work into context.

According to Smelik et al., procedural building generation is one of the best developed areas in PCG [20]. Some of the previously mentioned methods (such as the L-Systems) are also used in building generation. However, especially split and shape grammars play an important role in this area. Like the L-Systems, they also represent formal rewriting systems. Many building generation tools use these grammars as a basis.

Split and Shape Grammars

The concept of shape grammars was introduced by Stiny in the 1970s. These grammars operate on shapes and are useful in the construction of architectural designs. [21] [22] Shape grammars consist of shape rules (or spatial relations) that define how an existing shape can be transformed into another shape (or shapes). The rules also determine when a transformation can be applied [20]. The final models are created by applying these rules recursively.

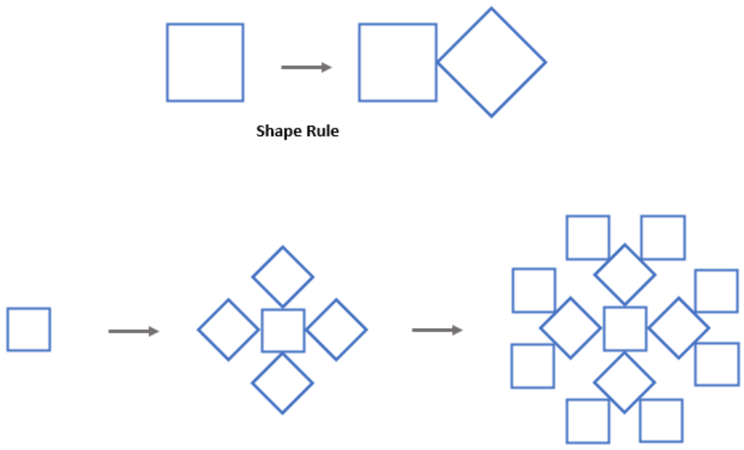


Figure 5: Example of using a simple 2D shape rule in multiple iterations, based on: [23].

Wonka et al. introduce a type of parametric set grammar, split grammar, that is used for the automatic modeling of architecture. Their approach is based on the concept of shape and draws from Stiny's work on shape grammars [24]. Split grammar restricts the types of allowed rules so that it is still able to model buildings but also simple enough to allow an automatic derivation of the grammar. It is a 3-dimensional design grammar and determines the spatial layout of a building. So-called basic shapes (like e.g. cuboids or cylinders) are manipulated by the grammar by means of a set of certain functions that define the allowable affine transformations for the shapes [24]. According to Smelik et al. split grammars focus on generating façades for simple shaped buildings [20].

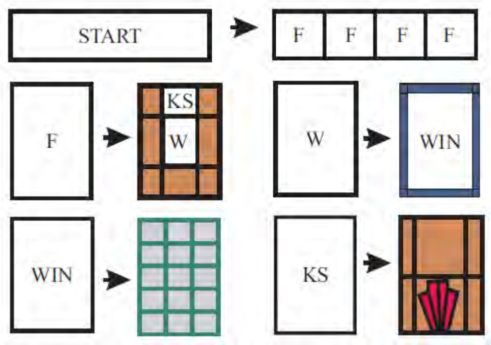


Figure 6: A simple 2D example for a split grammar. The white boxes represent non-terminal shapes, and the colored boxes represent terminal shapes. The start shape gets split into 4 façades, which are further split in different elements [24].

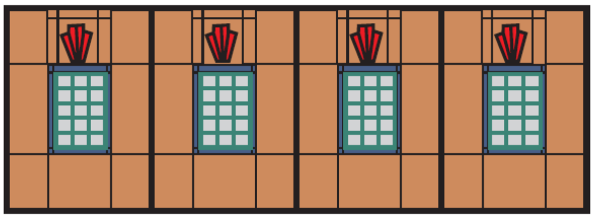


Figure 7: The final model resulting from the derivation of the split grammar shown in Figure 6 [24].

Extensions of Split and Shape Grammars

Computer Generated Architecture (CGA) by Müller et al. is an extension of the split grammar by Wonka et al. and belongs to the shape grammars [25]. The CGA shape grammar (CGA Shape) is used for 3D modeling of buildings and it also allows to create roofs and rotated shapes with arbitrary orientation. CGA Shape usually starts with extruding a polygon into a 3-dimensional shape and creating several floors. Afterwards, the resulting façades are subdivided by shape rules. The possibility to split a shape (with a split rule) was already presented in earlier works, but Müller et al. introduced the repeat split and the component split, which allows to work jointly with 1-, 2-, and 3-dimensional shapes. The notation of the grammar's rules to add, scale, translate, and rotate shapes was inspired by the L-Systems. However, the L-Systems represent parallel grammars (and can capture growth over time), whereas CGA Shape works sequentially [25].

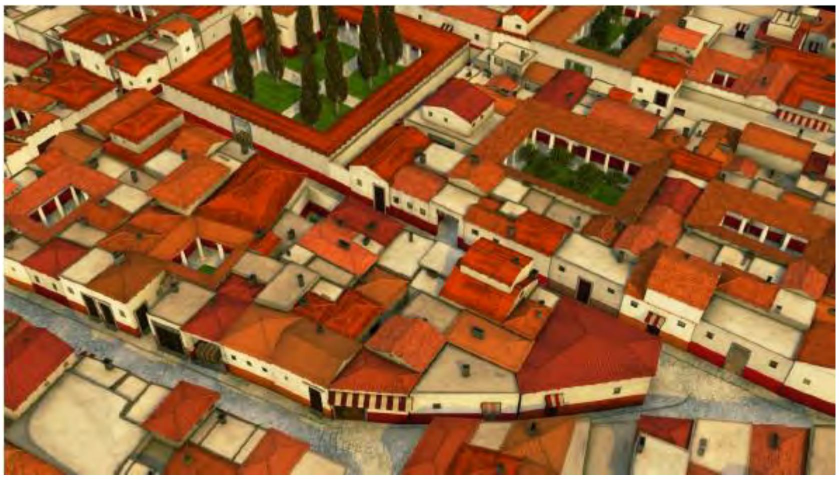


Figure 8: A procedural model of Pompeii, generated with CGA Shape [25].

Finkenzeller and Bender propose a further extension of shape grammars by capturing semantic information (regarding the role of shapes within a building) in a typed graph [26]. Their approach is designed for the creation of complex buildings and façades with high details. Their model description includes geometry and semantics and allows a partial automation of the modeling process.

Smelik et al. argue that the conventional methods that use split and shape grammars may create convincing results in building generation but require much authoring effort [20]. Another method that solves this problem is an automatic reconstruction of building façades by Müller et al. that uses image analysis to automatically derive 3D models from photographs. However, they still use shape grammars for their procedural modeling pipeline [27].

Further Building Generation Methods

Birch et al. introduce an approach for procedural-modeling of large-scale urban virtual-environments using constraints [28]. For the creation of large scenes, Birch et al. combine different tools and libraries for different modeling tasks (e.g. for basic building structures, windows, etc.). Each modeling tool uses a scene graph as a basis and a set of constraints to define the different building structures. The basis for their building generation process is a simple cube – the shell that can be modified by permitted actions and constraints. The permitted actions include the moving of single vertices, the expansion or contraction of opposing faces and moving the shell. Constraints that can be imposed on the shall allow to configure how the shell behaves in relation to other shells and to set editing options. Roofs and other external objects can be added to the buildings in the same way.

Merrell et al. present an approach for the procedural generation of building layouts [29]. Their architectural program is generated by a Bayesian network that was trained on real-world data. The program is realized in a set of floor plans, that are obtained through stochastic optimization. The floor plans can then be used to construct a 3-dimensional model of a building. Their approach goes beyond that of shape grammars, as they regard more than just pure shapes and their arrangements. They also take aspects into account like functional relationships between spaces or the “privacy gradient”, considering that common areas should be closer to the door and private rooms should be farther away. As all those architectural rules can hardly be implemented by hand, they used machine learning techniques to derive them from data.
The input to their application is a list of requirements, e.g. the number of bedrooms, bathrooms etc. These requirements are then expanded into an architectural program, containing a list of rooms, their adjacencies, and their target sizes. This architectural program is generated by a Bayesian network, that was trained on a corpus of real-world architectural programs, i.e. on a catalogue of residential layouts. The generated architectural programs can then be turned into building layouts, i.e. detailed floor plans for every floor. Finally, a given building layout can be used to generate a 3D model, decorated in many different styles. The possible styles are specified in style templates, listing the properties of every building element like windows, doors etc.

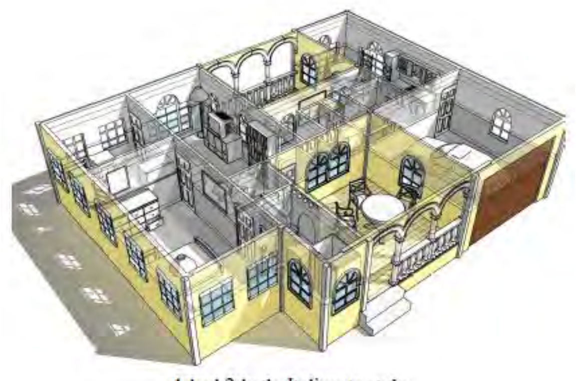


Figure 9: View of a 3D model generated by the method of Merrell et al. [29].

In his paper “Example-Based Model Synthesis” Paul Merrell also extended texture synthesis methods to an application that allows the generation of large 3D building models from smaller models [30]. His approach is further discussed in chapter 2.2.2 on texture synthesis.

Examples for the Use of Procedural Building Generation

Saldana and Johanson used procedural building generation to recreate historical cityscapes of Rome [31]. Beside procedural generation techniques, they used geographic information systems (GIS) for archaeological data and the game engine Unity to combine the

procedurally generated buildings and terrain. The architectural data contains time references so that the development of the cityscape over several centuries could be reconstructed.
For the building generation they used ArcGIS CityEngine, a design software for 3D cities that uses GIS data to procedurally generate e.g. buildings and terrain. Using this software, they defined architectural attributes and attached them to specific locations, but also assigned randomized functions to certain attributes so that they were distributed over the urban model. Various building types were described by different procedural rules, so that e.g. houses, temples, and basilicas could be distinguished and spread across the city model. Also, semantic descriptions for architectural elements were used so that different architectural styles could be created.

Greuter et al. presented an approach to procedurally generate “pseudo infinite” cities in real-time [32]. The cities consist of various procedurally generated buildings, which generation parameters are created by a pseudo random number generator. The building geometries were extruded from a set of floor plans. The floor plans are generated by an iterative process, from the top level of a building to the ground floor, combining randomly generated polygons. The first iteration generates a random polygon and subsequent iteration steps create floor plans for the lower floors by generating a random polygon and combining it with the previously created polygon. The shape of a building (i.e. the pseudo random numbers for its parameters) is determined by its location, so that the same building will always appear at the same location. Buildings that surround the viewpoint are generated and stored in memory and buildings that drop out of the viewing range will be deleted. Furthermore, they used view frustum filling to enhance performance. In contrast to view frustum *culling*, a method that detects non-visible but existing shapes before they are rendered, their approach detects potentially visible cells around the viewpoint of the camera before the content is generated.

A possible application for procedural building generation are computer games. As mentioned in the Introduction, Oskar Stålberg combined methods from procedural generation with texture synthesis and created the game *Townscaper* (see chapter 2.5).

2.2 Texture Synthesis

As the Wave Function Collapse algorithm was originally invented in the field of texture synthesis, this chapter will give an overview of this topic.

2.2.1 Basics

According to Karth and Smith, texture synthesis is the process of generating an output image with texture that resembles a smaller input image [4]. Some authors distinguish exemplar-based texture synthesis from a purely algorithmic generation of textures without input data (see [33], [30]). In the following, the term “texture synthesis” will be used synonymously for “exemplar-based texture synthesis”.

A fundamental question in this context is what *texture* means. According to Tuceryan and Jain there are many different definitions and it is difficult to find one that is universal [34]. Definitions they propose describe texture as a structure with a repetitive pattern or as a region in an image with a constant, slowly variant, or periodic set of local properties.

Furthermore, Efros and Leung point out that two types (respectively two extremes) of textures can be distinguished, between which all textures can be classified: regular and stochastic textures [35]. Regular textures consist of repeated texels (texture elements), whereas stochastic textures have no explicit texels.

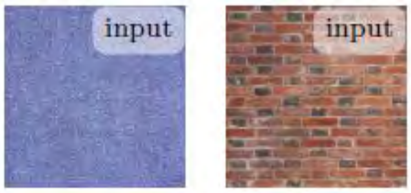


Figure 10: Example for a more stochastic texture on the left side and a more regular texture on the right side [33].

Oftentimes, the input and output images in texture synthesis are characterized by their local patterns, sub-images of just a few pixels. Many algorithms produce output in which every local pattern resembles a local pattern in the input and this resemblance need not be an exact matching. In contrast, Wave Function Collapse uses exact pattern matching. [4]

Different methods for texture synthesis have been developed that can be divided into parametric and non-parametric methods. *Parametric methods* estimate a set of statistics from an input image that define an underlying stochastic process. The output images will then be generated by this stochastic process and have the same statistics as the original texture. These methods may fail when the input image has certain structures that are important for a correct reproduction, i.e. they are more useful for stochastic textures.
Non-parametric methods reorganize neighboring local patterns to create output textures. These methods are especially useful for more regular textures but can possibly just recreate large parts of the input image or create useless output images that only reproduce one part of the input image and ignoring the rest of it. [33]

As will become clearer in chapter 2.3, the WFC algorithm belongs to the non-parametric methods, recreating and reorganizing existing patterns from the input.

2.2.2 Examples of Further Developments in Texture Synthesis

Paul Merrell proposes an extension of texture synthesis from the original 2-dimensional to a 3-dimensional application; he calls this approach *model synthesis* [30]. His method generates large 3-dimensional models that resemble a smaller 3D model that was given as input. Merrell mentions two main differences between texture synthesis and model synthesis: first, model synthesis uses a global search to find potential conflicts and, second, his method divides the task of generating a large model into several smaller tasks to make the generation easier to solve. His approach breaks the example model into small building blocks that are rearranged on a 3-dimensional grid. This method accepts many different example models, and he refers to it as a "general-purpose modeling tool".

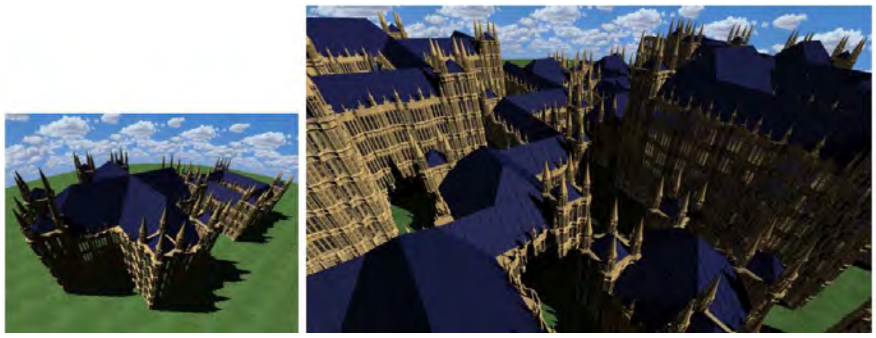


Figure 11: The large building complex on the right side was generated with model synthesis using the smaller model on the left [30].

Merrell uses predefined model pieces (building blocks of a model) to describe the input models. To achieve consistent results, model pieces must be put together using a set of rules. These rules must ensure that the pieces fit together correctly. E.g., one rule stipulates that a model piece may only be beneath (or behind) another model piece, if it was beneath (or behind) that piece in the example model. "More precisely, a model M is consistent with an example model E if all the model pieces that are adjacent to one another in M are found adjacent to one another in E along the same direction" [30].
To apply the rules correctly, Merrell uses labels for the different pieces of a model. If a model in the resulting 3D grid is missing labels, it is still incomplete.

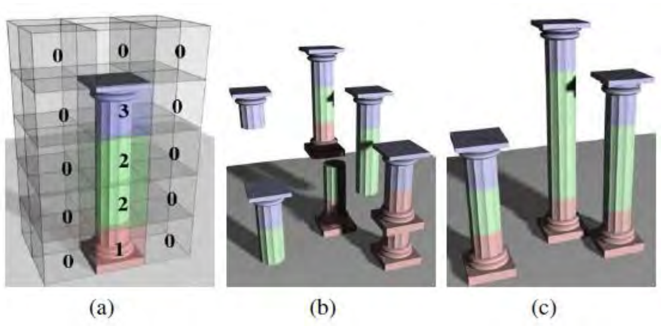


Figure 12: A model consisting of four model pieces can be seen in (a), (b) shows an inconsistent model that was generated without rules, and (c) shows a consistent model [30].

As you can see in Figure 12, this model is only complete if it contains the pieces 1, 2 and 3.
It is possible that a model becomes inconsistent when more labeled model pieces are added to it. To prevent this, Merrell uses a global search to remove model pieces that may lead to inconsistencies.

Model Synthesis shows some similarities to the Wave Function Collapse algorithm (see chapter 2.3), another development in the field of texture synthesis. In fact, Karth and Smith write that the WFC algorithm was inspired by Merrell's approach [4]. It will be seen that the rules Merrell uses to prevent inconsistencies in his results resemble the constraints of WFC. The requirement that adjacent model pieces from the input model should be equally adjacent in the result is present in both algorithms.
Further developments that are based on WFC are described in Chapter 2.3.2.

2.3 Wave Function Collapse

One of the two algorithms that forms the basis of this work is the Wave Function Collapse (WFC) algorithm[1] that was developed by Maxim Gumin. Karth and Smith describe Gumins approach in detail [4] and I will refer to their paper in the following.

WFC was originally invented to create new images based on given examples. It therefore belongs to texture synthesis as well as procedural generation. The images generated by WFC only contain local patterns that are also present in the original image. Gumin's original algorithm does not use backtracking and implements a greedy search. Karth and Smith classify WFC as a constraint solving method. WFC uses a minimal entropy heuristic and generates its output image incrementally by expanding the known regions of the output and completing them with local patterns from the input image.

[1] The name "Wave Function Collapse" is a reference to the wave function of quantum mechanics, which was a loose inspiration for the algorithm. It was mainly an inspiration for the superposition of the possible solutions for an image position during the expansion of the output image. [3]

2.3.1 The WFC Algorithm

Gumin's original algorithm as described from Karth and Smith is presented below.

In general, the algorithm executes 4 tasks:

1. Extracting local patterns from an input image
2. Processing these patterns in an index, so that constraint checking can be accelerated
3. Incrementally creating an output image by expanding a partial assignment
4. Rendering the total assignment into an image

In the following, the tasks will be described in more detail.

Regarding step 1: Extracting local patterns from an input image

A pattern is a unique combination of input tiles, it is composed of NxN tiles (e.g. 2x2 or 3x3). All patterns are extracted from the input image and can be described as sub-images of the original image. At this point two variations of the algorithm can be distinguished: the simple “tiled version” and the “overlapping version”. The two variations differ in the way in which the constraints are derived from the patterns.
The “tiled version” derives its constraints directly from the patterns so that one pattern defines exactly one allowed combination of tiles.
The “overlapping version” derives its constraints from the input image by creating a set of unique patterns. In short, the algorithm examines all patterns it got from the input image and identifies the ways these patterns can overlap (with different offsets). Patterns can overlap where they have identical tiles. A complete overlay is only possible with the same pattern, all other patterns can only overlap partially. Symmetry and reflection of the patterns can also be considered.

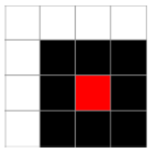


Figure 13: Simple example for an input image [4].

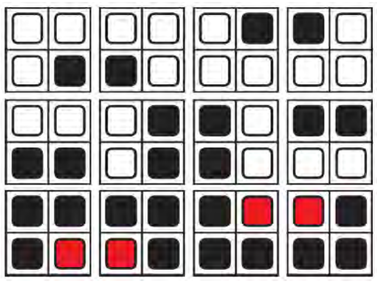


Figure 14: All 2x2 patterns derived from the input image above, including rotation and reflection [4].

