FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Procedural Generation of 3d Buildings and Interiors

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Visual Computing

by

Franz Richard Spitaler BSc.
Registration Number 0226436

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass.(FWF) Dr.techn. Dipl.-Mediensys.wiss. Przemyslaw Musialski

Vienna, 18th August, 2015

Franz Richard Spitaler

Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn.

Erklärung zur Verfassung der Arbeit

Franz Richard Spitaler BSc.
Address

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. August 2015

Franz Richard Spitaler

Acknowledgements

I want to thank everyone that made it possible for me to finish my studies. Especially I want to thank my girlfriend for always supporting me and never stop believing that an end exists!

Abstract

In this master thesis I describe a procedural system, which makes it possible to generate 3-dimensional buildings including their interior. I investigate previous work in the field of the procedural generation of content as well as the more specialized area of the procedural generation of buildings and façades. From the work that was already published, I identify useful concepts and unify them in this work by merging the geometry-generation techniques with the possibilities of a visual rule-editor, where no 'code' has to be written to generate a complex procedural system.

With the help of some so-called production rules, which I describe in the thesis, it is possible to create a wide variety of different buildings. By introducing two specialized production rules, the creation of the complex geometry of building-related elements like roofs or stairs is more comfortable and easy.

While the main focus of this thesis is the development of the procedural system itself, there are quite a few different other scientific domains that have an influence in this work. The thesis combines procedural systems with real-time computer graphics, floor planning, architecture, and user-interface design.

Contents

Abstract **iv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Definition . . . 3
1.3 Aim of the Work . . . 3

2 Related Work **6**
2.1 Procedural Systems . . . 7
2.2 Content Generation . . . 11
2.3 Generating façades of Buildings . . . 12
2.4 The Generation of Buildings and Cities . . . 14
2.5 Floor Layout Generation . . . 17
2.6 Furniture Placement . . . 19

3 Methodology and Approach **20**
3.1 Application Overview . . . 20
3.2 The Building . . . 27
3.3 How the Rules Work . . . 34
3.4 The Specialized Rules . . . 34
3.5 The Vertical Connector / Stairs Rule . . . 35
3.6 The Roof Rule . . . 43
3.7 Integrated algorithms . . . 47

4 Implementation **58**
4.1 Application Organization . . . 59
4.2 Data Formats . . . 64
4.3 The User Interface . . . 69
4.4 The 3-Dimensional View . . . 71
4.5 Graphs . . . 71
4.6 Step by Step Building Generation . . . 71
4.7 The MVVM Pattern Implementation . . . 73
4.8 Organization of the Rule-Classes . . . 75

4.9 Helix Toolkit . . . 77
4.10 Shape Implementation . . . 77
4.11 The Implemented Rules . . . 78
4.12 How to Add a Rule . . . 83
4.13 Important Variables and Fixed Values . . . 84
4.14 Limitations of the Created Procedural System . . . 84

5 Results **86**

6 Conclusion and Future Work **90**
6.1 Synopsis . . . 90
6.2 Conclusion . . . 91
6.3 Future Work . . . 92

Appendices **93**

A Definitions **94**
A.1 List of Terms . . . 94

B Pseudo Codes **97**
B.1 Example Code for the ”CityEngine“ . . . 97
B.2 The Floor-Planning Algorithm . . . 97
B.3 The Straight Skeleton Algorithm . . . 98

Bibliography **102**

List of Figures

2.1 L-System . . . 8
2.2 Dragon curve . . . 10
2.3 Quadratic Koch island . . . 10
2.4 Example procedural building . . . 15

3.1 MVVM design pattern . . . 21
3.2 Default and modified roof and stairs . . . 23
3.3 Visual rule editor . . . 24
3.4 Hierarchy of rooms example . . . 29
3.5 Complex hierarchy of rooms example . . . 30
3.6 Different room definitions . . . 30

3.7 Elements of stairs . . . 35
3.8 The individual stairs types . . . 36
3.9 Elevator example with doors . . . 37
3.10 Visualization of additional space requirements for stairs . . . 39
3.11 Space requirement of stairs for different floor-heights . . . 41
3.12 Stairs rule with automatically added rules . . . 42
3.13 Roof rule, wall added height . . . 45
3.14 Complex hipped roof from above . . . 46
3.15 Saddle roof example . . . 46
3.16 Comparison different roof types . . . 48
3.17 Roof definition using random values . . . 48
3.18 Floor planning steps visualized . . . 52
3.19 Complex roof geometry, purlins and rafters . . . 54
3.20 Different slopes affect the direction of roof ridges . . . 55

4.1 ViewModel Hiearchy . . . 64
4.2 Screen-shot with loaded property . . . 72
4.3 Different façade styles . . . 73
4.4 Hierarchy of the implemented rule classes . . . 75
4.5 Visualization of classes that interact with rule classes . . . 76
4.6 Path-rule, effect of stretching the cross-section . . . 82

5.1 Result with multiple vertical connections . . . 86
5.2 Result with many balconies . . . 87
5.3 Result with textured building and two façade definitions . . . 87
5.4 Faked furniture in rooms . . . 88
5.5 Result with stairs and reflective elements . . . 88
5.6 Result with some complex doors . . . 89
5.7 Result with a complex definition of a spiral stairs . . . 89

List of Algorithms

B.1 Part of the code of a CityEngine example scene . . . 99
B.2 Floor-layout algorithm . . . 100
B.3 Straight skeleton algorithm . . . 101

CHAPTER 1

Introduction

Computers nowadays are often used to create different kinds of digital content. The type of content ranges from simple text contents to digital paintings to sound- and video-productions and to 3-dimensional content. For each type of content there exist specialized tools that make the process of creation of these contents as easy and fast as possible.

Traditional approaches to create 3-dimensional content involve the creation of the desired objects or characters by modeling them manually. This modeling approach is very time-consuming and the designers and artists have to create every detail about the resulting 3d objects manually. The more detailed each object should be, the more work has to be put into the creation process and the more time is needed to create the desired objects.