Figure 13 shows an example for a simple input image. Figure 14 shows all 2x2 patterns that can be extracted from this image. It can be noted that every pattern is unique even if the pattern appears more than once in the input image. Besides, some of the patterns do not appear in the input image in exactly this way but are a rotation or reflection of a pattern.

Regarding step 2: Processing the patterns in an index, so that constraint checking can be accelerated

Based on the set of patterns an index data structure is built that describes how the patterns can be placed together.
Regarding the overlap version, the index data structure contains the information about if and how two patterns can be placed together. With 2x2 patterns there are 9 ways in which two patterns could be positioned together, see Figure 15.

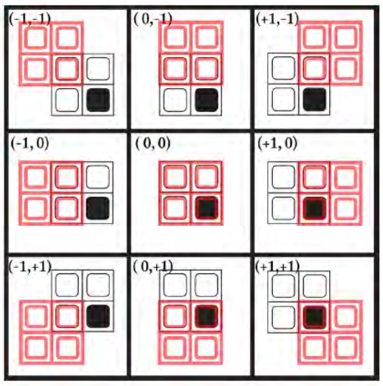


Figure 15: 9 options how 2x2 patterns can be positioned together [4].

For each pattern, the index data structure contains information on how it can be positioned together with other patterns. Gumin calls this index data structure a propagator.

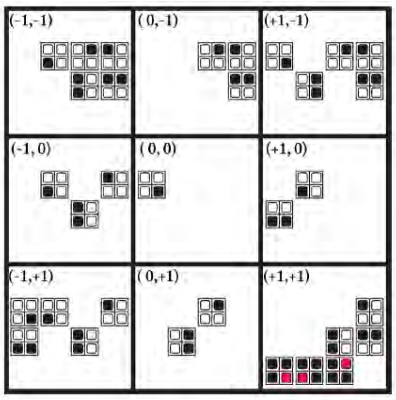


Figure 16: One part of the index data structure, describing how other patterns can be positioned together with the pattern in the middle panel (0,0) [4].

Regarding the tiled version, the index can directly be derived from the specific tile-relationships in the patterns.

Regarding step 3: Incrementally creating an output image by expanding a partial assignment

After looking at how patterns are gained from the original picture and how rules or constraints (describing the possible combinations of patterns) are derived from them, the creation of the output image is regarded now.

During the incremental generation process, decision variables are repeatedly selected and assigned. There is a decision variable for each grid location of the output image. A table, that Gumin calls Wave (a reference to the quantum wave function), keeps track of the assignments and the remaining possibilities for any location. The entries in this table (that Gumin calls coefficients) are Boolean values. These coefficients record whether the algorithm might still assign a certain pattern to a given location. In the beginning, all these variables are initialized with “true”, thus all coefficients have an unlimited domain. This means that there is one decision variable with a domain for each location on the grid. At the beginning each domain contains all possible patterns from the input image. The further the algorithm progresses, the smaller the domains of the variables become, until in the end only one possible value is left for each variable, which will then be assigned. It should be noted that Gumin’s original algorithm does not use backtracking, so if an error occurs a global restart is executed.

This third step can be divided into the sub-steps *observe* and *propagate*.

The purpose of the *observe* phase is to find the location with the lowest entropy (greater than 0). The cell with the lowest entropy is the one with the smallest domain, i.e. the variable with the fewest possible patterns that can be assigned. After the location with the lowest entropy was selected, it still got several valid patterns left, otherwise the location’s entropy would already have been 0 in the previous step, so this variable would already have been resolved.
Afterwards, one of the possible patterns of the chosen location must be selected. The pattern is chosen randomly, however the frequency with which the patterns occur in the original image are used as weights. This implements Gumin's goal that all patterns in the output image should be as frequent as in the input image.
After a variable has been assigned in the *observe* step, it receives a flag as a location that must be updated in the wave, i.e. the other variables (or their domains) must be updated by constraint propagation.

The *propagate* step uses “arc consistency”. This constraint solving method “ensures that a value only appears in a domain of a variable if there exists a valid value in the domain of related variables such that constraints over those variables could be satisfied” [4]. Updating a variable domain can mean that all neighboring variables must also be updated.
The actual propagation works like the Flood Fill algorithm. The algorithm ends when there is no cell left that must be updated. Starting from the first flagged location, every adjacent cell will be checked, whether it must update its domain. If an adjacent cell needs to be updated, it also receives a flag as needing to be updated in the next iteration.

To sum up, each observation step ends with a location receiving an assigned value. In addition, some adjacent variables are likely to get less entropy in their domains during the propagation.
If there is no more entropy in the system, i.e. all variables have a single value in their domain, the final image is created and can be returned and rendered. Alternatively, it is also possible that an image of the partial results can be output after each observation-propagation cycle.

2.3.2 Examples of Further Developments of Wave Function Collapse

Karth and Smith write, that since its release, Wave Function Collapse was especially used in the (indie) game development scene. *Proc Skater 2016* by Joseph Parker was the first game using WFC and generates levels with this method. Another game developer, Oskar Stålberg, contributed a lot to the popularization of WFC. He generalized the algorithm to broaden the possibilities for its application. Among other things, he started using WFC for 3-dimensional applications, he used performance optimizations and added backtracking [4]. Stålberg’s combination of WFC and Marching Cubes (MC is described in chapter 2.4) as in *Townscaper* [2] is described in chapter 2.5.

In the case of model synthesis, you can already see that the subject areas “texture synthesis” and “procedural generation of buildings” (or of 3D models in general) overlap. This is also the case when the WFC algorithm is extended into the 3-dimensional.

Khokhlov et al. present another method based on model synthesis and WFC that generates 3D models [36]. In contrast to model synthesis and Stålberg’s approach to applying WFC to 3D models, Khokhlov et al. apply WFC on voxels (volume elements; the 3D equivalent of pixels). They write, the advantage of their method over others that only use surfaces (i.e. polygonal meshes) for the generation is that the input model also provides information about its volume and inner structure. Thus, they can infer adjacency rules directly from the model without the need to manually define them. Instead of regarding the input’s pixels and how they form patterns in 2D, they regard its voxels and their patterns in 3D. Based on these patterns, they automatically derive the rules for the model generation, just like (standard) WFC does in 2D.

Kim et al. describe a graph-based WFC algorithm for procedurally creating 3D game content [37]. They use a graph-based data-structure in 3-dimensional world and thereby extend the WFC algorithm to work independently of a grid.

2.4 Marching Cubes

The other algorithm that forms the basis of this work besides Wave Function Collapse is the Marching Cubes algorithm. Marching Cubes (MC) was first presented by Lorensen and Cline in 1987 [3] and their paper will be referenced in the following.

The MC algorithm was originally developed for medical uses. 3D data from computed tomography (CT), magnetic resonance (MR), and single-photon emission computed tomography (SPECT) is processed to calculate surfaces that represent the boundaries between different density values within the data. Thus, the algorithm "creates triangle models of constant density surfaces". The 3D data consists of many 2-dimensional slices that are obtained through the above-mentioned medical methods and that form a 3D data array. With a divide-and-conquer approach MC generates inter-slice connectivity. Gradient information (regarding the density values) from the original data is used for shading the models. In the end, the algorithm produces high-resolution images from the generated surface models.

Medical algorithms that produce 3D images follow the same 4 steps:

1. *Data acquisition* – e.g. through CT, MR, SPECT
2. *Image processing* – to find structure within the data (needed by some algorithms)
3. *Surface construction* – to create a surface model from the data that usually consists of either voxels or polygons
4. *Display* – to display the surface

MC is a possible implementation of the third step: it creates a surface model consisting of triangles that corresponds to a density specified by the user.

2.4.1 The Marching Cubes Algorithm

The MC algorithm can be divided into two main steps:

1. Localizing the surface corresponding to a user-defined density value and creating triangles
2. Calculating the normals to the surface at each vertex of each triangle for a high-quality image of the surface

Regarding step 1: Localizing the surface

Using a divide-and-conquer technique, MC divides all 3D data into logical cubes consisting of 8 pixels, four pixels each from two adjacent slices.
For each cube, the algorithm determines the intersection with the surface and then moves on ("marches") to the next one.

To find the surface intersection for one cube, the cube's vertices are either assigned the number zero or one. "One" means that the corresponding vertex is inside the surface, i.e. the data value of this vertex exceeds or equals the density value that determines the surface. Vertices with lower density values are assigned a "zero" and are outside the surface. The surface intersects those cubes where the vertices of one edge have been assigned different values, so one vertex is inside and the other outside the surface. Put simply, the surface intersects those cubes whose vertices have been assigned different values.

As one cube consists of 8 vertices, each in one of two possible states, there are 2^8 (= 256) ways how the surface can intersect a cube. I.e., every cube has certain vertex (or node) states, and these vertex states constitute the cube state that can be represented as a number between 0 and 255. For each possible case (i.e. cube state) an index is created that bases on the states of the vertices. This leads to an 8-bit index representing the state of a cube (one bit for every vertex).
Thus, a table with 256 possibilities can be created that displays how a cube (respectively its edges) with a certain cube state is intersected by the surface. The table lists the intersected edges for all possible cube states.
However, it is not necessary to triangulate all 256 possibilities individually. Due to symmetries, the actual number of possible ways a cube can be intersected reduces to 14, as can be seen in Figure 17. On the one hand, the topology of the surface is unchanged, if the vertex states of a cube are reversed, because the same edges are intersected. On the other hand, cube states that are rotated against each other are also equivalent.

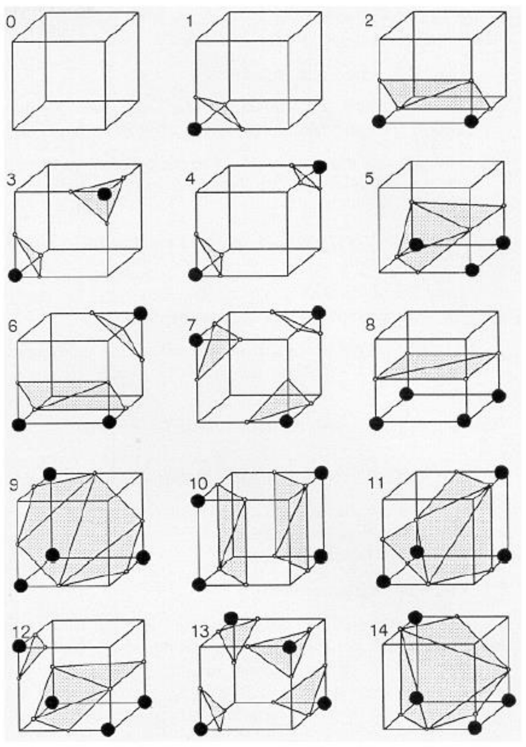


Figure 17: All 14 possibilities how a cube can be intersected by a surface [3].

Using the index of a cube as a pointer in the above-mentioned table to find the intersected edges, it is possible to linearly interpolate the surface intersection along the edge, based on the density values of the edge vertices.

Regarding step 2: Calculating the normals to the surface

Subsequently, the MC algorithm calculates a unit normal for every vertex of every triangle. These normals are used by the rendering algorithms to produce Gouraud-shaded images.

A surface with a constant density has a zero gradient component along the surface tangential direction. Hence, the direction of the gradient vector $\vec{g}$ is normal (orthogonal) to the surface. This fact can be used to calculate the surface normal vector $\vec{n}$ (if the gradient's magnitude is not zero, which is always the case, since the surface indicates the limit of different density values).

The gradient vector $\vec{g}$ represents the derivative of the density function at position (x, y, z).

$$\vec{g}(x,y,z) = \nabla\vec{f}(x,y,z)$$

Equation 1 [3]

To approximate the gradient vector of the surface, the gradient vectors at the cube vertices are estimated and, afterwards, the gradient at the point of intersection is obtained through linear interpolation.
To estimate the gradient at a cube vertex (i, j, k), central differences along the coordinate axes is used (see Equation 2, Equation 3, and Equation 4). So, a vertex gradient consists of (G_x, G_y, G_Z).

$$G_x(i,j,k) = \frac{D\ (i+1,j,k) - D\ (i-1,j,k)}{\Delta\ x}$$

Equation 2 [3]

$$G_y(i,j,k) = \frac{D\ (i,j+1,k) - D\ (i,j-1,k)}{\Delta\ y}$$

Equation 3 [3]

$$G_z(i,j,k) = \frac{D\ (i,j,k+1) - D\ (i,j,k-1)}{\Delta\ z}$$

Equation 4 [3]

D (i, j, k) is the density at pixel (i, j) in slice k and Δx, Δy and Δz are the lengths of the cube edges. The gradient is divided by its length to receive the unit normal at the vertex.
To calculate the gradient at all vertices of a cube, it is necessary to keep four data slices in memory at once.

After obtaining the normal gradient vector for every cube vertex (using Equation 2, Equation 3, and Equation 4), it is now possible to obtain the gradient vectors at the intersection between the surface and the cube by linear interpolation. A gradient is calculated for every triangle vertex of the surface within the respective cube.

To sum up, Marching Cubes generates a surface from 3D data as follows:
The algorithm divides the pixels of consecutive slices into cubes, consisting of four pixels each from two slices. The cube receives an 8-bit index that indicates which of its vertices are inside the surface and which are outside. A vertex is inside the surface when its density value is equal or above a user-defined threshold (the surface constant). The cube’s index is used as a pointer in a list to find the edges that are intersected by the surface. With the densities at the edge vertices the exact surface-edge intersection can be obtained by linear interpolation. A unit normal at each cube vertex is approximated using central differences. These normals are subsequently used to interpolate the normal to every triangle vertex of the surface within the cube. In the end, the triangle vertices and vertex normals are output.

2.5 Combining Wave Function Collapse and Marching Cubes

The game developer Oskar Stålberg experimented a lot with different applications for Wave Function Collapse [4], he held speeches about the algorithm [38] [39], created a browser game to demonstrate how WFC works[2], and published the strategy game *Bad North* [40] using WFC for level generation.
His latest game *Townscaper* [2] applies WFC on 3-dimensional building models and thus he created a simple city-building game where users can create single buildings or huge towns by clicking on cells in a 3D grid. For this application he used both the WFC and the MC algorithm that he applies on a modular tile set. His approach was an inspiration for this thesis.

Stålberg wrote about the functionality of *Townscaper*: "Marching Cubes generates the possibility space that WFC then collapse[s]."[3] For a better understanding of this statement, I will first look at how MC can be used for simple building generation purposes in 3D. Afterwards, I will regard how WFC can be used as a constraint solver in order to use concrete 3D tiles for the generation.

Stålberg did not publish specific information about his own implementation of *Townscaper*, which is why the procedure described below is only an interpretation of his statements. The details of my own approach are discussed in chapter 0.

A modification of MC for building generation purposes can work like this: A 3D grid consists of cells that are either "inside" or "outside" (just like in the original MC, where pixels from a slide can either be inside or outside a surface). When a cell is "inside", a building (or part of a building) is generated at the corresponding location. It is important to consider in which way "inner" cells adjoin each other, since the aim is to create consistent buildings. Depending on how the cells are adjacent, different building tiles are used. A browser game for building generation that Stålberg developed[4], uses "the corners between blocks" (i.e. the nodes of the grid) to place the building tiles [41]. Apparently, he used the nodes of the grid to keep track of the state of their 8 adjacent cells. Like in the original MC algorithm, an 8-bit index represents a state. But instead of a cube state that represents the state of its nodes, Stålberg uses a node state that represents the state of the adjacent cells (see Figure 18).[5] Depending on these node states, building tiles are set at the corresponding positions.

[2] See http://oskarstalberg.com/game/wave/wave.html (accessed 2020/19/8).
[3] See https://twitter.com/osksta/status/1176569884924416001 (accessed 2020/19/8).
[4] See https://oskarstalberg.com/game/house/index.html (accessed 2020/19/8).
[5] Depending on the concrete implementation, a grid can alternatively keep track of the states of its cells, where a cell state represents the states of its 8 nodes, placing the tiles in the center of a cell. But I will follow Stålberg's approach and use node states.

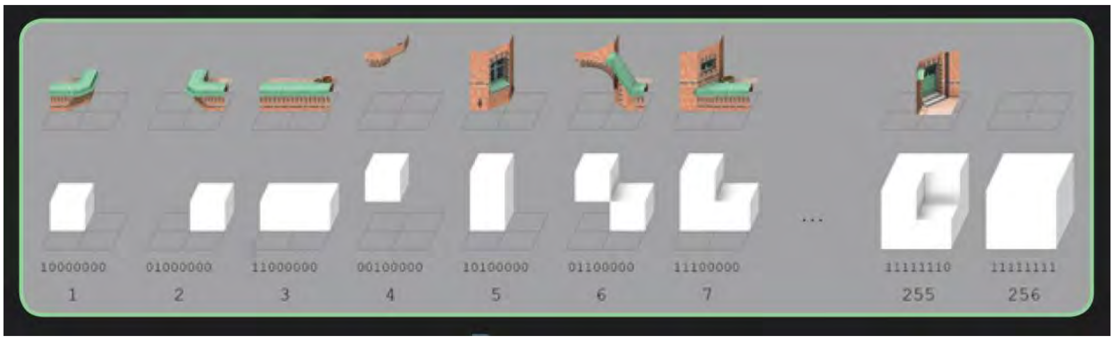


Figure 18: Different 8-bit node states determine the appropriate building tiles [41].

To determine which cells are “inside” (state: 1) and which are “outside” (state: 0), Stålberg lets the users decide about the structure of the generated buildings, by clicking on cells in a 3D grid.
Hence, when users click on an empty cell in the grid, the cell’s state is changed from “outside” to “inside”. Thereby the states of the 8 associated nodes also change and new tiles will be placed at the corresponding positions in the grid for each of these nodes forming a consistent building.

The approach discussed so far is a rigid way to procedurally generate buildings. By only using Marching Cubes, each building that was generated on the same way will always look the same. The combination of MC with WFC produces more varied results. To expand MC with WFC, the previous process basically remains the same. But instead of a single building tile that is always used for a certain node state, one node state corresponds to several building tiles. This is how MC creates the “possibility space” for WFC.
After every click, MC determines the possible building tiles for each node state and WFC then selects the concrete tiles. During the generation of the final buildings the steps *observe* and *propagate* are carried out alternately, like in the original WFC algorithm (see step 3 in chapter 2.3.1). At the beginning, WFC’s wave (the list with the remaining possible patterns for every location) contains all patterns, i.e. all tiles that can be used for the node states. Gradually, possibilities are sorted out with the help of self-defined constraints to finally create the finished image, respectively the finished 3D buildings.

3 Concept

The following chapter presents the logical concept of my application. I will not discuss technological issues or concrete implementations at this point, but rather the abstract idea of a possible implementation.

In chapter 2.5, the possible combination of the WFC and the MC algorithm has been discussed with regard to Stålberg’s implementation of *Townscaper*. These assumptions form the basis of my own concept, which is presented below. I discuss how WFC and MC were used in this thesis to develop an application for building generation.

3.1 Overview

As in Stålberg’s solution, my application is based on a 3D grid consisting of cells. Each cell has a state and is adjacent to 8 nodes. The cells can be in one of two states, “inside” or “outside” (respectively 1 or 0). In addition, the nodes have an 8-bit state that consists of the states of their adjacent cells.

If a node adjoins one or more cells that are “inside”, a 3D building model is assigned to the node. In the UI, this 3D tile is placed at the position of the corresponding node in the grid.

Which 3D tile is to be placed at the position of a node is decided with the help of WFC and MC. Considering the node states, MC determines the course of the buildings’ surfaces. Thus, the algorithm defines a certain “type” of tiles that can be used in a position. WFC regards the “types” of tiles that must be used in the grid and decides which specific tiles are used.

3.2 Modifications to the Marching Cubes Algorithm

As already described in chapter 2.4.1, the MC algorithm basically consists of two main steps: finding the surface and calculating the normals to the surface. Only the first step is required for this application.

3.2.1 Basic Process

The surface of the buildings is determined by assigning a possible *equivalence class* to each node state. An equivalence class comprises at least one, usually several node states and determines the “type” of tile that corresponds to the node states. I.e., node states are equivalent and belong to the same equivalence class, when the same 3D model could be used for them in the final step, see chapter 3.2.5. All node states are assigned to their corresponding equivalence class by MC. Thus, the main task of my modified MC algorithm is to find the equivalence classes for every node state.

The query to find an equivalence class for a specific node state is as follows:

Algorithm 1

```
- For all equivalence classes:
    o Regard all comprised node states:
        ▪ If a comprised node state matches the received node state:
            • Return the equivalence class.
```

3.2.2 Cube Creation at Runtime

The original MC algorithm divides existing data into 3D cubes. In my application, the data is not given in advance, but is defined by the user at runtime.

A UI shows a 3D grid which corresponds to the internal 3D data. Initially, all cells in the grid are empty. A click on a cell in the grid changes the state of the logical cell of the internal 3D data. A previously empty cell is now filled or “inside”. The other way round, a previously filled cell is now empty or “outside”. Eventually, a click into the grid either creates or deletes a visible building (or part of a building) in the user interface.

For an easier differentiation, a filled cell will be called “cube” from now on, while “cells” can be either filled or empty.

3.2.3 The Internal Grid

The internal 3D grid holds the cells and their positions. Boolean values can be stored in the grid for each position that indicate whether the corresponding cell is “inside” or “outside”.

The internal grid maps the state of the UI grid. A click in the visible grid in the user interface triggers the toggling of an entry in the internal grid. A click into the grid either creates or deletes a part of a building and, simultaneously, either sets an internal entry from false to true or vice versa.

Furthermore, the grid determines the positions and states of all nodes. The decision was made, that the position of a cell is determined by its node on the lower left front side. Thus, the cell and node state positions are mapped to one another and the node at the lower left front side of a cell has the same position as the cell in the grid.

3.2.4 Node States Represent Adjacent Cells

A cell has 8 associated nodes each responsible for its 8 corners. The other way round, one node stores the states of its 8 adjacent cells so that it can ensure a smooth transition between neighboring cubes. This means that changing a cell's state results in 8 adjacent nodes changing their state.

The 8-bit index that represents a node state can be determined by coding the node's cells as powers of two. The coding can be seen in Figure 19. Each cell corresponds to an entry in the index, depending on its position in relation to the node. The numbers represent the positions of the corresponding cell states within the 8-bit index of the node state. (The index positions are read from right to left.):
The state of the cell at position 1 corresponds to the index position 1 ($2^1 = 1$), the state of the cell at position 16 corresponds to the index position 4 ($2^4 = 16$). The node state is obtained by adding up the assigned numbers of all cells whose status is "inside". If, e.g., only the cells at position 1 and 16 are "inside", the node state was: 1 + 16 = 17, respectively: 00010001.

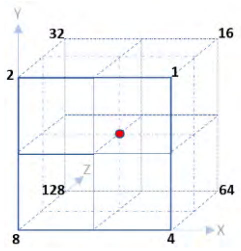


Figure 19: A node and the corners of its 8 adjacent cells. The numbers represent the positions of the corresponding cell states within the 8-bit index of the node state. E.g. if the cells at position 2 and 8 are "inside", the node state would be: 00001010.

The node state is determined as follows:

Algorithm 2

```
- Initialize the node state with 0.
- For every adjacent cell: Check whether the cell is "inside" (the entry in the grid is
  set to true) or "outside" (the entry is set to false).
    o If the cell is "inside", add the coded cube position to the node state.
- Return the node state.
```

Each node state corresponds to a certain "tile"[6], i.e. a 3D model that is placed at the position of the node.[7] If a node is only adjacent to one cube, the corresponding tile of the node fills one corner, i.e. one eighth of the cell. Only if none or all cells that are adjacent to a node are "inside", no tile is used for this node. If all cells of a node are "inside", it is located within a building and does not represent a surface.

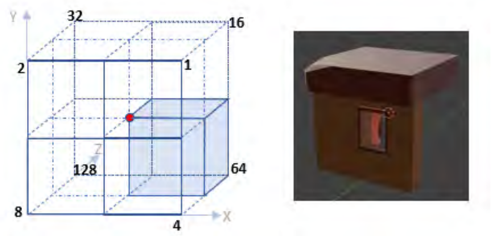


Figure 20: The node state 01000000 corresponds to a 3D tile that fills one eights of a cell. Together with other tiles of neighboring nodes a complete building is created.

As in the original algorithm, there are 2^8 (= 256) possible states. Hence, a table with 256 possibilities can be created that displays which node state results in which 3D tile. But like in original MC, it is not necessary to create 256 different tiles, because of possible rotations. I.e. some node states result in the same tile but with another rotation (those node states that belong to the same equivalence class). Still, there are more than 14 different tiles (as in the original algorithm) to model beforehand, because in our case it is not possible to regard reversed states as equivalent. In addition, a distinction must be made between roof and floor pieces, therefore not all rotations can be regarded as equivalent.

3.2.5 Equivalence Classes

To find out which tiles can be used for the same node states, I grouped them into *equivalence classes*. Node states are equivalent and belong to the same equivalence class, when the same 3D tile can be used for them, eventually.

Originally, reversed states resulted in the same surface, only with reversed normals. Also, all possible rotations of a node state had the same result. Thus, these node states were seen as equivalent. In this application, however, not only a simple surface is created, but 3D models with unique designs are used. Various aspects must be considered to classify node states as equivalent here.

[6] In this thesis, the term "tile" is used to describe 3-dimensional models. These models are previously created and integrated into the program so that the application can use and arrange them at runtime. The term "tile" is a reference to the Wave Function Collapse algorithm that uses tiles for texture synthesis.

[7] In fact, a node state in my application can be represented by different tiles and WFC eventually selects one of them, but only MC is considered at this point.

Mostly, reversed node states cannot be regarded as equivalent. Only the two node states in which all or none of the adjacent cells are “inside” are both reversed and equivalent, as they do not correspond to any tile. They belong to equivalence class 0. Node states that are rotations of each other can be equivalent but do not necessarily have to be.

A utility program[8] (see chapter 3.2.6) that was additionally developed in the context of this thesis has shown that there are 23 different equivalence classes if all possible rotations are allowed and reversed states are not considered to be the same. If you combine the two states that do not represent a surface in one class (equivalence class 0), you get 22 basic equivalence classes. However, a lot more states must be distinguished. The basic equivalence classes can be seen in the appendix, see 8.2.

3.2.6 Note on the Utility Program

The additional utility program is based on a tool that professor Jenke made available for this thesis. With his tool one could select a node state and display all its possible reflections and rotations.

Since node states that are in the same equivalence class can be mapped onto the same 3D tile, only node states that are rotations of one another, but not those that are reflections of one another, belong to the same class. That is because the tiles can be rotated and positioned by the algorithm, but not mirrored.

I extended the program to find all the equivalence classes under the assumption that node states are equivalent when they are rotations of one other. In this way 23 equivalence classes were found. As already mentioned, the two reversed node states that do not represent a surface are equivalent but were recognized by the algorithm as two different equivalence classes. These can be combined so that 22 basic classes remain.
In addition, these 22 classes need to be further subdivided, since e.g. tiles that represent a ceiling must look different than tiles that represent the floor. Due to the distinction between roof tiles and floor tiles, node states can only be equivalent if one state can be converted into the other state by rotating around the Y-axis (see Figure 21).

[8] Project wfc_ba: https://git.haw-hamburg.de/ace506/wfc_ba

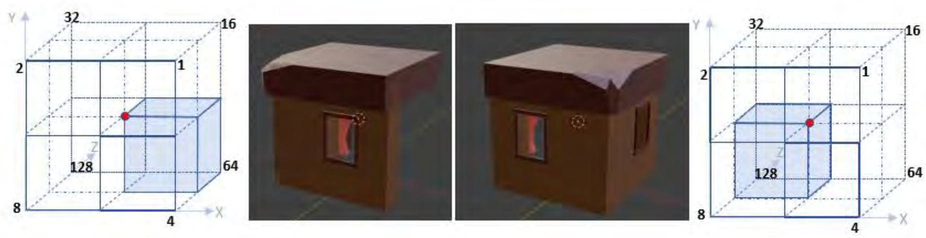


Figure 21: These node states (01000000 and 10000000) are equivalent and belong to the same equivalence class. Thus, the same 3D tile can be used for them with different rotations.

To find out the sub-equivalence classes, the 22 basic classes were subdivided by examining which of the node states can be converted into one another by rotations around the Y-axis. Therefore, I have further extended the program to find the node states within the 22 classes that are Y-rotations of one another.
The final (sub-)equivalence classes have a basic state and the other node states from the same class are rotations of this basic state. In the actual application of this thesis, a 3D tile that belongs to an equivalence class will be rotated by default in such a way that it corresponds to this basic state. If the tile is used for another node state of the same equivalence class, it must be further rotated. The utility program calculated these required rotations and wrote them into a table, which gets imported once when the actual application starts.

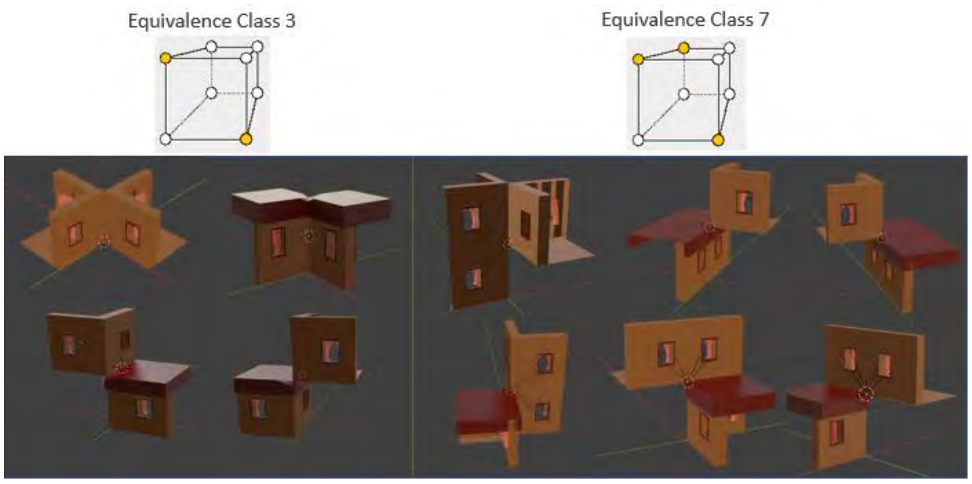


Figure 22: To give an example: The basic equivalence classes 3 and 7 comprise four respectively six sub-equivalence classes that are shown here as 3D tiles. Each sub-class comprises four different node states – the possible rotations around the Y-axis.

3.3 Modifications to the Wave Function Collapse Algorithm

As described in chapter 2.3.1, the WFC algorithm can be divided into 4 steps:

1. Extracting local patterns from an input image
2. Processing these patterns in an index, so that constraint checking can be accelerated
3. Incrementally creating an output image by expanding a partial assignment
4. Rendering the total assignment into an image

The first and second steps are not relevant for the application described here since no input image is used. However, they are replaced by the Marching Cubes algorithm. Instead of extracting local patterns from an input image, the patterns are given in advance (in the form of 3D tiles). Instead of an index, node states define the positions and rotations of the tiles. Also, constraints are not extracted from a given example but self-defined.

My adapted WFC algorithm works on top of the results of MC. Only the equivalence classes and their positions in the grid are relevant for the application of the algorithm since all associated node states correspond to the same tiles. WFC examines all possible tiles for an equivalence class and uses the defined constraints to decide how the respective location must be collapsed, i.e. which pattern (concrete tile)[9] should be used for a certain location.

For my application, the adaption of step 3, i.e. the incremental creation of an output image, is most important and will be described in the following.

3.3.1 The Wave Stores Decision Variables

The generation process remains incremental as in Gumin's original algorithm. After the WFC algorithm is triggered, all relevant locations receive decision variables that are sorted out step by step. However, in this application not the entire grid has to be collapsed, but only those positions at which a node is adjacent to one to seven cubes (i.e. those positions where the corresponding node states do not belong to equivalence class 0).

Thus, in the wave, the pattern options are saved for each node location of the output image that adjoins a visible cube. These decision variables, i.e. lists of pattern options, are to be collapsed. Although the lists of patterns have the same structure for all node locations, it is possible to offer different options depending on the equivalence class. The entries in the wave for some options can be set to false at the beginning. Pattern options in the wave include different colors for the tiles as well as the possibility to set doors or balconies at certain positions.

[9] The options for the different tiles that WFC stores in the wave are called "patterns" in the following to distinguish them from the concrete 3D tiles.

For each position, the wave saves a list of possibilities, which are Boolean values as in the original algorithm and initialized as “true”. Those locations that shall be disregarded by the algorithm (locations without cubes) are directly marked as collapsed from the beginning and receive a 0 entropy by setting all but one dedicated pattern to “false”.
With each pass of the algorithm, possibilities in the wave are sorted out and one position is collapsed, i.e. its value is finally determined and assigned. Eventually all locations will have been collapsed so that exactly one possibility is assigned to each location.

As in Gumin's original algorithm, my implementation will not use backtracking. If a contradiction occurs, WFC is executed from the beginning. This decision makes sense for my implementation because contradictions almost never occur since the constraints are not affected by each other.

The third step of Gumin’s WFC can be divided into the sub-phases *observe* and *propagate*, this also applies to my implementation. These two steps are executed sequentially over and over again until WFC finds a total assignment (or a contradiction occurs).

3.3.2 Observe

The purpose of the *observe* phase remains to find the location with the lowest entropy (that is greater than 0). So, the location with the fewest possible patterns is selected. However, the condition remains the same that only a location with more than one option can be selected. If a location has only one possibility left, it would have already been collapsed and the entropy was 0. If no more options were left, there was a contradiction and the algorithm had to restart.

Basic Process

The *observe* phase starts with finding the lowest entropy position in the wave. Based on this, the further process is decided.

Algorithm 3

```
- Find the position with the lowest entropy in the wave.
- If the lowest entropy could not be found and all node locations are at entropy 0:
    o The processing is completed, and all locations have already been
      collapsed. WFC is finished and the collapsed observations are returned.
- If at least one location has no more valid options and its entropy is below 0:
    o The algorithm ran into a contradiction and must restart.
- Otherwise:
    o The location with the lowest entropy was found. A pattern is selected and
      the entropy for this position is set to 0.
```

Finding the Lowest Entropy

The entropy of every node in the grid is checked by regarding its entries in the wave. The entropy value is determined by the remaining patterns that are stored in the wave. If one option is left, the entropy is 0; if two options are left, the entropy is 1; etc.

Algorithm 4

```
- A 3-dimensional vector is initialized with (-1, -1, -1).
- For every node in the wave:
    o If a node's entropy is greater than 0 and less than previously found
      entropies: Its position is stored in the vector.
    o If a location's entropy is below 0: The algorithm found a contradiction. The
      vector receives the value (-2, -2, -2) and is returned.
- If all entropies are 0: The entire grid has been collapsed and the unchanged vector
  is returned.
- Otherwise: The vector with the found position is returned.
```

When all positions are run through and a valid node with the lowest entropy is found, this location is used for the further steps of the algorithm.

Select a Pattern and Set Entropy to Zero

After the location with the lowest greater-than-zero entropy is selected, this location is collapsed. This means, one of the remaining patterns (i.e. options for the concrete 3D tiles) is selected and the entropy is set to 0.

Either all possibilities are still open for this location or some possibilities have already been ruled out. In either case, one of the remaining patterns is selected randomly with predetermined weights. The original WFC uses the frequency with which the patterns occur in the input image as weights. Since this is not possible in my case, the weights are set manually. The weights for the colors can be changed by the users.

The selection of a pattern works as follows:

Algorithm 5

```
- The weights of all remaining patterns of the chosen location are added.
- A random number between 0 and the sum of weights is created.
- For all patterns of the chosen location:
    o If the random number is less than the weight of the current pattern plus
      the summed weights of previously checked patterns:
        ▪ This pattern is selected, and the query is ended.
- After a pattern was selected, all other possibilities are ruled out.
```

This process ensures that only available patterns are selected and, in addition, that the selection is based on the pattern's weights. This can be seen from some examples:

- If the first pattern cannot be selected, its weight was 0. Therefore, the random number cannot be less than this weight and the first pattern will not be chosen.
- If the first pattern can be selected, but the second cannot and the random number is greater than the first pattern, than the random number will also be greater than the added weights of the first and the second pattern, so the second pattern will not be chosen.
- If the first and second pattern can be selected and the random number is bigger than the weight of the first pattern, but smaller than the added weights of the first and second patterns, the second pattern is chosen.

By adding the weights and choosing a random number between 0 and this sum, the weights determine the probability with which the associated patterns are selected. Because the larger a weight, the larger the range of numbers in which the random number can be, and the larger the probability for this case.

After a pattern was chosen and the location is collapsed, it and other variables in the wave must be updated by constraint propagation in the next step. The collapsed location is transferred to the subsequent *propagate* step as starting point for the constraint propagation. Whenever a location is collapsed, propagation is carried out afterwards. If a contradiction has occurred or the entire grid is already collapsed, this is not necessary, because either the algorithm must restart, or it is finished.

3.3.3 Propagate

My propagation phase works very similar to the original algorithm. "Arc consistency" ensures that a value only appears in the domain of a variable if there is a valid value in the domain of neighboring variables so that all constraints can be fulfilled. The propagation works iteratively and only stops when no more locations need to be updated.

Starting from the location that was chosen during the previous *observe* phase, every adjacent node location will be checked whether it must update its domain. If an adjacent node needs to be updated, it receives a flag that it needs to check its own neighbors for necessary updates in the next iteration step, etc.

There is a list of all nodes that need to update their neighbors. At the beginning the list contains exactly one node - the previously chosen node of the *observe* step.
The list functions as a queue: new nodes that need to update their neighbors are appended at the end, and the node used for the next update is always taken from the front.

Whenever a queue is mentioned in the following within this chapter, it is a reference to that list.

Basic Process

Algorithm 6

- While the queue is not empty, the next steps will be repeated in a loop:
    - One node (the updating node) is taken from the queue.
    - A list with the node's relevant direct neighbors is created (nodes that correspond to equivalence class 0 are irrelevant).
    - Subsequently, every neighbor is checked whether its domain must be adapted to the domain of the updating node based on the chosen constraints.
        - If the updating node has already excluded certain possibilities that the neighbor still includes:
            - The neighbor is adapted to the node and added to the queue with nodes that need to update their neighbors.

During the propagation process, the possibilities of the neighbors are adjusted to the respective updating node. The updating node cannot be changed. Furthermore, pattern options can only be excluded and not added.

Finding the Direct Neighbors of the Updating Node

The algorithm to find the direct neighbors of an updating node is the only time when WFC also needs the node state of a location. All other parts of the algorithm work exclusively on top of the equivalence classes. But only the node state holds the information about where a node adjoins a cube and therefore in what direction neighboring nodes can be found.

It must be pointed out that a node that is directly next to the updating node and does not belong to equivalence class 0 is not necessarily a direct neighbor. It is only a direct neighbor if it and the updating node also share a common cube. Otherwise these two nodes are not connected, see Figure 23.

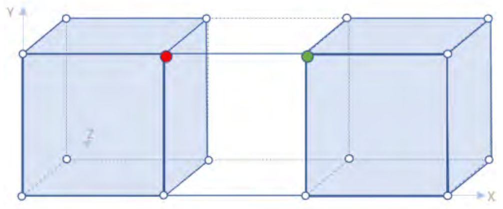


Figure 23: The red and green nodes are neighbors and do not correspond to the equivalence class 0. However, the green node is not a direct neighbor of the red node as they do not share a common cube.