1.1 Motivation

The digital content generation of 3-dimensional objects is addressed in this work because the creation of this type of content is very time-consuming and results in high investments from the industry. By using a different approach for the generation of 3-dimensional objects it is possible to enhance and to speed up the creation-process a lot. The approach for creating the objects is called **procedural content generation**.

The procedural generation of content is a very powerful technique to create many different kinds of objects. The idea behind the generation of objects procedurally is that a user needs just a relatively small set of so-called production-rules or 'rules' that define the procedural system. The procedural system consists of the set of defined rules, i.e. the rule-set. With such a procedural system, it is possible to create many different types of objects. Short descriptions of a rule A.1.1) and a rule-set A.1.1 can be found in the Appendix.

The possibilities for generating and creating objects with a procedural system range from the creation of fractals and 2-dimensional graphics [PL90, 6-18, 46-50, 209] and art [Pea11] to 3-dimensional objects like plants [PL90] and buildings [WWSR03, MWH+06, KW11]. Even the whole structure of a planet can be generated by using procedural-generation techniques [Cep10]. Many different cases can be handled and many different and realistic results are possible by using randomly altered values in the generation process and by the possibility of the rules to adapt to the environmental circumstances. Examples are "self-sensitive L-Systems" [PM01], which adapt to geometric circumstances or simulated chemical reactions [PL90, 40, 41]. The "related work"-section 2 on page 6 gives a short overview.

In the proposed master thesis, the focus lies on the procedural generation of buildings. Some quite powerful solutions already exist, the most well-known application using procedural generation techniques is the "CityEngine" [PM01] [MWH+06], but the abilities of those solutions are limited. The CGA-grammar[1] is a complex extension of L-systems[2], which allows the user to create complex 3-dimensional objects. Applications like the CityEngine use the rules defined in the CGA-grammar to create buildings and whole cities from GIS[3]-data or defined landscapes, but to achieve this, the user has to write a lot of code to create the buildings of the city. A short example of the code needed to create a simple scene can be found in the Appendix B.1.

The user has to write some kind of a program to create the procedural system in applications like the CityEngine, which is a major drawback of existing solutions. Many users in creative industries do not know how to write a program, which is a big problem for procedural generation tools. By removing this critical requirement, almost anybody can use these powerful tools. In the master thesis at hand a visual "rule-editor" is described and implemented. It allows the creation of rules in a visual manner and makes it possible to create the complete procedural system without coding.

Using procedural generation techniques, combined with a visual rule-editor to define the production-rules and to create detailed buildings enables us to create complex scenes in a shorter time than before. The users don't have to use traditional modeling techniques and do not need to write any kind of code to create the procedural system. The use of these techniques facilitates the creation of the needed rule-sets, i.e. the procedural systems, described in section 2.4.

[1] **C**omputer-**G**enerated-**A**rchitecture - grammar
[2] Lindenmayer - systems
[3] **G**eographic **I**nformation **S**ystem

The generated buildings can be used for illustration purposes, in the movie industry as well as in architectural fields. Another possible application for such a system is the computer game industry, where a lot of content needs to be created to build the levels and environments of the game. This creation of the content is very time-consuming and therefore expensive if the content is created manually. Reducing the amount of the time it takes to create those environments leads to sceneries, which can be created with less investment of time and therefore money. Even really big environments with many different buildings of different styles, sizes and levels of detail can be generated by such a procedural generation system.

1.2 Problem Definition

The task of creating realistic looking buildings is very complex and not solved yet. In this master thesis I try to improve already existing solutions by combining a "no-code" - visual rule-editor with the advantages of procedural generators. The generation of buildings by defining a procedural rule-set visually is one of the main aspect I want to work on.

To achieve a realistic look of the buildings, both the exterior as well as the interior, i.e. the rooms of the building, have to be generated. The rooms have to be placed in a realistic manner to create a plausible result. The positioning of the defined rooms is an active field of research and no perfect solution was found yet. The application should to be able to create floor plans which look plausible and may be defined in a flexible manner.

The positioning and generation of the floor-connecting elements, i.e. the elevators or stairs, is another problem that is worked on in the master thesis at hand. Since not only the exterior, but also the interior of the buildings has to be generated, there exists the need for connections between the floors in the buildings.

By having to write a lot of code to generate buildings and whole cities [PM01,CEW$^+$08] the procedural generation of buildings by using the available applications, is not an easy task for most people because at least a basic knowledge in programming is required. This is a big problem because the number of lines of code that has to be written explodes, when more details have to be added to the buildings.

1.3 Aim of the Work

The purpose of this master thesis is to develop a usable application to create 3-dimensional buildings. I try to improve the solutions for all the previously mentioned problems, which exist in current solutions. The most important part is to be able to generate 3-dimensional buildings with the implemented application by implementing a procedural generator.

This generator has to be as flexible as possible and it should be possible to add more different rules to the system later. The application should be able to work with any input-scene, i.e. property - file, which is loaded into the application.

The aim of the master thesis is to create a tool, which uses a procedural generation system to generate buildings. The generated buildings may be very detailed if the rules for all the desired building-details are created in the visual rule-editor. The buildings can have different styles, like residential buildings, office buildings or contemporary luxury houses, etc. . A big advantage of using a procedural system is that there is no limit on the level of detail of the generated buildings. The implemented tool should be able to allow the creation of those details if they are wanted in the resulting building. By defining the production rules in a hierarchical manner it is possible to generate every part of a building, therefore it should be possible to just generate the simple hull of a building or detailed façade-elements, rooms, doors, etc. if that level of detail is desired.

Another aim of the proposed master thesis is the possibility not only to generate the 3-dimensional geometry for the façades, but to actually generate the individual rooms inside each floor of the building. The problem of planning and distributing the rooms inside the floors of the buildings is very complex and a lot of different approaches to solve the problem exist. I want to implement an algorithm that is capable of subdividing the available space of a floor into the individual rooms realistically.