For this reason, first, a list is created with the positions of the cubes that belong to the updating node. The positions of the cubes are determined from the node position and the node state.

To be able to compare the cubes of different nodes with one another, it is necessary to find a uniform conversion which determines the locations of the associated cubes. I decided to define the position of a cube by its lower left front corner, see Figure 24.

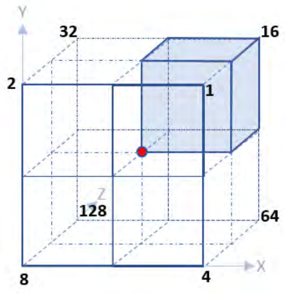


Figure 24: The lower left front corner of a cube belongs to this node. Since the lower left front corner of a cube defines the cube's position, the position of the cube matches the position of the node.

After the list with cubes adjoining the updating node was created, all neighboring nodes are checked whether they are direct neighbors of the updating node.

Algorithm 7

```
- For all three axes in all possible combinations, regard the updating node's
  neighboring nodes (for each axis, the search starts at: updating node's position - 1
  and ends at: updating node's position + 1).
    o If the equivalence class of the node at the corresponding position is not 0:
        ▪ A list is created with the positions of the cubes belonging to the
          possible neighboring node.
        ▪ If this list contains a cube that is also contained in the list of cubes
          belonging to the updating node:
            • A direct neighbor is found and added to the list of direct
              neighbors.
```

Apply Constraints to the Neighbors of the Updating Node

The method that is responsible for the application of the constraints is also the place where the constraints are implemented. The original WFC algorithm derives the constraints from the input image and stores them in an index, but this is not possible in my case, since there is no input image.

The entries in the wave are structured in the same way for all grid locations. Regardless of the node state or the specific location, the structure and length of the list with the different pattern options remain the same. This makes it possible to apply the same constraints uniformly to all nodes. Apart from that, it is possible to exclude options for nodes of certain equivalence classes from the beginning, by setting their entries in the wave to false.

To apply the constraints, the open pattern options of the updating node are successively compared to the options of its direct neighbors. This means, the entries in the wave are compared with each other for each constraint. The updating node specifies the desired state, and the neighbors are adapted to it if necessary.

The first constraint is applied to a neighbor as follows:

Algorithm 8

```
- Iterate over the updating node's list of patterns:
    o If the updating node has already ruled out the options for one color and
      the neighbor still allows it:
        ▪ Adapt the entries of the neighbor to the entries of the updating
          node.
        ▪ Add the neighbor to the queue of nodes that need to update their
          neighbors.
    o Otherwise do nothing.
```

The second constraint says that a building can only have doors on the ground and that doors cannot be directly next to each other. Only some equivalence classes allow a door to be used. For all other nodes, the option of a door is set to false when the wave is initialized. Besides, the position of the node can be used to determine whether it is on the ground.

The process for the application of the second constraint is as follows:

Algorithm 9

```
- If the neighbor's pattern options in the wave still include doors:
    o If the neighbor's position is not on the ground or the updating node still
      allows doors (or both):
        ▪ Set the door options of the neighbor to false.
```

On the one hand, this request ensures that nodes are only allowed to have the option for a door if they are on the floor. On the other hand, it ensures that a node can only have the option for a door if its neighbors do not.

After several rounds of the *observe* and *propagate* phases, the algorithm assigned a final pattern to every node location and they are returned.

3.3.4 Optimization of WFC

The application of WFC normally requires that parts of the grid or even the entire grid is scanned several times in the *observe* and the *propagate* phase. To improve the performance of my WFC, the grid area that is relevant for the algorithm has been limited, so that it is often not necessary to scan the whole grid.

Every time, WFC is called, it checks whether the grid has already been collapsed in a previous step. If so, then only the associated parts of the grid to which a new part was added are recalculated (i.e. the building to which a new part was added). Existing buildings that are not changed remain unchanged and the same result is returned for them as in the previous calculation.

The optimization only affects the initialization phase. During the initialization of the wave, the node locations that were not affected by the change receive the pattern that was selected for them in the previous WFC run. Only the affected nodes receive different pattern options.

Two possible paths are distinguished during the initialization of WFC:

Algorithm 10

```
- If WFC has already been executed before:
    o The affected nodes are determined.
- If WFC is executed for the first time:
    o All nodes that do not correspond to the equivalence class 0 are affected
      nodes.
- The wave is initialized with the affected nodes.
```

If WFC has been executed before, the affected nodes are determined as follows:

Algorithm 11

```
- An empty list for the affected nodes is created.
- For every location of the grid: The equivalence class from the last run of WFC is
  compared to the new equivalence class.
    o If the equivalence classes do not match at a certain location: The
      corresponding node is added to the list of affected nodes.
- Until all neighbors of all affected nodes are found:
    o Find the direct neighbors (see Algorithm 7) of every node in the list of
      affected nodes.
    o If a direct neighbor is not yet included, add it to the list.
```

The wave is initialized as follows:

Algorithm 12

```
- If WFC is executed for the first time:
    o Create an empty list for the wave.
- For every node in the grid:
    o Create an entry in the wave.
    o If the node is contained in the list of affected nodes:
        ▪ Set all possible patterns in the wave to true.
    o Else:
        ▪ Set the entropy for this node in the wave to 0. (I.e. set one
          dedicated pattern in the wave to true, all others to false.)
```

3.4 Combining Wave Function Collapse and Marching Cubes

To create buildings with a single click and make them also appear in different variations, it is necessary to combine the two algorithms WFC and MC.

3.4.1 Basic Process

When a user clicks into the grid, a cube is either created or deleted, i.e. building parts are either inserted or removed at the corresponding position. An internal 3D grid stores the states of the cells and can thus also calculate the node states.

A click into the visible grid changes the state of the internal grid and triggers the building generation. With every click:

Algorithm 13

```
- All visible 3D tiles are removed from the user interface.
- For every cell in the internal 3D grid:
    o Obtain the node state from the cube's lower left front side.
    o Obtain the equivalence class for the node state.
- After all node states and associated equivalence classes were obtained and saved
  in their own 3-dimensional list, WFC is applied to them.
- Finally, the node states and the patterns that were assigned by WFC are used to
  place the 3D tiles in the visible grid of the user interface.
```

3.4.2 Integration of MC and WFC

As can be seen in the basic process, chapter 3.4.1, the MC algorithm is carried out during the same loop as querying the node states, while WFC is executed afterwards.

MC can be executed before all node states are known because it only needs one node state at a time for the determination of its equivalence class.
To apply WFC, all equivalence classes must already be known. Not only the states of individual locations are of interest, but also the state of (almost) the entire grid. Both the observation and the propagation phase must be able to scan neighboring nodes. Thanks to the built-in optimization, however, the entire grid does not always have to be considered.

3.4.3 Placing the Tiles

After MC has selected the equivalence classes and WFC the final patterns for every node location, the 3D tiles need to be placed in the visible grid in the user interface.

Algorithm 14

```
- For every node:
    o If the node state does not correspond to the equivalence class 0:
        ▪ A game object is created with the 3D tile that corresponds to the
          pattern chosen by WFC.
        ▪ The game object is rotated depending on its node state.
        ▪ The game object is positioned in the grid in the game.
```

Rotation of the Game Object

With the utility program that also defined the equivalence classes, the necessary rotations were determined for all node states, starting from a basic position. The utility program generated a list with entries for every node state, defining how each node state is to be rotated. Node states that hold the basic position for their equivalence class do not need to be rotated. Thus, a game object is rotated depending on its node state.

Positioning of the Game Object

The node state is also important for the positioning of a game object. The internal grid already stores the positions of all its nodes. But for the correct positioning, it is also important to consider in which directions the 3D tile must extend. If, e.g., a node state only has cubes in the upper half, the 3D tile must be moved a quarter step upwards (one step corresponds to the length of a whole cube). If a node state has cubes both in the upper and lower half, the object's center on the Y-axis would be exactly where the node position is and would not have to be adjusted. The same principle applies to the other axes.

4 Implementation

This chapter gives an overview about the implementation of my application. First, important software engineering aspects of the project are discussed. Subsequently, the use of concrete development environments, libraries, and assets are examined.

4.1 Analysis

For the analysis of the software project, user stories and a user story map were created, stakeholders and use cases were set up.

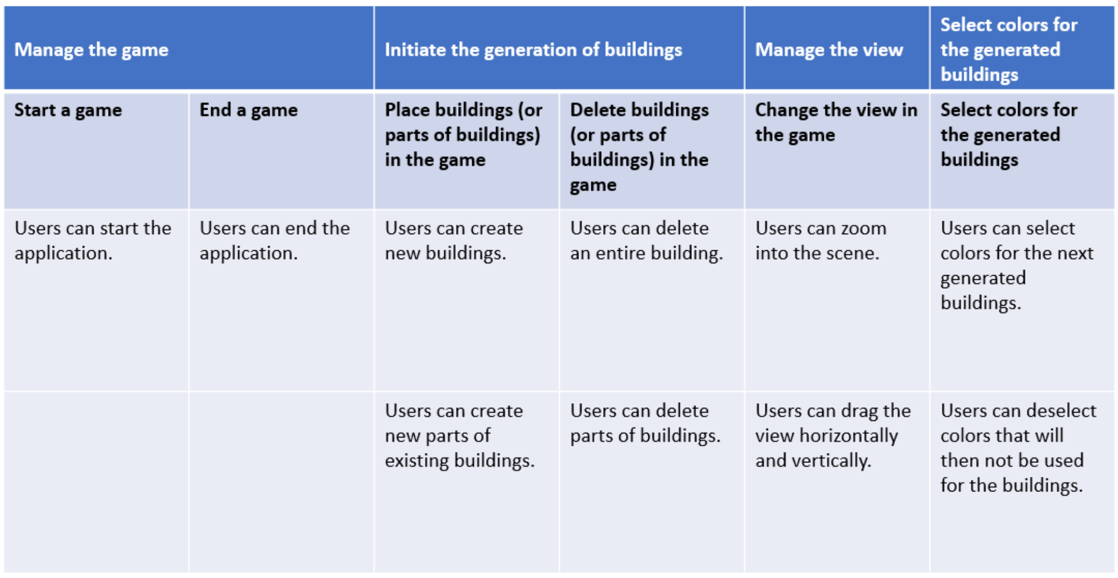
This chapter includes the requirements analysis as well as the specification. The specification is mainly intended to be used for an agreement with the customer, but also to create a basis for a later review of the system's correctness [42]. Hence, quality features and metrics are established for quality assurance, which will then be evaluated in chapter 0.

4.1.1 Stakeholders

Stakeholders can be divided into internal and external customers and users. The information about stakeholder is based on [43].
Usually the external stakeholders are the actual customers who give the order for the project. They provide the requirements and technical specifications. External customers only exist when the product is externally sold (i.e. outside of a company or organization), therefore, they do not have to be considered in my case.
Internal customers are usually product owners, product managers or team leads, i.e. contact persons for technical specifications, which are in your own company. In my case, the supervisors of this thesis can be seen as internal customers with whom I discuss the approach and implementation of the project.
Users are the ones that finally use the product and whose skills and needs must be considered during development. In truth, my application is used almost exclusively by me and the supervisors of the thesis (tech-experienced people). Nonetheless, since my project is inspired by Stålberg's *Townscaper*, I wanted to assume a similar target group as for his game. *Townscaper* should appeal to a wide audience, presumably people of almost all ages that are interested in computer games. The application must therefore have as few hurdles as possible so that children and the elderly can also use it intuitively.

4.1.2 User Stories and User Story Map

User stories depict the benefits and the interaction of the stakeholders with the system. Since, the different interactions with the system were kept little, the number of user stories is also limited and can be summarized in a user story map, see Figure 25.



Manage the game		**Initiate the generation of buildings**		**Manage the view**	**Select colors for the generated buildings**
Start a game	**End a game**	**Place buildings (or parts of buildings) in the game**	**Delete buildings (or parts of buildings) in the game**	**Change the view in the game**	**Select colors for the generated buildings**
Users can start the application.	Users can end the application.	Users can create new buildings.	Users can delete an entire building.	Users can zoom into the scene.	Users can select colors for the next generated buildings.
		Users can create new parts of existing buildings.	Users can delete parts of buildings.	Users can drag the view horizontally and vertically.	Users can deselect colors that will then not be used for the buildings.

Figure 25: A simple user story map of the application.

The user stories in a map are in an abbreviated form. The first line shows the user activities, the second line the user tasks. The associated user stories are located under the user tasks [44] [45].

This map shows how the system looks from the user's perspective and distinguishes whether a single building is generated (or removed), or only a part of an already existing building. However, this makes no difference internally. Either way, only the state of one cell and its adjacent nodes change and the generation process of MC and WFC is restarted for all neighboring cubes (see chapter 0).

4.1.3 Use Cases

Use cases describe the behavior of a system in response to actions by a user from the outside perspective. They are derived from the system requirements, e.g. from the user stories. Only the crucial user stories are worked out as use cases to clarify certain processes and workflows. [44] [46]
To clearly differentiate between the creation of a single building and a part of the building, I created a use case for these two cases. As already mentioned, this makes no difference

internally, but it appears different when you look at the system from the outside – which is what you do with use cases. The template for the use cases can be found here: [47].

Use Case 01: User creates a new building

Use case name:	UC01: User creates a new building
Use case overview:	A user left-clicks on an empty cell in the 3D grid whose adjacent cells are all empty. A new building is generated.
Actors:	User(U); System(S)
Preconditions:	The application must have been started.
Termination outcome:	A new building on the grid is created.
Input summary:	Left-click by the user
Output summary:	New grid state, new building in the UI

Standard flow:
U: left-clicks on an empty cell in the grid with empty adjacent cells.
S: shows a new building (that can – e.g. – differ in color from other buildings).
Use case notes:
The new building is not affected by constraints from other buildings.

Use Case 02: User creates a new part of an existing building

Use case name:	UC02: User creates a new part of an existing building
Use case overview:	A user left-clicks on an empty cell in the 3D grid whose adjacent cells are not all empty. An already existing building is being expanded or already existing buildings are being connected.
Actors:	User(U); System(S)
Preconditions:	The application must have been started, at least one building must have been generated.
Termination outcome:	A new part of an existing building is created, or two or more buildings are connected.
Input summary:	Left-click by the user
Output summary:	New grid state, expanded building(s) in the UI

Standard flow:
U: left-clicks on an empty cell in the grid that adjoins a building.
S: shows a new part of a building, its appearance is influenced by the exterior of the building to which it belongs.
Alternative flow:
U: clicks on an empty cell on the grid that adjoins two or more buildings
S: shows a new building part that now connects two or more buildings. The appearance of the buildings (that are now one building) adapts to each other.
Use case notes:
Constraints are applied to all parts of the building.

4.1.4 Overview of User Interactions

Since the interactions of the user with the system are limited and not complicated, it is not necessary to write a separate use case for each. Instead, I decided to create a brief overview of the possible interactions.

User Action	System Reaction
- Left-click in the grid	- Create a (part of a) building
- Right-click in the grid	- Delete a (part of a) building
- Scroll with the middle mouse button	- Zoom in the game scene vertically
- Drag the middle mouse button	- Drag the game view horizontally
- Press ESC button	- End the game
- Press “X” button in UI	- End the game
- Deselect a color button in UI	- Corresponding color is no longer used for generation
- Select a color button in UI	- Corresponding color is once again used for generation

4.1.5 Quality Assurance

So far, the functional requirements for the project have been posed. The following section on quality assurance describes the non-functional requirements.

First, generally applicable quality factors are briefly regarded. Afterwards, my own requirements are placed on the quality of the software.

General Quality Factors

Different generally applicable quality factors are defined in different sources. For example, the IEEE defines the following quality factors: correctness, reliability, integrity, usability, maintainability, efficiency, flexibility, testability, portability, reusability, and interoperability [44]. Within a project, it must be decided individually which quality features should receive special attention.

In my project, particular attention has been paid to the following general quality factors:
- maintainability
- extensibility
- usability

Methods to achieve the chosen quality factors are (see [46]):

- the use of the principles:
    - modular design, low coupling, high cohesion, separation of concerns (SoC), single-responsibility, interface segregation
- unit testing
- usability testing

Even if the focus is only on three quality factors, the others can still be achieved, as the used methods also address other factors like reusability, correctness, and reliability.

The implementation of these methods can be seen in chapter 4.2 on software design on the basis of component-, class-, and further diagrams. The evaluation of these methods with regard to the achievement of the quality features can be found in chapter 0.

The following metrics can be used to check the quality requirements [48]:

- Velocity
- Failure and repair time
- Code churn (necessary code modifications)

Further Quality Requirements

In addition, I set up own requirements for the quality of the software.

The following 6 properties should apply to the generated buildings:

- Buildings can be placed on a designated area in the game that is easy to identify.
- Connected buildings should have a consistent appearance.
    - A consistent exterior means that all parts of the building connect to one another and there are, e.g., no open house walls.
- Connected buildings should have the same color.
- Buildings can only have doors on the floor.
- Two or more doors cannot be directly next to each other.
- Building parts should be able to be connected to each other over all possible corners, edges, and surfaces.

Since I wanted to assume a similar target group as for *Townscaper*, a quality requirement for my project is that it should be playable for people of all ages. The game cannot present high hurdles so that even children and the elderly can use it. To check this requirement, usability tests can be carried out.

Another requirement that is partly related to the previous requirement is the intuitive usability of the software. However, intuitive usability is not only interesting for younger and elderly users, but it is also a benefit for all other users. This requirement can also be examined using usability testing.

To implement the two requirements for usability, I decided to limit the possibilities for users to interact with the system. In *Townscaper*, Stålberg seems to use a similar strategy. Basically, users can only set and delete buildings, and choose the colors for the generation.

Compliance with the quality requirements is checked in chapter 0.

4.2 Software Design

4.2.1 Component Diagram

To give an initial overview of the software design, Figure 26 shows the component diagram for my project. In this case, the components correspond exactly to the two layers of my layered architecture: UI and logic. Smaller component boundaries did not appear reasonable in the context of this project, since the contained classes already represent meaningful boundaries to one another and cannot be grouped any further.

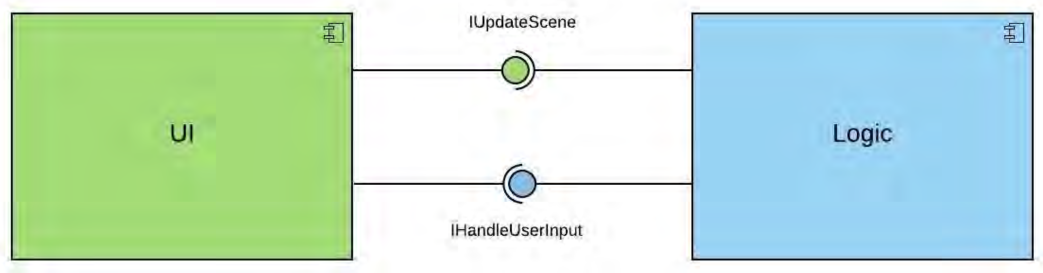


Figure 26: The component diagram shows the two layers of the project, UI and Logic.[10]

The interaction between the two components is clearly limited to the two interfaces. The logic component uses the interface *IUpdateScene* to trigger the redrawing of a scene in the UI component, nevertheless, the logic component must know, when there was a user input and gets informed via its interface *IHandleUserInput*.

The division into components enables compliance with SoC because each component is responsible for a clearly defined area of responsibility. The narrow interfaces, which are precisely tailored to the components, implement the interface segregation principle.

[10] The diagrams were created using a free license of *Lucidchart*: https://www.lucidchart.com/.