The positioning of the vertical-connectors should be handled in the implemented procedural-system. The assumption that every room in a floor has to be accessible is made. This means, there has to be at least one room-connection to another room for all of the rooms in a floor. This assumption makes the positioning of the vertical-connectors more difficult because the areas of valid positions of the vertical connectors are reduced to ensure each room-connection is actually reachable and not positioned behind the vertical connector. I want to create a positioning-system for the vertical connectors, which ensures the room-connectivity and also handles small rooms containing vertical connectors by increasing their size until a valid position is found. Not all rooms are allowed to be connected to each other, even if they are adjacent because the hierarchical definition of the room-structure implicitly defines the connectivity of the rooms. In section 3.2.1 more information about why this is necessary, wanted and very useful, can be found.

The application works without writing a single line of code, there is not even the possibility to do so. It is one goal of this master thesis to show that the previously very complex task to write a procedural system can be substituted by creating rule nodes and connecting them via connectors in a visual-rule-editor. To be able to create the procedural system and all the CGA-rules that are needed to create the buildings, I implement a visual rule-editor, which is based on the work [Dav12]. It is possible to create and connect the rule-nodes, which are the visual representation of the CGA-rules of the procedural-system. The behavior of every rule in the procedural system can be changed as well.

CHAPTER 2

Related Work

Before describing the details of how the application works, which assumptions were made during the development and how the application was implemented, I shortly want to discuss some previously published papers that have influenced the master thesis at hand. I begin with related works regarding procedural systems themselves and their development with a few examples. A few works about fractals and a short description about how they relate to the L-Systems and the procedural system I implemented in the application follows. Some examples about the procedural nature of many objects is shown and the benefits of using these techniques are outlined.

From the discussed works that form the foundation of this work, I further investigate work that covers the creation of procedural content itself. Papers that describe how it is possible to use procedural generation methods for the creation of buildings, façades and room-layouts are discussed. Since there are a lot of different approaches that cover the creation of façades of buildings I will shortly discuss a few of them. One of the most problematic and open topics of the procedural generation of buildings is the floor-planning of the buildings. I will give an overview of some simple ways to generate the floor-plans of buildings and briefly describe the benefits and drawbacks of my solution.

The generation of cities and street-networks is another very important related topic. I show how the L-Systems can be extended so that it is possible to create street-networks for cities. With a street-network alone, it is not possible to create a realistic city because there are many different looking buildings in the city that serve a different purpose. It is shown how it is possible to distribute all the different buildings that occur in a city realistically.

2.1 Procedural Systems

In the master thesis at hand I implemented an application that works by using a procedural generation system. L-Systems are procedural generation systems and were developed by the biologist Aristid Lindenmayer and the computer scientist Przemyslaw Prusinkiewicz in the work "The Algorithmic Beauty of Plants" in 1990 [PL90]. The motivation for the development of the L-Systems was to be able to simulate and visualize the growth of different plants.

L-Systems are essentially parallel re-writing systems. An L-System is defined by a Tuple of some elements, the complete alphabet, the axiom, and the set of production rules. The L-Systems are strings of characters and they can easily be interpreted e.g by using a so-called "turtle graphics".

A simple L-System is shown in equation 2.1 below.

$$\begin{aligned} &n : 5 \\ &\delta : 20^\circ \\ &\omega : F \\ &p_1 : F \rightarrow F[+F]F[-F][F] \end{aligned} \tag{2.1}$$

The meaning of the equation above is described in the following paragraphs. The value of n defines the number of iterations for the procedural generator. This means that in this case the string defining the L-System is processed five times. The rewriting, i.e. the iterative processing, of the L-System is started by using the defined axiom ω, which is only the symbol F in this case. The axiom can be a lot more complex in other examples.

In the first step, the symbol F is replaced by the right hand-side of the first fitting "production-rule", which is the rule p_1 in the example. The result of the first "rewriting" of the symbol F is $F[+F]F[-F][F]$. After the second iteration, the L-System looks like this: $F[+F]F[-F][F][+F[+F]F[-F][F]]F[+F]F[-F][F][-F[+F]F[-F][F]][F[+F]F[-F][F]]$. Even in this simple example that final result is a long string of characters which can be interpreted graphically using a turtle graphics. The value of δ defines the angle that is used for the rotation of the turtle whenever the interpreter encounters one of the symbols $-$ or $+$. The interpretation of the two characters turn the turtle to the left or right.

Figure 2.1 displays the result of the L-System above. The figure shows a complex representation of a plant which probably would take a lot of time to draw manually.

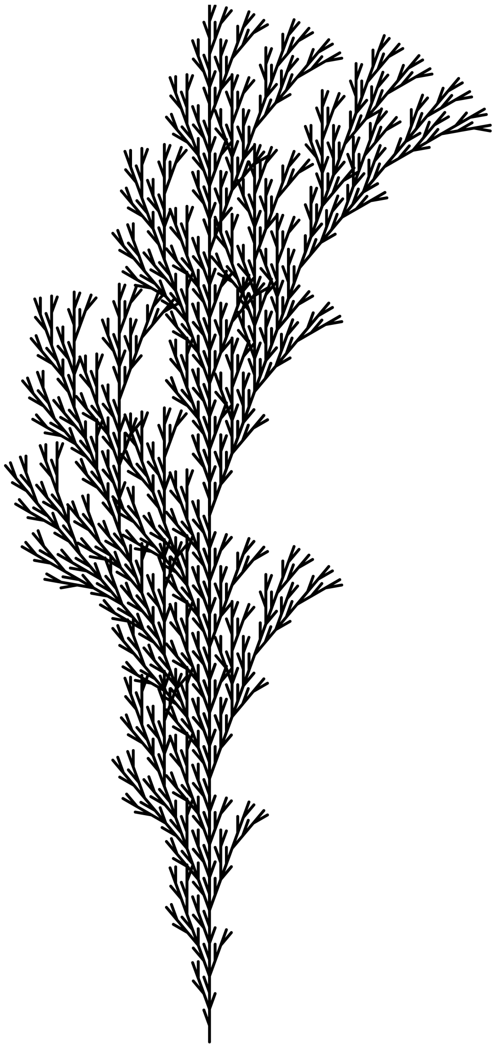

Figure 2.1: The resulting graphical interpretation of the L-System defined in equation (2.1) after five iterations. The visualization is created by the mentioned interpretation of the symbols of the resulting L-System using turtle-graphics.

The example from equation (2.1) and it's visualization in figure 2.1 demonstrates how simple it is to generate a complex structure with just one production-rule. It is a so-called deterministic and context-free L-System, which means that the result is always exactly the same, no matter how often the process of the generation is started with the same variables. Possibilities to randomize the resulting L-System exist though. In the work by Lindenmayer and Prusinkiewicz [PL90] stochastic- and context-sensitive L-Systems were introduced. Stochastic L-Systems use an assigned probability-value for every production-rule. It is possible to randomly select a fitting production rule out of the set all fitting rules that way. This means that there may be more than one production-rule applicable for the replacement of a symbol in a stochastic L-System. For simple L-Systems there may be only exactly one production-rule present in the L-System. The equation (2.1) demonstrates a simple L-System. Context-sensitive L-Systems were introduced to simulate the propagation of nutrients through plants for example. A context-sensitive L-System produces more realistic results because not only the local circumstances are taken into account when one iteration step is performed but also its neighborhood. It is possible to apply a rule only if there are enough nutrients present for example. A context-sensitive production-rule might look like equation 2.2.

$$a < F > b \rightarrow F[F] \tag{2.2}$$

The definition of the context-sensitive production-rule means that the symbol F is replaced by $F[F]$ if and only if it is directly preceded by the symbol a and directly followed by the symbol b.

Equation (2.1) in this section already gives a hint about the possibilities that arise with the use of L-Systems. There are a few more extensions presented in [PL90] like "parametric L-Systems" and extensions for the 3-dimensional interpretation of the resulting string. Parametric L-Systems enable us to define variables and use calculations to modify the variables and thus the resulting system. Another possibility is to use conditions for the production-rules. This means that an optional condition is checked before the selection of a production-rule is performed. If the test is successful the production-rule may be selected, but if the condition is not satisfied the production-rule cannot be applied and another rule must be selected. It is possible e.g. to reduce the value of a variable that represents the amount of nutrients in each iteration to simulate the aging of the plant or to restrict the growth of the plant to realistic measures e.g. by using the mentioned conditions.

Symbol-replacement - systems are not limited to create plants, but they are usable to create some famous fractals like the dragon curve which is visualized in figure 2.2 or a quadratic Koch island which is visualized in figure 2.3.

Aside from the possibility to create plants many additional applications for the use of the L-Systems are outlined in the work "L-systems and Beyond" by Prusinkiewicz et.al. [FKMP03]. Possibilities to use L-Systems to solve partial differential equations are shown for example [FKMP03, 2-39]. Context-sensitive and parametric L-Systems are used to perform the needed calculations and operations, which lead to the solution that is then visualized.

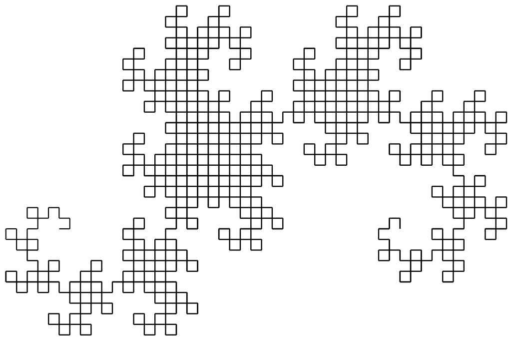

Figure 2.2: Visualization of a dragon curve after a few iterations of the L-System (taken from [PL90]).

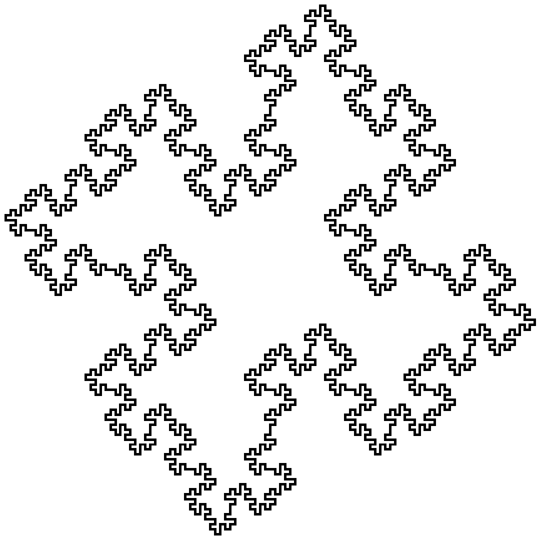

Figure 2.3: Visualization of a quadratic Koch island. Produced with an L-System (taken from [PL90]).

Another interesting application of L-Systems is the subdivision of curves [FKMP03, 3-1] and 3-dimensional meshes [FKMP03, 3-31]. An example shows how it is easily possible to create a subdivision scheme that results in the same subdivision results like the "Chaikin's Algorithm" [Cha74]. The algorithm subdivides the line segments that are connected to "inner" points of the surface meshes. Since Chaikin's Algorithm works iteratively, it is a good candidate for the realization and calculation with an L-System. In each iteration-step each inner point of the curve or mesh is replaced by two new points. The positions of the two new points are calculated by a simple formula which can easily be expressed using an L-System. The formula is shown in equation 2.3.

$$P(v_l) < P(v) > P(v_r) \rightarrow P(\frac{1}{4} \times v_l + \frac{3}{4} \times v)P(\frac{3}{4} \times v + \frac{1}{4} \times v_r) \tag{2.3}$$

The meaning of the production-rule in equation 2.3 is that for every point $P(v)$ both neighboring points are also taken into account for the calculation of the two new points that replace the point $P(v)$ in the next iteration step. The production-rule is context-sensitive and is therefore not only applicable to closed curves, but also to open ones. The production-rule is not applicable and therefore not executed for the two endpoints of the curve because those points have a different context. The position of the points are represented by the variables v_l, v and v_r and the two newly generated points are calculated according to the Chaikin's Algorithm.