4.2.2 Class Diagram

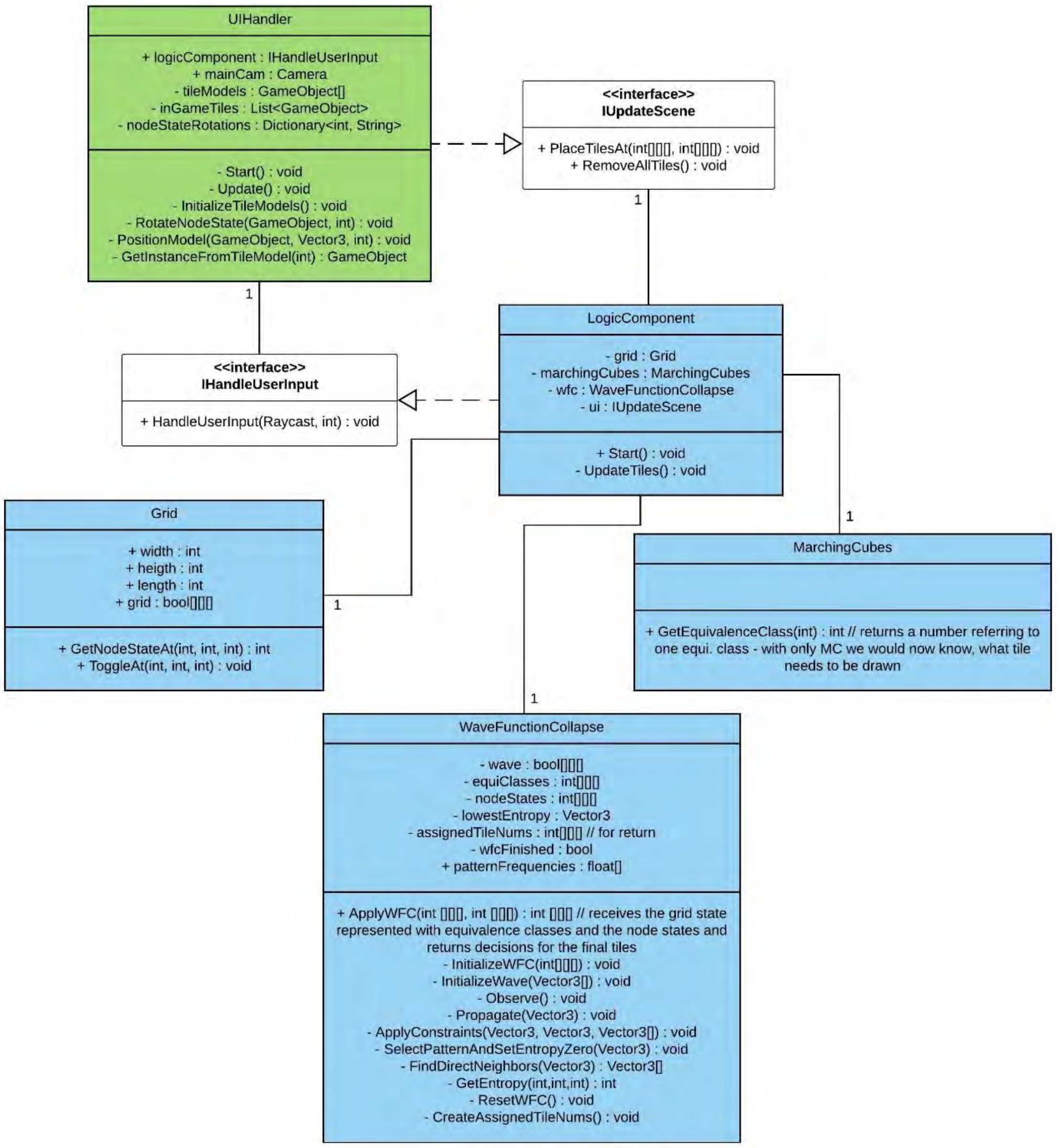

UIHandler
+ logicComponent : IHandleUserInput
+ mainCam : Camera
- tileModels : GameObject[]
- inGameTiles : List<GameObject>
- nodeStateRotations : Dictionary<int, String>
- Start() : void
- Update() : void
- InitializeTileModels() : void
- RotateNodeState(GameObject, int) : void
- PositionModel(GameObject, Vector3, int) : void
- GetInstanceFromTileModel(int) : GameObject
<<interface>>
IUpdateScene
+ PlaceTilesAt(int[][][], int[][][]) : void
+ RemoveAllTiles() : void
1
1
LogicComponent
- grid : Grid
- marchingCubes : MarchingCubes
- wfc : WaveFunctionCollapse
- ui : IUpdateScene
+ Start() : void
- UpdateTiles() : void
<<interface>>
IHandleUserInput
+ HandleUserInput(Raycast, int) : void
Grid
+ width : int
+ heigth : int
+ length : int
+ grid : bool[][][]
+ GetNodeStateAt(int, int, int) : int
+ ToggleAt(int, int, int) : void
1
1
MarchingCubes
+ GetEquivalenceClass(int) : int // returns a number referring to one equi. class - with only MC we would now know, what tile needs to be drawn
1
WaveFunctionCollapse
- wave : bool[][][]
- equiClasses : int[][][]
- nodeStates : int[][][]
- lowestEntropy : Vector3
- assignedTileNums : int[][][] // for return
- wfcFinished : bool
+ patternFrequencies : float[]
+ ApplyWFC(int [][][], int [][][]) : int [][][] // receives the grid state represented with equivalence classes and the node states and returns decisions for the final tiles
- InitializeWFC(int[][][]) : void
- InitializeWave(Vector3[]) : void
- Observe() : void
- Propagate(Vector3) : void
- ApplyConstraints(Vector3, Vector3, Vector3[]) : void
- SelectPatternAndSetEntropyZero(Vector3) : void
- FindDirectNeighbors(Vector3) : Vector3[]
- GetEntropy(int,int,int) : int
- ResetWFC() : void
- CreateAssignedTileNums() : void

Figure 27: The colors in this class diagram mark the different components.

The classes were created according to the single responsibility principle. A class is responsible for a clearly defined task or a clearly defined area. As a result, a modular design with high cohesion is implemented.
In addition, an attempt was made to implement low coupling of the classes by using a component class in the logic component. The *LogicComponent* class is responsible for the

coordination of the different classes within the component; it knows all the other classes and therefore the other classes do not have to know each other (see: 4.2.3 MVC Pattern). The class diagram also shows that the two components only communicate with each other via their interfaces, another means for low coupling.

Scripts outside the class diagram

Apart from the classes from the diagram, there are three additional scripts. There is one utility class and two scripts which control the behavior of game objects and which do not really act in the concept of a class diagram.

Within the utility class, the equivalence classes and the cube states were coded uniformly (with Enums) so that this uniform coding could be used throughout the project.

The two other scripts were necessary because of the way in which Unity (the game engine I used, see 4.3.2) works. In Unity, game objects in a scene can receive associated scripts that control the behavior of these objects.
One script was used for controlling the camera movement. It listens to input from the middle mouse button and lets the user drag and zoom into the view.
Another script was necessary to control the buttons that are responsible for the color selection of the buildings. This script monitors the buttons and reacts when they are clicked. It must also access the WFC class because the buttons define the colors that WFC can select. In doing so, the button script violates the component limits. This “unclean” implementation is caused by the way implementation works in Unity. Nevertheless, it would have been possible to maintain the component limits through a few detours. The button script could be linked with the UI component class. In this case, however, the interfaces and the component classes would also have had to be adapted. Another possibility would have been to use the observer pattern. The WFC class would observe the button script. However, in this case, the observer must be registered with the subject, which would have required changes in several places. For this reason, the decision was made in favor of the lighter and less clean solution at this point. In return, the code has remained clearer and less cumbersome.

4.2.3 Patterns

Model View Controller Pattern

A classic for the implementation of games and other software is the MVC pattern that was also used in my application.

This pattern says that an application is divided into three parts: Model, View and Controller. The model contains the data (for example classic objects of object-oriented programming), the view is responsible for the visual representation and the controller controls the communication between the two other parts and also the access of the user to the program [49].

In my application, the model-part is in the logic component, these are namely the classes: *MarchingCubes*, *WaveFunctionCollapse* and *Grid*.
The controller is responsible for the data flow, it updates the models and the view. Therefore, the *LogicComponent* clearly implements this function.
The view is responsible for the visualization and is implemented with the *UIHandler* class.

Utility Abstraction Pattern

To avoid redundant implementations in different parts of an application that do not necessarily belong together the Utility Abstraction pattern creates an additional utility layer that can be globally accessed [50].

In my application this pattern was implemented with the utility class *Util* that contains the global coding of the node states and the equivalence classes.

4.2.4 Sequence Diagrams

Overview of the Application

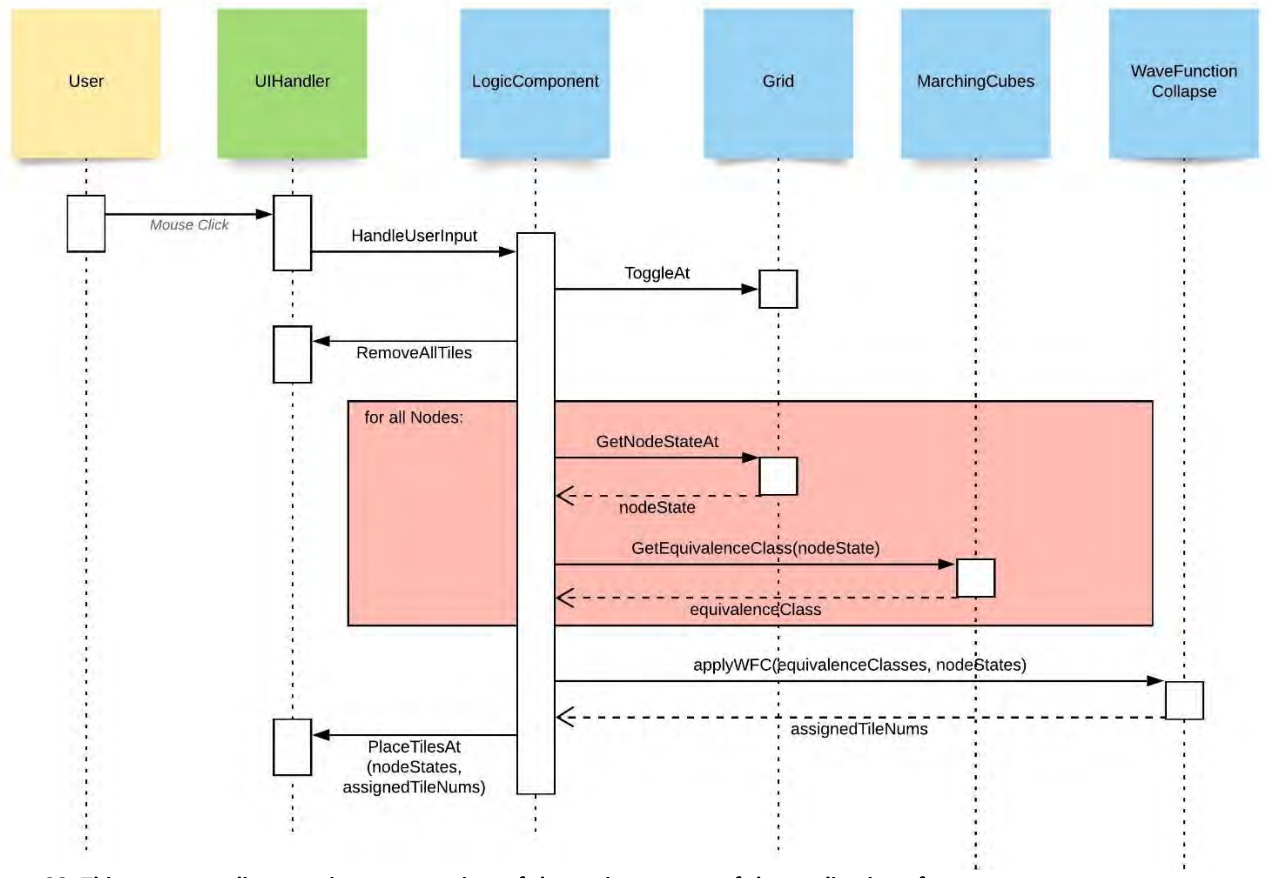


Figure 28: This sequence diagram gives an overview of the entire process of the application after one mouse click into the visual grid.

This sequence diagram gives an overview of how the different classes interact with each other. A valid click into the 3D grid starts the process. The *LogicComponent* receives the user input (it knows which key was clicked in which field of the grid) and manages the entire process. The click can be done with either the right or left mouse button, i.e. either a building (part) can be added or removed. The same process is started in both cases.

The Implementation of Wave Function Collapse

The WFC algorithm represents a fundamental function in my application. After all, this thesis was intended to examine the possible connection between the MC and WFC algorithms. After considering the adapted concept of MC for my application and working out the equivalence classes (see chapter 3.2), the actual implementation of MC is relatively simple. Basically, the associated equivalence class only needs to be found for a specific node state. The implementation of WFC, on the other hand, is a lot more complex.

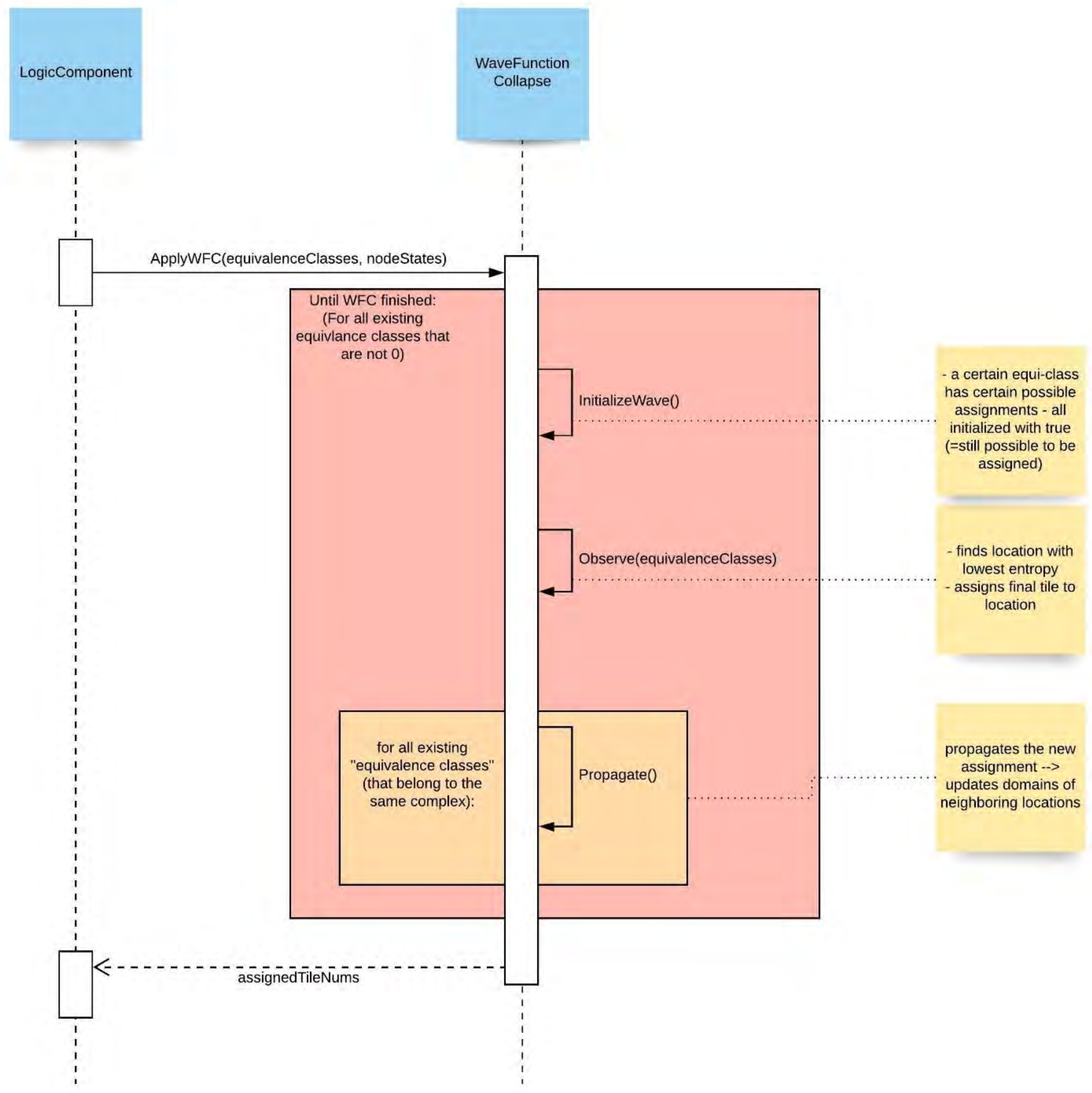

LogicComponent
WaveFunction Collapse
ApplyWFC(equivalenceClasses, nodeStates)
Until WFC finished: (For all existing equivlance classes that are not 0)
InitializeWave()
- a certain equi-class has certain possible assignments - all initialized with true (=still possible to be assigned)
Observe(equivalenceClasses)
- finds location with lowest entropy - assigns final tile to location
for all existing "equivalence classes" (that belong to the same complex):
Propagate()
propagates the new assignment --> updates domains of neighboring locations
assignedTileNums

Figure 29: The sequence diagram shows the basic process of the Wave Function Collapse function. The process is triggered by the public method *ApplyWFC*.

The overview of the process of WFC is still rough here. The actual process is divided into many small sub-functions, but that coarseness in the diagram makes it easier to understand what WFC is doing.

4.3 Tools

Various tools were used for the concrete implementation of the project.

4.3.1 Agile Software Development

One tool for the implementation of my project was agile software development (at least as far as you can implement it in a one-person-team).

Iterative Software Development

One big aspect of agile software development is the iterative approach to development. In contrast to the waterfall model, where every step (analysis, design, implementation, testing, deployment) is executed sequentially and only once, the iterative approach goes through these phases repeatedly and always adds new functionality to the software [46].

The phases of my software development include the following sprints (a sprint is one pass through several steps of the development process and takes approximately three weeks):

1. In the first sprint, the structure of the project was roughly established. The *LogicComponent* was identified as the controller, *MarchingCubes* and *WaveFunctionCollapse* were initially integrated as empty classes.
2. Subsequently, the MC algorithm was roughly implemented. Initially, dummies (white cubes) were used instead of real building models and the basic functionality of MC was built in.
3. The first building models (3D tiles) were integrated and first basic functions of WFC could be implemented.
4. All basic functions of WFC were implemented. At this point, the algorithm only contained a single constraint, which states that only building parts of the same color may be next to each other.
5. All 3D tiles were modelled and integrated into the project. The MC algorithm was fully implemented, the equivalence classes were fully developed and assigned to a 3D tile.
6. The WFC algorithm was completely installed and a second constraint was added.

7. Finally, an additional function was built in, with which the user can choose which colors can be used for building generation. In addition, the building models and the surroundings in the game scene were refurbished.

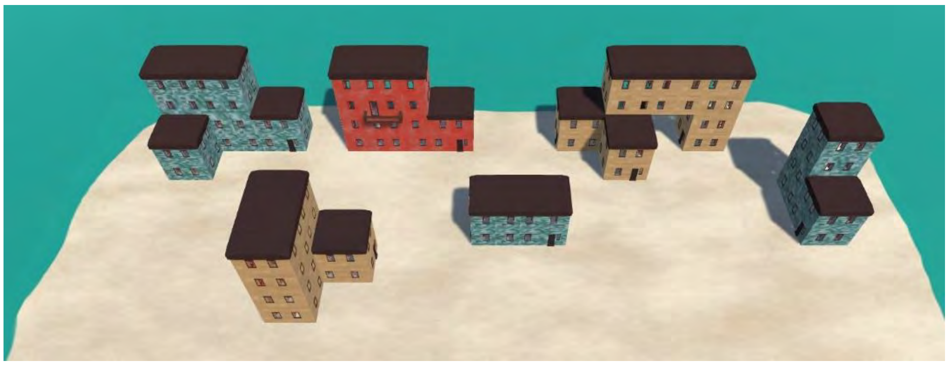


Figure 30: The final building models and refurbished surroundings in the game.