2.2 Content Generation

L-Systems provide a great way to describe natural organisms like plants in a simple way and with these L-Systems it is possible to create a wide variety of plants and flowers, where each individual object may be different in size and shape compared to all the other generated ones in a scene, even if they are created using the same production-rules. Procedural systems provide enough flexibility to create almost any type of object that needs to be placed in a scenery and it is even possible to create the complete content of a scenery procedurally, including all background objects like landscapes, floors, etc. .

While it is possible to use procedural systems to perform a wide range of different tasks like the use for simulations, the main purpose of using them is the creation of content. I want to introduce a few works that describe the possibilities of how to use the procedural system for content creation purposes. Many industries benefit from being able to generate content in a fast and easy way with procedural systems and a lot of work was already done to research the abilities of dynamic content creation for e.g. the use in computer games and in the movie industry.

The generation of game-levels is a complex and difficult task for example. I want to give a small overview over two of the many published papers that propose methods and ideas of how to generate game-levels procedurally. The first work I want to discuss briefly is a work by Andrew Doull "The Death of the Level Designer" [Dou]. In the series of blog posts he mentions many different aspects and possibilities of how to use procedural content creation for game development. The possibilities to create game-content procedurally range from the creation of the game-levels e.g. for games like Diablo, Hellgate London and Minecraft to almost all other types of content of a computer game. In [Dou, 4] is it shown that it is possible to even dynamically create faces for all occurring characters in a game-world. The game "Eve-Online" uses procedural generators to create unique faces for the characters for example because it is important to be able to distinguish individual persons easily in this game. Another possibility is the creation of big amounts of different assets that are placed inside the levels of the games. A good example for a tool that generates trees is the ”Speedtree“ [Spe] tool mentioned in [Dou, 4]. It is used to generate flowers and trees in games but it is also used in the movie industry.

AI[1] is also considered to be procedural content because the individual characters are behaving differently, i.e. a randomly selected behavior from all the possible reactions to an event is selected for each character. Other mentioned examples for procedural content that can be created and used in games are the sound-effects, which are used in the game ”Spore“ for example, the changing weather situations and also puzzles and weapons.

In the published article ”A Generic Approach to Challenge Modeling for the Procedural Creation of Video Game Levels“ by Sorenson et. al. [SPD11] the focus solely lies on the automatic creation of levels for computer games. The authors use a top-down approach for the modeling of the game-levels by using a ”fun“-value that is calculated by taking several aspects of the generated game-level into account. This function is then used with an evolutionary algorithm to alter the created game-level. A game-level is considered to be an ”individual“. By using the defined fitness-function, i.e. the fun-value, the level with the best fun-value is selected.

As mentioned before, a lot of games [Uni, Per, Sta, Eve, Hel] that use some kind of procedural content generation - technique already have been developed. Aside from the development of games, a lot of tools for the authoring and creation of 3-dimensional scenes which use procedural techniques have been developed [Spe, Reg].

2.3 Generating façades of Buildings

With the development of the CGA-grammar in the work ”Procedural Modeling of Buildings“ by Pascal Müller et. al. [MWH+06] it became possible to easily create 3-dimensional objects by the use of production-rules. The CGA-grammar is an extension of L-Systems for 3-dimensional modeling purposes and will be discussed in more detail in the section 2.4.

A specialization of the generation of 3-dimensional content is the generation of the façades of buildings. Existing applications typically just produce a realistic-looking hull for the buildings that are generated because this level of detail is enough for many use cases of the generated buildings.

[1] **A**rtificial **I**ntelligence

Some papers that try to easily the problem of façade-creation based on existing images have been published in the past. The first paper I want to mention is titled "Image-based Procedural Modeling of Facades" by Pascal Müller et. al from the year 2007 and uses pictures of façades to create a 3-dimensional representation of it. The different elements of a façade are created by subdividing the façade into smaller elements in several steps.

The subdivisions are determined automatically and a resulting rule-set is generated from the input image. This approach leads to the creation of 3-dimensional façades which adapt to different environmental circumstances. It is possible to use the derived rule-set that creates a façade with other geometric measures for example. It is possible to generate façades for a building that contains more or less floors, than the building in the input-image. The derived production-rules automatically adapt to the desired amount of floors.

Another work that focuses on the semi-automatic creation of high-quality façades of buildings is "Image-based Façade Modeling" by Jianxiong Xiao et. al. [XFT+08]. A sequence of images of a façade is used to determine the shapes and different depths for the individual resulting elements. A complete texture of the façade is reconstructed from the different input-images at first and by detecting the horizontal and vertical lines in the texture, a façade decomposition is performed to create the shapes of the façade. By using the sequence of input-images it is possible to calculate a point-cloud for some prominent façade-points and by using a "Markov Random Field" the depth of the façade-elements is estimated [XFT+08, 6-7].

The article "Grammar-based Encoding of Façades" by Haegler et. al. [HWM+10] introduces another grammar, i.e. a different set of possible production-rules, to model the façades. The F(açade)-shade grammar is introduced to provide a simple and compact solution to be able to navigate and render very big scenes like the "Munich-scene" with approx. 55Mio. triangles. By using texture-atlases it is possible to reduce the amount of the memory needed to store the information for all the textures in the scene. One assumption made in the work is that a lot of similar elements of the façades occur in the city and on a building. This means that the window-elements of each floor may be represented by only one shape if they are similar [HWM+10, 1, 4].

Another paper that describes a semi-automatic creation of high-quality façades of buildings from input-images is the work "Interactive Coherence-Based Façade Modeling" by Musialski et. al. [MWW12]. The paper focuses on the creation of high-quality 3-dimensional models of façades. Compared to other modeling-tools, this work shows that it is a better approach when 'the human designer is in control of the modeling workflow' [MWW12, 2]. Automatic splitting-operations are implemented and the individual positions of the splits are determined by a coherence-based decision process. Another contribution of this work is the possibility to interactively modify a whole set of shapes at once. The grouping of similar shapes is calculated automatically and e.g. split-operations can be used on the whole set of the grouped shapes in one step.

The work ”Structure Completion for Facade Layouts“ [FMLW14] by Fan et. al. published in 2014 describes an approach for improving the quality of the captured façade-images. Occluded image-parts are replaced by other parts of the captured image, so that the resulting reconstructed façade-image represents the captured façade as good as possible with the incomplete data available through the input images. By using image-reconstruction techniques it is possible to create obstacle-free façade-images that can then be used to actually model the façade as described in the mentioned works and papers above.

The article ”Layer-Based Procedural Design of Facades“ [IMAW15] published in 2015 by Ilcik et. al. proposes the use of multiple layers for façade-modeling. It is shown how irregular façades can easily be created by using more than just one layer of different rule-sets. The layers may be applied only to a sub-region of the desired complete façade and are merged together to one 3-dimensional façade at the end.

2.4 The Generation of Buildings and Cities

The previously mentioned L-Systems form the foundation of the procedural generation of content and of buildings. The generation is based on rules that define what happens to previously generated intermediate elements and shapes. The work ”Instant Architecture“ by Wonka et. al. [WWSR03] describes the use of a split-base shape grammar to model the buildings for a city-scene. The previously mentioned CGA-grammar extends the ”production-rules“ of L-Systems to geometric operations and was introduced in the work [MWH+06] and are used for the purpose of creating the façades of buildings. A big set of operations were developed and are used to position and to modify the generated 3-dimensional elements. The most important and most used rules are the transformation-rules, which move, rotate or scale the individual elements, i.e. the generated geometry, as well as more sophisticated rules like the ”split-rule“, which is used to create many individual shapes from one base-shape. Some more rules are described in detail in the paper [MWH+06].

In the work published in the year 2008 by Lipp et. al. [LWW08] an approach to avoid the need to create the procedural systems to generate buildings manually is described. A few ways of interactive editing of the rules in a 3-dimensional view of the created buildings are developed. The possibility to create and edit the production-rules visually enables many more people to use the application. The work describes a system, where no production-rule has to be written by hand, but is instead automatically generated by the user-input.

Tom Kelly and Peter Wonka developed the paper ”Interactive Architectural Modeling with Procedural Extrusions“ in 2011 [KW11]. It focuses on the modeling of exteriors of buildings and it is also not necessary to manually write the code of the needed production-rules. It is possible to visually define ”profiles“ for the different sides of the buildings. A lot of different styles of buildings, like temples, residential buildings, etc. can be created with this technique. Not all sides of the buildings need to be defined by the same profile, but different profiles can be defined for the different sides, resulting in complex and realistic buildings. An example of a generated building by using this technique can be seen in figure 2.4.

Figure 2.4: A building that was generated by the use of the procedural modeling system introduced in [KW11]. The image is taken from [KW11, 1].

All previously described papers in this section are related to procedural building and façade generation, but the following works focus on the creation of complete cities and enable us to bring the content generation to a bigger scale. I will first introduce papers that describe how it is possible to generate the street-networks of cities and how the different buildings are placed in the generated spaces [PM01, CEW$^+$08, VAW$^+$09, LSWW11]. The work [WMWG09] describes the possibility to simulate the development of a city over time.

The first work I present was developed in 2001 by Parish and Müller and is titled ”Procedural Modeling of Cities“ [PM01]. It describes how it is possible to generate street-networks with the use of L-Systems. An extension to the already mentioned L-Systems is developed in the work. The so-called ”self-sensitive“ L-Systems [PM01, 5] use an additional property that ensures that only non-overlapping elements of the generated street-network are generated.

The placement of the street-network is controlled by some input-images that define intensity-”values“ for different variables. One of the input-images is used to control where the street-networks should be positioned on the landscape for example. It is possible to control the intensity of different street-patterns that are used to create realistic results. The street-patterns are used to mimic different distributions of streets similar to real cities. The typical look of the street-networks of cities like Paris or Manhattan can be reconstructed by using street-patterns like the a ”radial“-pattern, a ”raster“-pattern or a ”branching“-pattern.

The work also describes the procedure to create all the individual ”spaces“ and building areas in the city because the initial result of the procedural generation of the street-network only yields a connected graph which represents the street-elements in the city. The spaces between the streets are reduced in size to account for the needed street-areas at first. A subdivision algorithm is used to create the allotments connected to the streets afterwards and the buildings are placed later. The generation of the actual building-geometry is again handled by using L-Systems. The work describes a ”pre-version“ of the later developed CGA [MWH+06].

A different approach for the generation of street-networks is used in the work by Chen et. al. [CEW+08]. The authors use tensor-fields that can interactively be altered locally to change the directions of the generated street-elements. The tensor-field defines the directions of the streets and is visualized in real-time, so every modification of the values is visible immediately.

A big advantage of this method is that no previously created textures to control the street-patterns are needed because there is no need for street-patterns at all. An initial tensor-field which respects the directions of any water-areas on the map is created at the beginning of the creation process. It is possible to modify the directions of the major or minor roads of the initial tensor field by e.g. adding a radial structure.

Approaches that ’combines the power of procedural modeling with the flexibility of manual modeling’ [LSWW11, 1] are presented in the work [LSWW11] by Lipp et. al. Transforming- and merging-operators are introduced in the work. The solution allows the manipulation of streets of the city as well as the modification of complete regions of the network. The bigger areas of a city can be modified by using different layers for those elements.

It is possible to translate and rotate a complete street interactively, while all affected blocks and lots are automatically updated to fit the changed environmental circumstances. Adding structures like a whole block from another source into the generated city layout or to translate or rotate a big area of the generated street-layout is another possible interactive manipulation of the street-network. All affected parts of the previously generated layout are automatically updated. This means that all parts that are located inside the newly positioned element, are cut out from the street-network and the new part in the new layer is automatically connected to the rest of the layout through new street-elements. The cut out lots are also re-generated to fill the gaps that were created.