4.3.2 Technical Means

Game Engine and Development Environment

As already briefly mentioned, the application was developed with the game engine Unity. During the project, the Unity version was renewed several times. The latest version I used was: *Unity Version 2019.4.9f1 Personal*.

Furthermore, the code was developed in C# with the .NET Framework version 4.8 and the C# version 8.0. The code was developed in the Microsoft Visual Studio Community 2017 edition, version 15.7.3.

The utility program was developed in Eclipse 2020-03 with Java 11.

Assets, Libraries, and Additional Programs

I used solely the standard Unity Library *UnityEngine* and the standard C# *System* library.

The assets I used are mostly either self-created or standard objects offered by the Unity Engine. The only exception are the textures that I used for the 3D models of the buildings and for the underground in the game.[11]

[11] The different textures I used can be found here: https://cc0textures.com/ (accessed 2020/10/24). All textures are licensed under the Creative Commons CC0 License.

The following sources were used for the camera script that controls its movement in the game: [51], [52]. Furthermore, the *gitignore* I used bases on a template (licensed under the Creative Commons CC0 License) for Unity projects: [53].

The 3D models for the buildings parts where created in Blender v 2.8.2a.

Version Control

Git version 2.17.1.windows.2 was used for version control. My project was stored in the HAW GitLab.

I used branching for this project. As a result, there was always a well-functioning version of the project in the main branch[12] and I could work in the develop branch.

[12] In my git version, the main branch is still called "master", but git renamed it for reasons of sensitivity towards minorities, see: https://www.zdnet.com/article/github-to-replace-master-with-main-starting-next-month/ (accessed 2020/10/24).

5 Evaluation

The following chapter is intended to evaluate the results of this thesis. The achievements of this thesis are shown as well as the extent to which the set goals were implemented and the requirements met.

5.1 Goals

An application for procedural building generation in 3D that is based on the algorithms MC and WFC was created. The two algorithms were successfully modified and combined.
The MC algorithm was extended in such a way that it can be used for user-initiated content generation. In my application, MC is used together with WFC, but my implementation of MC could also generate content on its own (with the well-known limitation that it has one clearly defined solution for one node state).

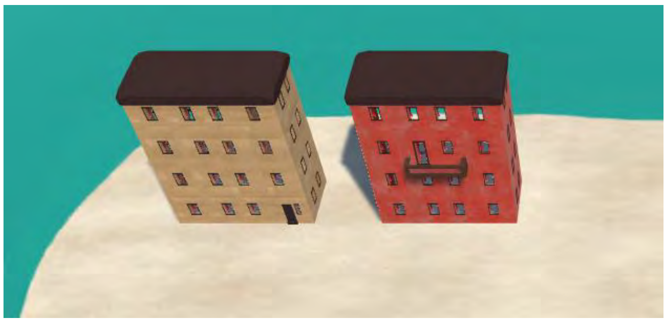


Figure 31: The two buildings would look the same if MC was used on its own. Through WFC they differ.

My modification of WFC is set on top of MC. Every cell combination that would result in one solution with MC gets several possible solutions with WFC. The different possible 3D tiles for one node state are weighted and subject to constraints. Constraint solving decides about the concrete 3D tiles.

Thus, the goals stated in chapter 1.2 have been achieved and an alternative use of the algorithms WFC and MC could be shown.

5.2 Requirements

The following explains the extent to which the requirements specified in the analysis have been implemented. The user stories and use cases defined the functional requirements for this work. The required functions were all implemented, as can be seen in the application. The implementation of the non-functional requirements should be checked in detail below.

The general quality requirements which I wanted to pay particular attention to are:

- maintainability
- extensibility
- usability

Further quality requirements that I set up are:

- six properties that should apply to the generated buildings
- playability for people of all ages
- intuitive usability

5.2.1 Maintainability and Extensibility

Methods

I implemented the requirements maintainability and extensibility by using the principles: modular design, low coupling, high cohesion, separation of concerns (SoC), single-responsibility, and interface segregation (for an overview of principles of object-oriented design see [54]). Another method that I have used is testing the software.

The principles were implemented through a meaningful division into components and classes. The modules are decoupled and interchangeable insofar that components (almost) exclusively communicate with one another via interfaces. Most classes contain one public method that is used to communicate with the coordinator (the component class *LogicComponent*). Almost all communication within the component takes place via the component class (see the class diagram in chapter 4.2.2).

Wherever possible, classes have been implemented in such a way that they have a clearly defined area of responsibility. Attempts were also made to give them a single responsibility (e.g. *MarchingCubes* is exclusively responsible for the implementation of the MC algorithm). This could be implemented quite well in the logic component; however, this is only partly implemented in the UI component. The *UIHandler*'s area of responsibility is the mapping of the patterns that were assigned in the logic component to concrete 3D tiles. In addition, this class handles the user input. In the *Update()* method, the input of the user is checked and, if necessary, passed on to the logic component. During the implementation, I regarded I/O as one area of responsibility, but actually there are two different responsibilities here.

As already mentioned in Chapter 4.2.2, some principles (of e.g. modularity, low coupling, SoC) were not followed in one case. The script that controls the buttons for the color selection belongs to the UI component but accesses the *WaveFunctionCollapse* class directly. This was done due to technological necessities and to avoid a large overhead.

During the software engineering process, individual functionalities were constantly tested and, if necessary, improved. Ideally, systematic unit and component tests would have been carried out as part of Continuous Integration with regular automatic builds and tests. In the context of this project, this approach might not be in proportion to the additional quality gained and was omitted. However, this step would be essential in a bigger project.

Metrics

The established metrics should check the success in achieving the quality requirements.

The **velocity** estimates the time a programmer needs to develop a product [48]. It is a good metric for the maintainability as well as the extensibility of a software product. If a software project can be easily expanded and maintained, the velocity is high because programming is much faster. It is difficult to put the velocity of my project into proportions. Nevertheless, the iterative work during the project went well. This is due to the fact that basic functionalities were built in, first, and the extensions later. During this process, adjustments sometimes had to be made to the modules, but since the extensions were planned from the beginning, it did not require much additional time.

The **failure and repair time** is a good indicator for software maintainability. The shorter the time between the occurrence of an error and its repair, the easier it is to maintain the software.
Of course, failures did occur during development, but due to the modular structure and the use of the other principles, it was easy to narrow down the area in which these errors occurred. For example, there was a problem with the correct rotation of the 3D tiles. Although the algorithm selected and positioned the tiles correctly, the rotations were wrong, so that, for example, parts of the roof pointed downwards. However, since there was only one module that was responsible for the rotation, the source of the error could soon be identified. The failure and repair time was, thus, relatively low.

Low **code churn** is another indicator for good software quality in terms of maintainability and extensibility. Code churn displays how often the code had to be changed. At best, the code churn should get smaller towards the end of the project [55].

To determine the code churn in a simple way, I regarded the added lines in comparison to the deleted lines:

$$Code\ churn = deleted\ lines - \ added\ lines$$

The lower the value that is calculated in this way, the fewer lines were deleted compared to added. This is a very simple approach that could be implemented using Git. All added and deleted lines could be determined for each commit in chronological order. However, it is difficult to determine which result would represent low code churn. For this reason, I compare the code churn of the first commits with that of the last commits. The code churn should be lower for the later commits. All added and deleted lines of the commits can be seen in Appendix A1: Code Churn.[13]
For the first 37 commits, I had 1335 added and 1119 deleted lines, resulting in a code churn of: –216. For the latest 37 commits, I had 2146 added and 964 deleted lines, what results in a code churn of: –1182.
As you can see, the code churn decreased towards the end of the project. This is an indication of good software quality in terms of how easily it can be maintained and extended.

Overall, the requirements maintainability and extensibility were mostly met. In some cases, the implementation of these requirements was waived to avoid unnecessary overhead. This middle ground allowed to implement extensions and corrections quickly during software development and appeared like the best way.

5.2.2 Usability

The requirements intuitive usability and playability for people of all ages can be summarized in the generic term usability. The following information on software usability can be found in [44].

Methods

Usability is a quality attribute that defines how easy it is to interact with a user interface. Jakob Nielsen stated five quality components of good usability:

- Learnability (using a system should be easy to learn)
- Efficiency (the system should be efficient to use)
- Memorability (the usage should be easy to remember)
- Errors (users should make as few mistakes as possible)
- Satisfaction (users should be satisfied while working with the system)

[13] The commits were compared at a time when the project was finished. Afterwards only further comments and three more 3D tiles were added.

The ideal program would have no learning curve but would be easy to use right from the start. This is also the aim when talking about intuitive usability. Of course, this goal can never be fully implemented. I attempted to approach this aim by keeping the possible interactions of the user with the system to a minimum which should also contribute to memorability. Nevertheless, a short README was also created as instructions for the user.

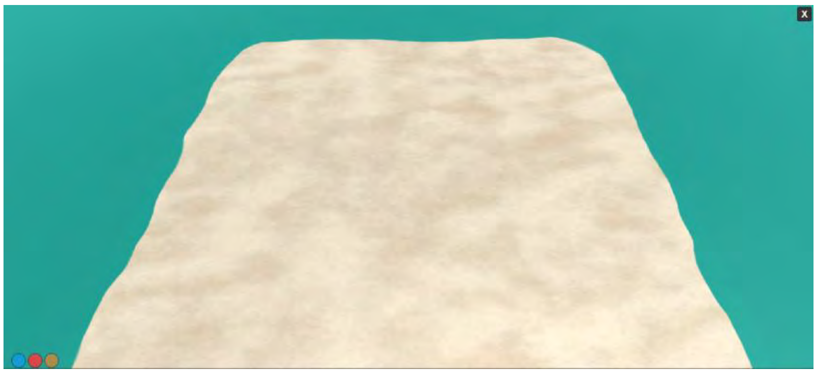


Figure 32: For a better usability, interactions with the UI are kept simple: Buildings can be generated on the island, colors can be selected, and the application can be ended. Users can also move and zoom the camera.

The actions of the users have a direct influence on the system, which can be seen in the UI. The only time the system will not respond is if the user input is invalid (e.g. when users try to delete a house at a point where there is none). The users therefore receive direct feedback on their actions. This contributes to good learnability and efficiency. It still allows users to make mistakes (as invalid entries), but it should be easy to see which entries are valid and which are not. Furthermore, the ground in the scene is designed in such a way that users can see where houses can and cannot be built. The area in which the houses can be generated looks like an island, while the rest looks like water.
The user's satisfaction is a subjective feeling. However, the points mentioned above should also contribute to a greater satisfaction.

Verification

To check the extent to which good usability has been achieved, usability tests would have to be carried out. Unfortunately, no usability tests could be carried out for this thesis due to lack of time. It would be advisable to let test persons from different age groups try the game. Moreover, different tasks would have to be assigned to people from the same age groups. Possible tasks are creating houses in certain shapes, using certain colors, etc. Subsequently, a questionnaire could be filled out or an oral survey could be carried out. It would also be interesting to let the test persons try the game without specific instructions and see whether they recognize the functionalities on their own.

Based on my own reviews of the software, it is difficult to see how well I have achieved the set goals for good usability. Nonetheless, I could still make out some problems.
The camera can be moved parallel to the ground and zoomed in with the middle mouse button, but it would be nice if it were also possible to select a point in the game and rotate the camera around it.
In addition, the selection of a position in the grid at which a building should be created or deleted sometimes does not work perfectly. A possible reason for this could be that the colliders of the 3D tiles are not attached perfectly. The colliders are used to register clicks in the game. Based on the position that is clicked on, a “ray” is thrown into the scene, which then collides with the collider of the 3D tile. If a 3D tile is missing a collider, it is possible that a click is not registered or that the ray even hits a collider of another tile.

5.2.3 Properties of Generated Buildings

I stated six properties that should apply to the generated buildings. The achievement of these goals is checked below.

Achievements

It is possible to place buildings on a designated area in the game that is easy to identify. The ground in the game is designed in a way that the area in which buildings can be generated looks like an island while the rest (outside the grid) looks like water. The distinction between land and water can be seen in Figure 33. The buildings in this game scenario are placed at the outer edges on which buildings can still be built.

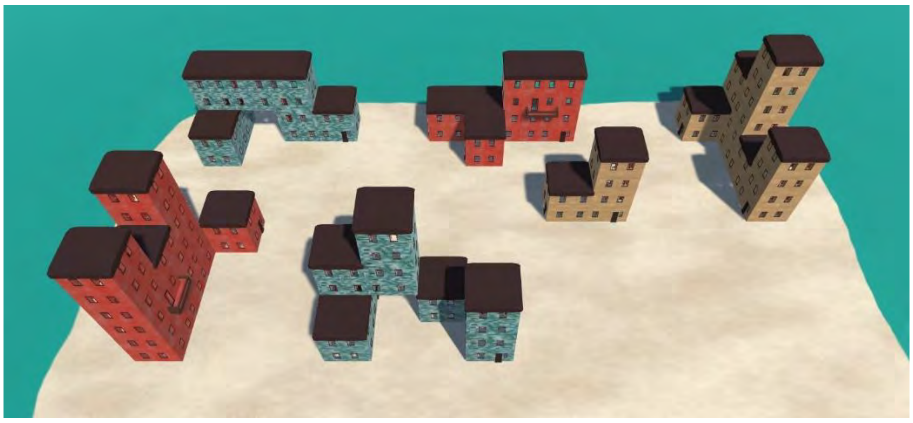


Figure 33: Buildings are set at the outer edges of the designated area in the game.

A constraint stipulated that building tiles may only be placed next to each other if they correspond to the same color. This can be seen in Figure 33. You can also see the implementation of the constraint, which states that doors may only be on the floor and not directly next to each other.

It can also be seen that the buildings have a consistent appearance. The walls and roofs fit exactly together. All parts of the buildings connect to one another and there are no open house walls.

Building parts can furthermore be connected to each other over all possible corners, edges, and surfaces. This was achieved by mapping all 256 possible node states to equivalence classes and by assigning a corresponding 3D tile to every equivalence class. Thereby, all possible cube combinations are covered.

Difficulties and Possible Extensions

As planned, doors only appear on the floor and they occur in different positions of the buildings. However, it could happen that a building has no door at all. That is because the possibilities in the wave for a door occur with a certain probability.
If it had been determined that a building must always have a door, then a door would always appear in the same place, namely where the algorithm begins to assemble the building. This is due to the way WFC works, i.e. because WFC builds images iteratively.
To avoid that a building does not have a door, the probability for doors was set very high. As a result, the doors still often appear in the same place, but not always as can be seen in Figure 33.

In fact, the generated buildings have a consistent appearance, but the 3D tiles could be more versatile. The focus of this thesis was to show how the WFC and MC algorithms can be combined to create buildings using predesigned tiles. Nevertheless, the design of the buildings is quite uniform. Different types of roofs or planting on the façades could be added. With the help of WFC, these additional 3D tiles could be used depending on constraints and a certain probability which would lead to more variety.

5.3 Possible Improvements

In chapter 5.2.3, suggestions were already made on how the generation of the buildings could be improved. The following ideas apply to the entire project and could not be implemented due to time constraints.

The size of the grid could be adjustable. This could either have been implemented in such a way that the island automatically enlarges when buildings are set on its edge. Or in a way that the user can enlarge the island by clicking on its edges.

The possible expansion of a building could be displayed in the UI when you hover the mouse over a part of the building. The area to which a new part of the building would be attached could be highlighted.

At the moment, buildings can only have one color (this is regulated by a constraint). However, it would also be possible to incorporate “intermediate pieces”, which allow two colors to be combined. For example, these intermediate pieces would be red on one side and blue on the other with a color transition in between.

The color selection is implemented in a way that all colors are selected at the beginning and the color of the generated buildings is decided by WFC. This was done to demonstrate the functionality of WFC with help of the color constraint. This implementation seems tedious when only one color is selected, and you want to choose another color. In this case the new color must be selected and the previous color deselected, i.e. two clicks are necessary. Alternatively, only one color can be selected at a time, but this would make the use of WFC less obvious.

6 Conclusion and Perspective

This chapter summarizes the thesis and addresses some opportunities for further development.

6.1 Conclusion

In this thesis it was shown how the algorithms Wave Function Collapse (WFC) and Marching Cubes (MC) can be used to generate buildings in 3D. The generation works on the "tile level", i.e. previously created 3D models are assembled to create consistent buildings. Furthermore, the generation is user-initiated: By selecting and deselecting certain cells in a grid, the user determines the surface of the buildings. While MC identifies the course of the building surfaces, WFC decides on the concrete appearance of the façades.

The way in which the various parts of the buildings must be assembled is determined by equivalence classes. The equivalence classes contain the node states and each node in the grid is assigned to an equivalence class. Overall, they depict all possible ways in which cubes can be connected.

My modification of the MC algorithm basically determines the surface on which the building façades are created by assigning the individual node states to their equivalence classes. Thus, the basic shape of the 3D tiles is determined. WFC then decides which specific 3D tile is used.

My WFC adaptation works on top of the results of MC. For each equivalence class assigned to a particular node WFC chooses from at least three different options. Depending on constraints and weights, the algorithm determines which specific pattern should be selected for every node in the grid. The chosen patterns correspond to concrete 3D tiles that are placed in the position of the node in the visible grid.

6.2 Perspective

Possible extensions to my application have already been discussed in 5.2.3 and 5.3. At this point, possibilities are discussed how WFC and MC could be used for further generation tasks.

As with the original WFC, an input image can also be used in 3D. Based on a 3D input image, 3D tiles can be derived, which can then be used to automatically create larger 3D images. On the one hand, this would be possible at voxel level, so that WFC uses voxels instead of pixels. On the other hand, this is also conceivable at tile level, so that tiles (as separate 3D models) in a certain size are taken from the image and assembled to form a result image. This would probably lead to similar results as Merrell's approach to model synthesis (see 2.2.2). Instead of rules and the assignment of labels to the individual 3D tiles (as in model synthesis), constraints would be automatically derived from the input image which determine how the tiles may be assembled.

Another possibility to expand my approach would be to use WFC and MC to automatically generate levels instead of using user input. If, e.g., Perlin noise is used to randomly distribute the buildings in a scene. In this case, however, some restrictions must be established so that the buildings look authentic. For example, some rules had to stipulate the maximum height buildings can have. Furthermore, weights could be used to determine the average height or the density of buildings.

After all, it would be interesting not only to assemble buildings with WFC using 3D tiles, but also to assemble the tiles with WFC. This allows the building parts to be composed of individual walls, windows, doors, and decorative elements. Thereby, WFC would be used on two levels: firstly, to generate the parts of the buildings and, secondly, to generate the buildings. This would probably lead to even more diverse results.

7 Bibliography

[1] N. Shaker, J. Togelius and M. J. Nelson, Procedural Content Generation in Games: A Textbook and an Overview of Current Research, Cham, Switzerland: Springer, 2016.

[2] O. Stålberg, "Townscaper," 2020. [Online]. Available: https://store.steampowered.com/app/1291340/Townscaper/. [Accessed 04 11 2020].

[3] W. Lorensen and H. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *ACM SIGGRAPH Computer Graphics, Volume 21, Number 4,* pp. 163-169, July 1987.

[4] I. Karth and A. M. Smith, "WaveFunctionCollapse is Constraint Solving in the Wild," *Proceedings of the 12th International Conference on the Foundations of Digital Games,* August 2017.

[5] J. K. E. S. D. Y. G. Togelius, "What is procedural content generation? Mario on the borderline.," *Proceedings of the 2nd Workshop on Procedural Content Generation in Games,* 28 June 2011.

[6] M. M. S. v. d. V. J. a. I. A. Hendrikx, "Procedural Content Generation for Games: A Survey," *ACM Transactions on Multimedia Computing, Commununications, and Applications,* February 2011.

[7] J. Freiknecht and W. Effelsberg, "A Survey on the Procedural Generation of Virtual Worlds," *Multimodal Technologies and Interaction,* 30 October 2017.

[8] K. Compton, J. C. Osborn and M. Mateas, "Generative methods," *The Fourth Procedural Content Generation in Games workshop, PCG,* May 2013.