City-modeling is an interesting field of research and many other papers exist. When modeling cities procedurally it is possible not only to generate a street-network and the buildings, but also it is possible to simulate the development and growth of a city over the time of several years or even decades. Weber et. al. developed a system in [WMWG09] that is able to simulate the development of a city not only on a regular grid, but using the real geometric configurations [WMWG09, 1].

The simulation works with discrete time steps. At first some major streets are selected for expansion by calculating a probability for each possible street depending on the distance to the nearest "growth-center" in the city. Whenever the street-expansion leads to the creation of new quarters or blocks it is determined if they are actually generated or not. If the traffic-simulation that is performed for each street-element, yields a value that is big enough, the previously "planned" new street-elements are actually generated. More and more buildings can be placed in the city when additional blocks and lots are generated. The land-use and type of the building is also simulated in the application and can change over time. Defined value-functions [WMWG09, 7-8] are optimized to assign the different land uses to the existing and the new generated lots in the city.

2.5 Floor Layout Generation

With the previously mentioned and presented papers it is possible to create scenes with a big scale, a great level of detail and a lot of variation of the generated buildings. It is not enough to only create façades and roofs to create really interesting and realistic scenes though. If a scene should contain buildings with higher quality, e.g. including their interior, additional generation steps need to be performed and the floor, which defines the available space, has to be subdivided into the individual rooms. The creation of the subspaces of a floor of a building is most of the time achieved by assigning the available space to a specific room. Creating floor-plans with a predefined outline is considered to be the hardest possible problem to solve and more complex than the creation of floor-plans with no predefined building-outline. There exist some approaches to create the floor-plans of buildings that result in the creation of more or less realistic room distributions and placements in the floor.

The work "Computer-generated Residential Building Layouts" by Merrell et. al. [MSK10] from 2010 describes an approach that is only applicable to the generation of a residential buildings. A learning-strategy is used to calculate the amount and types of rooms needed in a building and it's floors. The application developed in the paper uses a 'high-level' input, e.g. "two bedrooms and one kitchen" are needed in the building.

An architectural program is then generated from the possibly incomplete list of rooms. This means that all needed rooms, their sizes and adjacencies are defined in the created architectural program. The generation of the full list of rooms is based on real-world data and results in realistic room-arrangements.

The works [HBW06] and [LTS+10] try to solve the problem of creating a realistic floor-plan for a building using mainly geometric properties and a given floor/building outline. The first mentioned work "Persistent Realtime Building Interior Generation" by Hahn et. al. from 2006 focuses on the generation of interior spaces only where they are needed. If e.g. a player in a computer-game walks through a big office building it is probably not necessary to create every room inside this building. By exploiting the fact that most rooms in buildings are connected through a kind of "portal" like a door and therefore not all rooms can be seen from inside the building, only the current room has to be generated (and probably the adjacent ones). To be able to only generate the needed rooms, the work implements mechanisms that ensure that all the rooms are "independent" from all other rooms in the building. The second mentioned paper "A CONSTRAINED GROWTH METHOD FOR PROCEDURAL FLOOR PLAN GENERATION" by Lopes et. al. influenced the work at hand a lot. It describes a simple algorithm, which generates the defined rooms in a floor by using a grid-based approach. The rooms are generated by selecting start-points for the rooms and then expanding the rooms from these start-points until there is no space left to assign to. A description of the algorithm can be found in sections 3.7.1 and 3.7.2, the pseudo-code can be found in Appendix B.2.

In contrast to the previously mentioned papers which calculate the positioning of the rooms using geometric properties, the paper [Mar06] and the previously mentioned work [MSK10] are based on room-graphs. While the work "Procedural House Generation: A method for dynamically generating floor plans" by Martin describes a graph-generation algorithm in four steps [Mar06, 2], the generation of the graph in the second work is based on previously added training-data and a Bayesian network.

Another interesting work "Computing Layouts with Deformable Templates" by Peng et. al. [PYW14] describes a system that can handle buildings with non-axis-aligned layouts. The tiling and subdivision of the available space in a floor uses deformable "tile-templates" that define the allowed shapes for the building-spaces. At first the "problem-domain' [PYW14, 3] is used to create a quadrangulation. The quadrangulation results in a set of quads that fill the space in the problem-domain. After this operation, the defined tile-templates are used to fill the space. Different transformations to the tile-templates are allowed to create the needed tiles that fill the space. An error function is used to calculate the total "error" of the proposed solution and "linear programming" is used to enhance the solution further, i.e. tho reduce the calculated error.

2.6 Furniture Placement

When buildings including their floor-plans should be generated, they are still empty spaces and do not really represent a realistic result. To achieve the generation of such a realistic result, the furniture in the generated rooms of the buildings also needs be generated. Some solutions that compute the placement of previously modeled furniture-parts according to design guidelines or by learning the placement by example exist [MSL+11, FRS+12].

In the work ”Interactive Furniture Layout Using Interior Design Guidelines“ by Merell et. al. [MSL+11] some design-guidelines are first identified and then used to calculate the placements of the furniture in a room. The application works by creating a room and then adding furniture to it. By using a ”density-function“ some suggestions are generated base on guidelines like ”conversation“, ”balance“, ”alignment“ and ”emphasis“. The user can then select a suggested arrangement of the furniture-elements and interact with them. Some new suggestions are then generated based on the user-interactions and can then be selected again until a good-looking arrangement with all furniture is generated.

The work ”Example-based Synthesis of 3D Object Arrangements“ by Fisher et. al. [FRS+12] on the other hand uses only a few input-scenes to learn the spatial relations between different furniture. The relations of the different parts and their typical positioning are learned by the use of an ”occurrence model“ and an ”arrangement model“ [MSL+11, 4, 5]. By using an additional larger database that contains additional objects, which are used to modify the input-layouts and ”fill in the gaps“ of objects not present in the input-scenes [MSL+11, 3], a wide variety of possible new furniture layouts can be generated.