[9] J. Togelius, G. Yannakakis, K. Stanley and C. Browne, "Search-based procedural content generation," *Applications of Evolutionary Computation,* 2010.

[10] A. Lindenmayer and P. Prusinkiewicz, The Algorithmic Beauty of Plants, Berlin, Germany: Springer Science & Business Media, 2004.

[11] R. M. Smelik, K. J. De Kraker, S. A. Groenewegen, T. Tutenel and R. Bidarra, "A survey of procedural methods for terrain modelling," in *Proceedings of the CASA workshop in 3D advanced media in gaming and simulation*, A. Egges, W. Hürst and R. C. Veltkamp, Eds., 2009, pp. 25-34.

[12] G. S. P. Miller, "The Definition and Rendering of Terrain Maps," *SIGGRAPH*

’86:Proceedings of the13thAnnual Conference on Computer Graphics and Interactive Techniques,volume 20, p. 39–48, 18-22 August 1986.

[13] K. Perlin, "An Image Synthesizer," *SIGGRAPH ’85: Proceedings of the12s tAnnual Conference on Computer Graphics and Interactive Techniques, volume 19,* p. 287–296, 22-26 July 1985.

[14] A. G. E. M. S. a. G. J. Peytavie, "Arches: a Framework for Modeling Complex Terrains," *Eurographics 2009 Proceedings,* vol. II, no. 28, 2009.

[15] R. Linden, R. Lopes and R. Bidarra, "Procedural Generation of Dungeons," *IEEE Transactions on Computational Intelligence and AI in Games,* March 2014.

[16] I. D. Horswill and L. Foged, "Fast Procedural Level Population with Playability Constraints," *The Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment,* January 2012.

[17] A. M. Smith and M. Mateas, "Answer Set Programming for Procedural Content Generation: A Design Space Approach," *IEEE Transactions on Computational Intelligence and AI in Games,* 7 June 2011.

[18] X. Neufeld, S. Mostaghim and D. Perez-Liebana, "Procedural Level Generation with Answer Set Programming for General Video Game Playing," *7th Computer Science and Electronic Engineering Conference, CEEC 2015 - Conference Proceedings.,* 2015.

[19] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Transactions on Affective Computing 2,* 5 April 2011.

[20] R. M. Smelik, T. Tutenel, R. Bidarra and B. Benes, "A Survey on Procedural Modelling for Virtual Worlds," *Computer Graphics Forum 33(6),* p. 31–50, January 2014.

[21] G. Stiny and J. Gips, "Shape Grammars and the Generative Specification of Painting and Sculpture," *IFIP Congress,* 1971.

[22] G. Stiny, Pictorial and Formal Aspects of Shape and Shape Grammars, Basel: Birkhauser Verlag, 1975.

[23] S. Y. Shing, "Shape Grammar," 2012. [Online]. Available: http://sohyoushingbscarc2013.blogspot.com/2014/12/shape-grammar.html. [Accessed 30 7 2020].

[24] P. Wonka, M. Wimmer, F. Sillion and W. Ribarsky, "Instant Architecture," *ACM Transaction on Graphics, 22(3),* pp. 669-677, July 2003.

[25] P. Müller, P. Wonka, S. Haegler, A. Ulmer and L. V. Gool, "Procedural modeling of buildings," *ACM SIGGRAPH,* pp. 614-623, December 2006.

[26] D. Finkenzeller and J. Bender, "Semantic representation of complex building structures," *Computer Graphics and Visualization,* p. 259–264, July 2008.

[27] P. Müller, Z. Gang, W. Peter and G. L. Van, "Image-based procedural modeling of facades," *ACM Transactions on Graphics,* pp. 85:1-85:10, July 2007.

[28] P. Birch, S. Browne, V. Jennings, A. Day and D. Arnold, "Rapid Procedural-modelling of Architectural Structures," *Proceedings of the 2001 conference on Virtual reality,*

archeology, and cultural heritage, p. 187–196, November 2001.

[29] P. Merrell, E. Schkufza and V. Koltun, "Computer-generated residential building layouts," *ACM Transactions on Graphics 29(6),* p. 181, December 2010.

[30] P. Merrell, "Model synthesis," Ph.D. Dissertation, University of North Carolina at Chapel, 2009.

[31] M. Saldana and C. Johanson, "Procedural Modeling for Rapid-Prototyping of Multiple Building Phases," *Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci., XL-5/W1,* pp. 205-210, 2013.

[32] S. Greuter, J. Parker, N. Stewart and G. Leach, "Real-time procedural generation of ‘pseudo infinite’ cities," *GRAPHITE ’03: Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia,* pp. 87-94, 2003.

[33] L. Raad, A. Davy, A. Desolneux and J.-M. Morel, "A survey of exemplar-based texture synthesis," *Annals of Mathematical Sciences and Applications,* 17 October 2018.

[34] M. Tuceryan and A. K. Jain, "Texture Analysis," in *The Handbook of Pattern Recognition and Computer Vision (2nd Edition)*, NJ, United States, 1998, p. 207–248.

[35] A. A. Efros and T. K. Leung, "Texture Synthesis by Non-parametric Sampling," *Proceedings of IEEE International Conference on Computer Vision,* September 1999.

[36] M. Khokhlov, I. Koh and J. Huang, "Voxel Synthesis for Generative Design," in *Design Computing and Cognition '18*, Springer, Ed., Cham, Switzerland, 2019, pp. 227-244.

[37] H. Kim, S. Lee, H. Lee, T. Hahn and S. Kang, "Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm," 2019. [Online]. Available: https://ieee-cog.org/2019/papers/paper_187.pdf. [Accessed 14 08 2020].

[38] O. Stålberg, "EPC2018 - Oskar Stalberg - Wave Function Collapse in Bad North," 11 07 2018. [Online]. Available: https://www.youtube.com/watch?v=0bcZb-SsnrA. [Accessed 19 08 2020].

[39] O. Stålberg, "ORGANIC TOWNS FROM SQUARE TILES - a talk by OSKAR STÅLBERG at INDIECADE EUROPE 2019," 17 03 2020. [Online]. Available: https://www.youtube.com/watch?time_continue=29&v=1hqt8JkYRdI&feature=emb_logo. [Accessed 20 08 2020].

[40] O. Stålberg, *Bad North,* Malmö, Sweden: Raw Fury, 2018.

[41] O. Stålberg, "Voxel House," R. Hawkins, Ed., pp. 170-175.

[42] W. Zuser, S. Biffl, T. Grechenig and M. Köhle, Software Engineering mit UML und dem Unified Process, 1 ed., München, Deutschland: Pearson Studium, 2001.

[43] C. Bradley, "Scrum: Who are the Key Stakeholders that Should be Attending Every Sprint Review?," 2015. [Online]. Available: https://www.scrum.org/resources/blog/scrum-who-are-key-stakeholders-should-be-attending-every-sprint-review. [Accessed 14 10 2020].

[44] T. Grechenig, M. Bernhart, R. Breitender and K. Kappel, Softwaretechnik. Mit

Fallbeispielen aus realen Entwicklungsprojekten, München: Pearson Studium, 2010.

[45] A. Naumann, "Gute User-Storys schreiben – alles rund um User-Storys," 2012. [Online]. Available: https://blog.ibo.de/agile-business-analyse-alles-rund-um-user-stories/. [Accessed 17 10 2020].

[46] J. Ludewig and H. Lichter, Software Engineering. Grundlagen, Menschen, Prozesse, Techniken, 3. ed., Heidelberg: dpunkt.verlag GmbH, 2013.

[47] K. C. IBM, "Use-case template," 2020. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSWSR9_11.6.0/com.ibm.pim.dev.doc/pim_ref_usecasetemp.html. [Accessed 17 10 2020].

[48] D. Writer, "Top 10 Software Quality Metrics That Matter," 2020. [Online]. Available: https://diceus.com/top-7-software-quality-metrics-matter/. [Accessed 29 10 2020].

[49] R. Friesen, M. Stollenwerk and D. Valentin, "MVC Pattern," 2007. [Online]. Available: https://public.hochschule-trier.de/~rudolph/gdv/cg/node46.html. [Accessed 22 10 2020].

[50] T. Erl, "Utility Abstraction," 2008. [Online]. Available: https://patterns.arcitura.com/soa-patterns/design_patterns/utility_abstraction. [Accessed 22 10 2020].

[51] Learn Everything Fast, "Move Camera with Mouse in Unity 3D," 2017. [Online]. Available: https://www.youtube.com/watch?v=lmmf-6xmUk4. [Accessed 24 10 2020].

[52] Design and Deploy, "Unity 5 - How to Zoom Camera with Mouse Scroll Wheel," 2016. [Online]. Available: https://www.youtube.com/watch?v=4yK4PoZQ4qI. [Accessed 24 10 2020].

[53] GitHub, "Unity.gitignore," 2020. [Online]. Available: https://github.com/github/gitignore/blob/master/Unity.gitignore. [Accessed 24 10 2020].

[54] M. Umair, "SOLID, GRASP, and Other Basic Principles of Object-Oriented Design," 2018. [Online]. Available: https://dzone.com/articles/solid-grasp-and-other-basic-principles-of-object-o. [Accessed 31 10 2020].

[55] T. Osbourn, "What is Code Churn and How to Reduce it," 2019. [Online]. Available: https://textexpander.com/blog/what-is-code-churn-and-how-to-reduce-it/. [Accessed 6 11 2020].

8 Appendix

8.1 A1: Code Churn

8.1.1 List of All Commits

The left column contains the added lines, the middle column the deleted lines, and the right column contains the file names. Only .cs files were considered. At the end the added and deleted lines are summed up.

git log --numstat --pretty="%H" *.cs		
507be55d73b247acbafbb93afa97b4f03ba1c031		
6	6	Assets/Scripts/ButtonLogic.cs
3	3	Assets/Scripts/CameraScript.cs
2	0	Assets/Scripts/IHandleUserInput.cs
2	0	Assets/Scripts/IUpdateScene.cs
6	1	Assets/Scripts/LogicComponent.cs
1	5	Assets/Scripts/MarchingCubes.cs
9	4	Assets/Scripts/UIHandler.cs
18	13	Assets/Scripts/WaveFunctionCollapse.cs
c19177463bb55823acf4113e2806f630c27809fb		
1	2	Assets/Scripts/Grid.cs
1	1	Assets/Scripts/LogicComponent.cs
bc92558258cc0d20afacd8a8b2b108914f5f198e		
16	11	Assets/Scripts/CameraScript.cs
086e642c661797b7d16f0b97f09157d1c986dcae		
0	40	Assets/Scripts/Grid.cs
0	1	Assets/Scripts/UIHandler.cs

1	1	Assets/Scripts/WaveFunctionCollapse.cs
34fae2d57e231b0adbcba54a3dc30c44cc4f4e1b		
128	0	Assets/Scripts/ButtonLogic.cs
6f578cefad24aac89e99df23195f3aba757e6f58		
5	5	Assets/Scripts/WaveFunctionCollapse.cs
b80c62a1e17951cd203fd75f09996e08cc657faa		
19	14	Assets/Scripts/LogicComponent.cs
1	2	Assets/Scripts/UIHandler.cs
950d6caefdf0d395cb8de9b96cd729827cfba939		
2	5	Assets/Scripts/UIHandler.cs
6	6	Assets/Scripts/WaveFunctionCollapse.cs
fb9a6ed3fb0f90e858fdc9f196c74ff842960f95		
98	158	Assets/Scripts/UIHandler.cs
2	2	Assets/Scripts/WaveFunctionCollapse.cs
87d88f0eecf58803aa456555bc36a26fa9c011d4		
3	0	Assets/Scripts/UIHandler.cs
2	2	Assets/Scripts/WaveFunctionCollapse.cs
53b909125d3b0bda549810dfcea6cc885621e215		
1	1	Assets/Scripts/UIHandler.cs
4	4	Assets/Scripts/Util.cs
2	2	Assets/Scripts/WaveFunctionCollapse.cs
3de4d6fa3329160b3a4deb1560f793d759895bd4		
0	1	Assets/Scripts/MarchingCubes.cs
1	1	Assets/Scripts/UIHandler.cs
3	3	Assets/Scripts/Util.cs
34	54	Assets/Scripts/WaveFunctionCollapse.cs
2b11ecfecebfe1cc1d2a5c4f8f9c679592b2ebc4		
39	14	Assets/Scripts/WaveFunctionCollapse.cs
f08b92637c0ae120544847e18875febde9d56925		

1	0	Assets/Scripts/MarchingCubes.cs
1	1	Assets/Scripts/UIHandler.cs
3	3	Assets/Scripts/Util.cs
15	20	Assets/Scripts/WaveFunctionCollapse.cs
f82e77c505eda9e320592c862c6763251a600e36		
14	39	Assets/Scripts/WaveFunctionCollapse.cs
95804d0a75c5a8518846cd8999379c6162343cdc		
2	2	Assets/Scripts/UIHandler.cs
2	2	Assets/Scripts/Util.cs
67	26	Assets/Scripts/WaveFunctionCollapse.cs
19677fc6ba4d0ce24ff1a66a8a5c7d964ea891b6		
14	18	Assets/Scripts/UIHandler.cs
68	31	Assets/Scripts/WaveFunctionCollapse.cs
c7fc5cb54f04d434d0c08039e4be029334fe3981		
1	1	Assets/Scripts/Util.cs
14	25	Assets/Scripts/WaveFunctionCollapse.cs
121331465b41dd786dd89c8327033661f6d190a0		
0	15	Assets/Scripts/MarchingCubes.cs
8	50	Assets/Scripts/UIHandler.cs
82	77	Assets/Scripts/Util.cs
cebee0f399b32ed212b5d7126d64efbbcf6edded		
5	0	Assets/Scripts/UIHandler.cs
7	1	Assets/Scripts/Util.cs
ef1bb188a7f0c63a364f946c5d0258417a5fc448		
3	0	Assets/Scripts/UIHandler.cs
10	4	Assets/Scripts/Util.cs
b3411025f1c6bf6c0b9863ac32fd9f5388f1673b		
9	0	Assets/Scripts/UIHandler.cs
13	2	Assets/Scripts/Util.cs

f38df9f0ad6ac344b2e44dc948a2c3b2af7946c1		
15	0	Assets/Scripts/UIHandler.cs
21	0	Assets/Scripts/Util.cs
f1e82c8aa44e0fcbca80a22aa0de67b7ff629b25		
13	0	Assets/Scripts/UIHandler.cs
15	0	Assets/Scripts/Util.cs
a244c64eeb9b3da02c0931159dd6971beb3e71ac		
5	0	Assets/Scripts/UIHandler.cs
8	2	Assets/Scripts/Util.cs
9150018a1e00b0eff6180f536a6abc1d5501a7a4		
13	0	Assets/Scripts/UIHandler.cs
16	0	Assets/Scripts/Util.cs
cdfd945cbb1afbf93666b7197fd7abea9d31dc80		
13	0	Assets/Scripts/UIHandler.cs
18	3	Assets/Scripts/Util.cs
aa8a4bf5c7e9eaea191237a9bae4aa39f3745395		
13	6	Assets/Scripts/UIHandler.cs
15	6	Assets/Scripts/Util.cs
b89ab889adf12bd14aed4b23b058e0cb753e86d0		
4	4	Assets/Scripts/WaveFunctionCollapse.cs
cbf4012a54b924cb1cdea26874232d997ef107a2		
5	0	Assets/Scripts/UIHandler.cs
7	0	Assets/Scripts/Util.cs
d0170f4718b3d8e61a29947cb6d498f03100480d		
5	0	Assets/Scripts/UIHandler.cs
8	0	Assets/Scripts/Util.cs
9a2211832aec28114b9db9fabd84d68c12de60bc		
10	2	Assets/Scripts/LogicComponent.cs

19	0	Assets/Scripts/UIHandler.cs
22	14	Assets/Scripts/Util.cs
f44d899fe6cccfa6209096633fef4274910d4b55		
1	1	Assets/Scripts/LogicComponent.cs
3	0	Assets/Scripts/UIHandler.cs
32	11	Assets/Scripts/Util.cs
10f870e5a102a65ec9443e11aa7dac56971f897c		
5	0	Assets/Scripts/LogicComponent.cs
8	4	Assets/Scripts/UIHandler.cs
44	197	Assets/Scripts/Util.cs
9d066a3c6be021ea14bdc400d729dea9f203a79a		
38	10	Assets/Scripts/UIHandler.cs
157	57	Assets/Scripts/Util.cs
ce54ed8df87917c795d43b396e2ee311cd4232a8		
2	1	Assets/Scripts/UIHandler.cs
0b7266f595883f9e0849f53c50133dba3a9bfb41		
45	112	Assets/Scripts/UIHandler.cs
0	10	Assets/Scripts/Util.cs
d106720aac1b54828e0a8825834be016baa6ce58		
1	1	Assets/Scripts/MarchingCubes.cs
11	0	Assets/Scripts/Util.cs
1f7dabb93b43e7064f193000fc1b95ad9c06c944		
34	0	Assets/Scripts/UIHandler.cs
77	34	Assets/Scripts/Util.cs
f962d107ffc078ff450db4f7b9928efc381ddf66		
29	4	Assets/Scripts/UIHandler.cs
4f037c2da882665d303ab355df118dd3419c6312		
0	68	Assets/Scripts/CubePositions.cs
17	24	Assets/Scripts/MarchingCubes.cs

12	11	Assets/Scripts/UIHandler.cs
129	9	Assets/Scripts/Util.cs
1	1	Assets/Scripts/WaveFunctionCollapse.cs
3546ac1310a380a67e4612a0c3738f2c44f74ebe		
1	1	Assets/Scripts/LogicComponent.cs
17	26	Assets/Scripts/UIHandler.cs
6	1	Assets/Scripts/{SharedItems.cs => Util.cs}
80	24	Assets/Scripts/WaveFunctionCollapse.cs
6204ef4d44968c275323e2c72d75f322bb0e269d		
140	55	Assets/Scripts/WaveFunctionCollapse.cs
99a5b7703e86395c245e63c4be448d8ed0a7a777		
40	0	Assets/Scripts/CameraScript.cs
81	29	Assets/Scripts/WaveFunctionCollapse.cs
0aa7c2696b5dca9863432b7a40f52d51531b987f		
1	23	Assets/Scripts/Grid.cs
1	1	Assets/Scripts/IHandleUserInput.cs
8	16	Assets/Scripts/LogicComponent.cs
15	4	Assets/Scripts/UIHandler.cs
b9752af4e55c53d45eb58432fa415d922c758709		
18	0	Assets/Scripts/SharedItems.cs
8	6	Assets/Scripts/UIHandler.cs
1	4	Assets/Scripts/WaveFunctionCollapse.cs
eee72197d031bee4a97d6b4afb33dd3b6131a800		
24	62	Assets/Scripts/WaveFunctionCollapse.cs
89cbb9177d87df901f3b25ce4e326ca0721ac1fa		
9	7	Assets/Scripts/UIHandler.cs
66	11	Assets/Scripts/WaveFunctionCollapse.cs
ab426c1dc4096d2daed64b3686f211fb0bf1fd24		
24	30	Assets/Scripts/WaveFunctionCollapse.cs

710d45398bc2bd671495dd7fd918c8ee573fe683		
12	7	Assets/Scripts/UIHandler.cs
70	52	Assets/Scripts/WaveFunctionCollapse.cs
6b0fa6df7810f0338b99715b1bcd4a85dca40f07		
73	10	Assets/Scripts/WaveFunctionCollapse.cs
d1577f50291cb9964436e9c00d7756e80c64dff3		
4	1	Assets/Scripts/CubePositions.cs
2	0	Assets/Scripts/MarchingCubes.cs
35	12	Assets/Scripts/UIHandler.cs
140	2	Assets/Scripts/WaveFunctionCollapse.cs
3dd4697749a0c663de8968340fa94360c3284b25		
11	3	Assets/Scripts/CubePositions.cs
4	1	Assets/Scripts/MarchingCubes.cs
17	0	Assets/Scripts/UIHandler.cs
190310ca81c6814a1253de03ae4aaa7a379ab945		
0	2	Assets/Scripts/UIHandler.cs
2538728f0fab2244c6006a4bb9f8a79169741645		
41	26	Assets/Scripts/UIHandler.cs
1f7921e1a004f0e007b7c28700c4f2aa3637e0bd		
41	97	Assets/Scripts/UIHandler.cs
0b33e05fad7b1c08dff6ad166818da04a742443a		
10	15	Assets/Scripts/UIHandler.cs
f6ebfc3253322b1ec259439ccd4061a42c8285e3		
2	0	Assets/Scripts/CubePositions.cs
86	93	Assets/Scripts/UIHandler.cs
ba0da5f833bb86c3261d9e733d853681de6c460b		
2	1	Assets/Scripts/CubePositions.cs
9	1	Assets/Scripts/MarchingCubes.cs

39	5	Assets/Scripts/UIHandler.cs
9233a1925badffeac59aff8285f0620217e6121a		
1	1	Assets/Scripts/CubePositions.cs
1	1	Assets/Scripts/IUpdateScene.cs
20	13	Assets/Scripts/LogicComponent.cs
136	125	Assets/Scripts/UIHandler.cs
5	0	Assets/Scripts/WaveFunctionCollapse.cs
982f7c12757a243c254954df2eb70084851aa648		
19	2	Assets/Scripts/CubePositions.cs
1	2	Assets/Scripts/MarchingCubes.cs
ad42845f579b59fd1d7b6d445162756cba9eeb17		
37	0	Assets/Scripts/CubePositions.cs
8	8	Assets/Scripts/Grid.cs
1	1	Assets/Scripts/IUpdateScene.cs
3	3	Assets/Scripts/LogicComponent.cs
14	7	Assets/Scripts/MarchingCubes.cs
39	10	Assets/Scripts/UIHandler.cs
6cc10d6025360297d74685a1611430b6ef484e16		
9	1	Assets/Scripts/MarchingCubes.cs
12	10	Assets/Scripts/UIHandler.cs
0ad6a1c81c0e7492500c00b6284ac5b261ded645		
108	0	Assets/Scripts/Grid.cs
8	0	Assets/Scripts/IHandleUserInput.cs
9	0	Assets/Scripts/IUpdateScene.cs
62	0	Assets/Scripts/LogicComponent.cs
23	0	Assets/Scripts/MarchingCubes.cs
170	0	Assets/Scripts/UIHandler.cs
11	0	Assets/Scripts/WaveFunctionCollapse.cs
3481	**2083**	

8.1.2 List of First Commits

The left column contains the added lines, the middle column the deleted lines, and the right column contains the file names. Only .cs files were considered. At the end the added and deleted lines are summed up.

git log -37 --numstat --pretty="%H" *.cs		
507be55d73b247acbafbb93afa97b4f03ba1c031		
6	6	Assets/Scripts/ButtonLogic.cs
3	3	Assets/Scripts/CameraScript.cs
2	0	Assets/Scripts/IHandleUserInput.cs
2	0	Assets/Scripts/IUpdateScene.cs
6	1	Assets/Scripts/LogicComponent.cs
1	5	Assets/Scripts/MarchingCubes.cs
9	4	Assets/Scripts/UIHandler.cs
18	13	Assets/Scripts/WaveFunctionCollapse.cs
c19177463bb55823acf4113e2806f630c27809fb		
1	2	Assets/Scripts/Grid.cs
1	1	Assets/Scripts/LogicComponent.cs
bc92558258cc0d20afacd8a8b2b108914f5f198e		
16	11	Assets/Scripts/CameraScript.cs
086e642c661797b7d16f0b97f09157d1c986dcae		
0	40	Assets/Scripts/Grid.cs
0	1	Assets/Scripts/UIHandler.cs
1	1	Assets/Scripts/WaveFunctionCollapse.cs
34fae2d57e231b0adbcba54a3dc30c44cc4f4e1b		
128	0	Assets/Scripts/ButtonLogic.cs
6f578cefad24aac89e99df23195f3aba757e6f58		
5	5	Assets/Scripts/WaveFunctionCollapse.cs
b80c62a1e17951cd203fd75f09996e08cc657faa		
19	14	Assets/Scripts/LogicComponent.cs
1	2	Assets/Scripts/UIHandler.cs

950d6caefdf0d395cb8de9b96cd729827cfba939		
2	5	Assets/Scripts/UIHandler.cs
6	6	Assets/Scripts/WaveFunctionCollapse.cs
fb9a6ed3fb0f90e858fdc9f196c74ff842960f95		
98	158	Assets/Scripts/UIHandler.cs
2	2	Assets/Scripts/WaveFunctionCollapse.cs
87d88f0eecf58803aa456555bc36a26fa9c011d4		
3	0	Assets/Scripts/UIHandler.cs
2	2	Assets/Scripts/WaveFunctionCollapse.cs
53b909125d3b0bda549810dfcea6cc885621e215		
1	1	Assets/Scripts/UIHandler.cs
4	4	Assets/Scripts/Util.cs
2	2	Assets/Scripts/WaveFunctionCollapse.cs
3de4d6fa3329160b3a4deb1560f793d759895bd4		
0	1	Assets/Scripts/MarchingCubes.cs
1	1	Assets/Scripts/UIHandler.cs
3	3	Assets/Scripts/Util.cs
34	54	Assets/Scripts/WaveFunctionCollapse.cs
2b11ecfecebfe1cc1d2a5c4f8f9c679592b2ebc4		
39	14	Assets/Scripts/WaveFunctionCollapse.cs
f08b92637c0ae120544847e18875febde9d56925		
1	0	Assets/Scripts/MarchingCubes.cs
1	1	Assets/Scripts/UIHandler.cs
3	3	Assets/Scripts/Util.cs
15	20	Assets/Scripts/WaveFunctionCollapse.cs
f82e77c505eda9e320592c862c6763251a600e36		
14	39	Assets/Scripts/WaveFunctionCollapse.cs
95804d0a75c5a8518846cd8999379c6162343cdc		
2	2	Assets/Scripts/UIHandler.cs

2	2	Assets/Scripts/Util.cs
67	26	Assets/Scripts/WaveFunctionCollapse.cs
19677fc6ba4d0ce24ff1a66a8a5c7d964ea891b6		
14	18	Assets/Scripts/UIHandler.cs
68	31	Assets/Scripts/WaveFunctionCollapse.cs
c7fc5cb54f04d434d0c08039e4be029334fe3981		
1	1	Assets/Scripts/Util.cs
14	25	Assets/Scripts/WaveFunctionCollapse.cs
121331465b41dd786dd89c8327033661f6d190a0		
0	15	Assets/Scripts/MarchingCubes.cs
8	50	Assets/Scripts/UIHandler.cs
82	77	Assets/Scripts/Util.cs
cebee0f399b32ed212b5d7126d64efbbcf6edded		
5	0	Assets/Scripts/UIHandler.cs
7	1	Assets/Scripts/Util.cs
ef1bb188a7f0c63a364f946c5d0258417a5fc448		
3	0	Assets/Scripts/UIHandler.cs
10	4	Assets/Scripts/Util.cs
b3411025f1c6bf6c0b9863ac32fd9f5388f1673b		
9	0	Assets/Scripts/UIHandler.cs
13	2	Assets/Scripts/Util.cs
f38df9f0ad6ac344b2e44dc948a2c3b2af7946c1		
15	0	Assets/Scripts/UIHandler.cs
21	0	Assets/Scripts/Util.cs
f1e82c8aa44e0fcbca80a22aa0de67b7ff629b25		
13	0	Assets/Scripts/UIHandler.cs
15	0	Assets/Scripts/Util.cs
a244c64eeb9b3da02c0931159dd6971beb3e71ac		
5	0	Assets/Scripts/UIHandler.cs

8	2	Assets/Scripts/Util.cs
9150018a1e00b0eff6180f536a6abc1d5501a7a4		
13	0	Assets/Scripts/UIHandler.cs
16	0	Assets/Scripts/Util.cs
cdfd945cbb1afbf93666b7197fd7abea9d31dc80		
13	0	Assets/Scripts/UIHandler.cs
18	3	Assets/Scripts/Util.cs
aa8a4bf5c7e9eaea191237a9bae4aa39f3745395		
13	6	Assets/Scripts/UIHandler.cs
15	6	Assets/Scripts/Util.cs
b89ab889adf12bd14aed4b23b058e0cb753e86d0		
4	4	Assets/Scripts/WaveFunctionCollapse.cs
cbf4012a54b924cb1cdea26874232d997ef107a2		
5	0	Assets/Scripts/UIHandler.cs
7	0	Assets/Scripts/Util.cs
d0170f4718b3d8e61a29947cb6d498f03100480d		
5	0	Assets/Scripts/UIHandler.cs
8	0	Assets/Scripts/Util.cs
9a2211832aec28114b9db9fabd84d68c12de60bc		
10	2	Assets/Scripts/LogicComponent.cs
19	0	Assets/Scripts/UIHandler.cs
22	14	Assets/Scripts/Util.cs
f44d899fe6cccfa6209096633fef4274910d4b55		
1	1	Assets/Scripts/LogicComponent.cs
3	0	Assets/Scripts/UIHandler.cs
32	11	Assets/Scripts/Util.cs
10f870e5a102a65ec9443e11aa7dac56971f897c		
5	0	Assets/Scripts/LogicComponent.cs
8	4	Assets/Scripts/UIHandler.cs

44	197	Assets/Scripts/Util.cs
9d066a3c6be021ea14bdc400d729dea9f203a79a		
38	10	Assets/Scripts/UIHandler.cs
157	57	Assets/Scripts/Util.cs
ce54ed8df87917c795d43b396e2ee311cd4232a8		
2	1	Assets/Scripts/UIHandler.cs
0b7266f595883f9e0849f53c50133dba3a9bfb41		
45	112	Assets/Scripts/UIHandler.cs
0	10	Assets/Scripts/Util.cs
1335	**1119**	

8.1.3 List of Latest Commits

The left column contains the added lines, the middle column the deleted lines, and the right column contains the file names. Only .cs files were considered. At the end, the added and deleted lines are summed up.

Other 37 commits (first list minus second list):		
1	1	Assets/Scripts/MarchingCubes.cs
11	0	Assets/Scripts/Util.cs
1f7dabb93b43e7064f193000fc1b95ad9c06c944		
34	0	Assets/Scripts/UIHandler.cs
77	34	Assets/Scripts/Util.cs
f962d107ffc078ff450db4f7b9928efc381ddf66		
29	4	Assets/Scripts/UIHandler.cs
4f037c2da882665d303ab355df118dd3419c6312		
0	68	Assets/Scripts/CubePositions.cs
17	24	Assets/Scripts/MarchingCubes.cs
12	11	Assets/Scripts/UIHandler.cs
129	9	Assets/Scripts/Util.cs

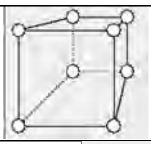
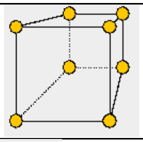
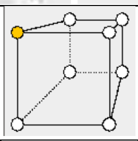
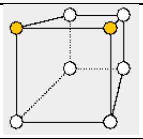
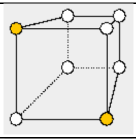
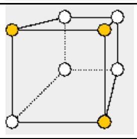
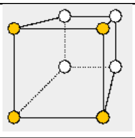
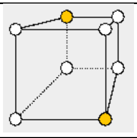
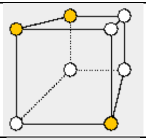
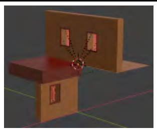
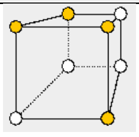
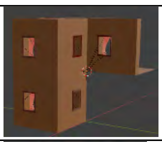
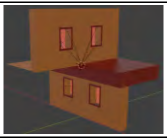
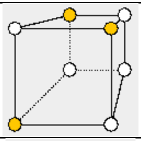
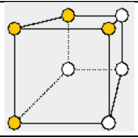
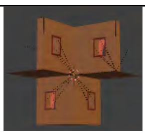
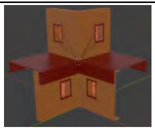
1	1	Assets/Scripts/WaveFunctionCollapse.cs
3546ac1310a380a67e4612a0c3738f2c44f74ebe		
1	1	Assets/Scripts/LogicComponent.cs
17	26	Assets/Scripts/UIHandler.cs
6	1	Assets/Scripts/{SharedItems.cs => Util.cs}
80	24	Assets/Scripts/WaveFunctionCollapse.cs
6204ef4d44968c275323e2c72d75f322bb0e269d		
140	55	Assets/Scripts/WaveFunctionCollapse.cs
99a5b7703e86395c245e63c4be448d8ed0a7a777		
40	0	Assets/Scripts/CameraScript.cs
81	29	Assets/Scripts/WaveFunctionCollapse.cs
0aa7c2696b5dca9863432b7a40f52d51531b987f		
1	23	Assets/Scripts/Grid.cs
1	1	Assets/Scripts/IHandleUserInput.cs
8	16	Assets/Scripts/LogicComponent.cs
15	4	Assets/Scripts/UIHandler.cs
b9752af4e55c53d45eb58432fa415d922c758709		
18	0	Assets/Scripts/SharedItems.cs
8	6	Assets/Scripts/UIHandler.cs
1	4	Assets/Scripts/WaveFunctionCollapse.cs
eee72197d031bee4a97d6b4afb33dd3b6131a800		
24	62	Assets/Scripts/WaveFunctionCollapse.cs
89cbb9177d87df901f3b25ce4e326ca0721ac1fa		
9	7	Assets/Scripts/UIHandler.cs
66	11	Assets/Scripts/WaveFunctionCollapse.cs
ab426c1dc4096d2daed64b3686f211fb0bf1fd24		
24	30	Assets/Scripts/WaveFunctionCollapse.cs
710d45398bc2bd671495dd7fd918c8ee573fe683		
12	7	Assets/Scripts/UIHandler.cs

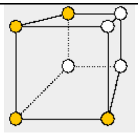
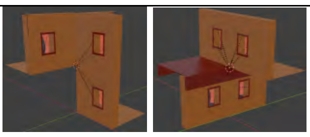
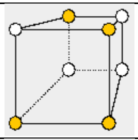
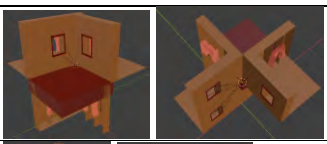
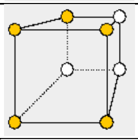
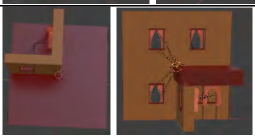
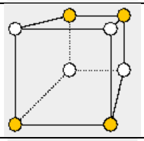
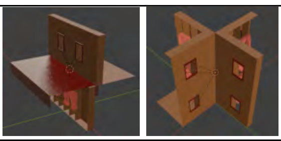
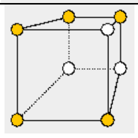
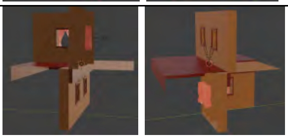
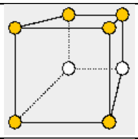
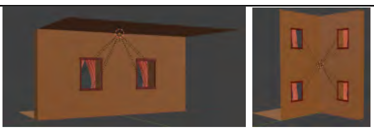
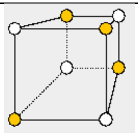
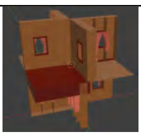
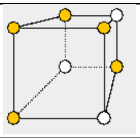
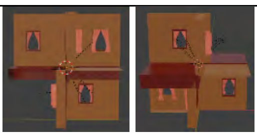
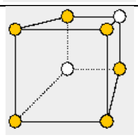
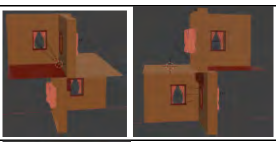
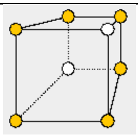
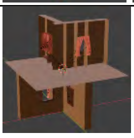
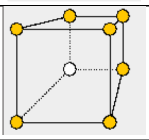
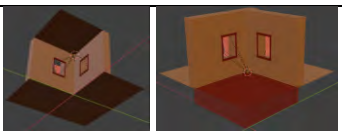
70	52	Assets/Scripts/WaveFunctionCollapse.cs
6b0fa6df7810f0338b99715b1bcd4a85dca40f07		
73	10	Assets/Scripts/WaveFunctionCollapse.cs
d1577f50291cb9964436e9c00d7756e80c64dff3		
4	1	Assets/Scripts/CubePositions.cs
2	0	Assets/Scripts/MarchingCubes.cs
35	12	Assets/Scripts/UIHandler.cs
140	2	Assets/Scripts/WaveFunctionCollapse.cs
3dd4697749a0c663de8968340fa94360c3284b25		
11	3	Assets/Scripts/CubePositions.cs
4	1	Assets/Scripts/MarchingCubes.cs
17	0	Assets/Scripts/UIHandler.cs
190310ca81c6814a1253de03ae4aaa7a379ab945		
0	2	Assets/Scripts/UIHandler.cs
2538728f0fab2244c6006a4bb9f8a79169741645		
41	26	Assets/Scripts/UIHandler.cs
1f7921e1a004f0e007b7c28700c4f2aa3637e0bd		
41	97	Assets/Scripts/UIHandler.cs
0b33e05fad7b1c08dff6ad166818da04a742443a		
10	15	Assets/Scripts/UIHandler.cs
f6ebfc3253322b1ec259439ccd4061a42c8285e3		
2	0	Assets/Scripts/CubePositions.cs
86	93	Assets/Scripts/UIHandler.cs
ba0da5f833bb86c3261d9e733d853681de6c460b		
2	1	Assets/Scripts/CubePositions.cs
9	1	Assets/Scripts/MarchingCubes.cs
39	5	Assets/Scripts/UIHandler.cs
9233a1925badffeac59aff8285f0620217e6121a		

1	1	Assets/Scripts/CubePositions.cs
1	1	Assets/Scripts/IUpdateScene.cs
20	13	Assets/Scripts/LogicComponent.cs
136	125	Assets/Scripts/UIHandler.cs
5	0	Assets/Scripts/WaveFunctionCollapse.cs
982f7c12757a243c254954df2eb70084851aa648		
19	2	Assets/Scripts/CubePositions.cs
1	2	Assets/Scripts/MarchingCubes.cs
ad42845f579b59fd1d7b6d445162756cba9eeb17		
37	0	Assets/Scripts/CubePositions.cs
8	8	Assets/Scripts/Grid.cs
1	1	Assets/Scripts/IUpdateScene.cs
3	3	Assets/Scripts/LogicComponent.cs
14	7	Assets/Scripts/MarchingCubes.cs
39	10	Assets/Scripts/UIHandler.cs
6cc10d6025360297d74685a1611430b6ef484e16		
9	1	Assets/Scripts/MarchingCubes.cs
12	10	Assets/Scripts/UIHandler.cs
0ad6a1c81c0e7492500c00b6284ac5b261ded645		
108	0	Assets/Scripts/Grid.cs
8	0	Assets/Scripts/IHandleUserInput.cs
9	0	Assets/Scripts/IUpdateScene.cs
62	0	Assets/Scripts/LogicComponent.cs
23	0	Assets/Scripts/MarchingCubes.cs
170	0	Assets/Scripts/UIHandler.cs
11	0	Assets/Scripts/WaveFunctionCollapse.cs
2146	**964**	

8.2 A2: Basic Equivalence Classes

Equivalence Class	**Basic Node State**	**3D Tiles (examples from different sub-classes)**

0	 	/
1		
2		
3		
4		
5		
6		
7		
8		 
9		
10		 

11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ___________________________________

Vorname: ___________________________________

dass ich die vorliegende Bachelorarbeit mit dem Thema:

Procedural Generation of Buildings with Wave Function Collapse and Marching Cubes

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_______________	_______________	________________________________
Ort	Datum	Unterschrift im Original