CHAPTER 3

Methodology and Approach

A description of the design of the application follows in this section. The decisions which were made to create a good-performing and a simple application are discussed. Simple here means that the application should be easy to use, it should be simple to implement, simple to modify and it should be simple to extend by creating new production-rules. Some of those decisions are described in a detailed manner in sections 3.1.1, 3.1.2, 3.1.3 and 3.1.4.

3.1 Application Overview

Many aspects of the generation process had to be taken into account to be able to create detailed and realistic-looking buildings with the application. The aim of this work is mainly to create an application, which is able to generate good looking and realistic results by using a procedural system. Another important aspect of a usable application is that it performs it's calculations at interactive speeds, i.e. no long waiting times are allowed, and is extendable. The application needs to be implemented in an elegant and well-performing way.

In the following sections I will describe which techniques were used to implement the procedural system. I will show why I selected the used algorithms and what advantages and disadvantages they have. Starting with an explanation of how it is possible to create a system, which can be extended in the future, I will then discuss how the application is organized to keep things as simple as possible, both, when using the application and in the process of e.g. adding a new rule to the application. The application is implemented in a way that allows to modify as much as possible while providing a system that is as simple as possible at the same time. Especially the two most complex rules in the application, the vertical-connector rule described in section 3.5 and the roof-rule described in section 3.6 were developed to facilitate the generation of the most complex elements of buildings. In the following parts of the master-thesis I will give a description on how this is possible.

3.1.1 Simplicity of the Application

An important aspect of this master-thesis is to keep things as simple as possible. A lot of work went into the process of planning the whole application, so that it is easily possible to implement the functionality that is wanted, i.e. the generation of the buildings and their interiors, while always keeping an overview. While it is always a good idea to keep the application as simple as possible, it is often not quite easy to also keep the structure of the classes and interfaces of the code simple as well.

To achieve simplicity on both sides, the creation of the application and also it's use described in section 3.1.4, I use some techniques that I want to discuss in the following sections. The following section covers the simplicity of the code I had to write for the program, the simplicity of the use of the application is covered in more detail in the sections 3.1.4 and 3.2.1, 3.2.2, 3.2.3.

To be able to manage the creation of the application I decided to split up the individual parts of the program according to the functionality they provide. Details on how the whole application is split up can be found in section 4.1.

MVVM

One very helpful technique to keeping things simple while implementing the application, was the usage of the MVVM[1] design-pattern to create the nodes of the visual rule-editor and define how they work together with the rules of the procedural system, i.e. the rule-set. Implementing the user-controls with the MVVM design-pattern leads to the creation of more classes, but this in turn results in a simplification for the overall system because the different classes are simpler and not tightly coupled. With the MVVM design-pattern it is easily possible to split the user-control definition from the logic and from the data.

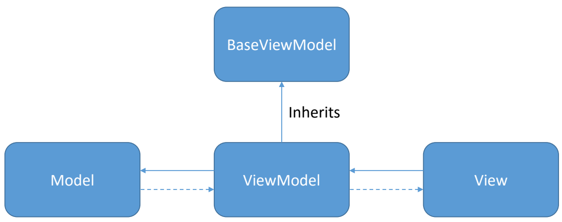

Figure 3.1: The MVVM basic scheme used in the implemented application.

In figure 3.1 the simple structure of the MVVM design-pattern is visualized. For more information about the actual implementation of this design-pattern in the application, please see section 4.7, where the details of the implementation are described.

[1] **M**odel-**V**iew-**V**iew**M**odel

Using Hierarchies of Classes

To ensure a simple structure, all of the rules have one base-class. The only exceptions are the two specialized rules described later in section 3.4. This fact results in the possibility to add more rules to the procedural system quite easily. There also exist base-classes for many other important parts of the application as well. There exists e.g. a base-class for all the implemented view-models and there exists a base-class for the shapes that are used in combination with the production-rules. See [Smi09] for a detailed description of how the MVVM-pattern works. The section 3.3 describes where the shapes are used and section 4.2.4 explains some more details about the shapes, and which types there exist. For a description of the implemented rules and how they are structured, please see sections 3.3, 4.8 and 4.11. The approach of using simple base-classes and deriving from them makes it relatively easy to create new parts of the application like the creation of a new rule. I will describe how a new rule can be added to the system and which steps have to be taken to make it work in section 4.12.

3.1.2 As Flexible as Possible

Another challenge in the development of the application was the desire to keep everything as flexible as possible. There are default definitions for doors, e.g. the door-width, and façades for example, i.e. how the façade-parts are split and modified. All of those definitions are realized by just using the simple implemented CGA-rules like the split-, the scale- and the axiom-rules. Those default definitions can easily be modified, or even removed if no details should be generated at all.

Whenever a scene is loaded into the application, a property node is created in the top left corner of the visual rule-editor. By default there are four rules attached to the node. These rules define the look and the behavior of the respective parts of the buildings. There exists a 'wall'-named rule, a 'door'-named rule, a 'floor'-named rule and a 'facade'-named rule.

The mentioned rules define procedural systems for the named special parts of the buildings, which are generated and handled in a special manner. The behavior of the rules is described in detail in section 4.11. The purpose of those attached rules is that without the need for a basic setup, it should be possible to immediately start creating and defining a building. All of the mentioned rules are attached to the 'root'-rule, i.e. the property node, but can be redefined on a lower level as well. This means that if the user wants e.g. a room to have a different façade, a 'facade'-named rule has to be attached to the respective room-node in the visual rule-editor. This behavior ensures more flexibility over how the resulting building looks like. See figure 4.3 on page 73 for an example. You can see two 'facade'-rules defined in the visual rule-editor at the top of the screen-shot. The left definition controls the façade-generation globally for the whole building, while the right definition is attached to a sub-space and therefore overrides the globally defined other façade-definition for the affected rooms.

Flexibility is an important part of a procedural system and no fixed values are used in the program, everywhere it was easily possible. Some fixed values exist though, section 4.13 describes them and contains more information.

One of the most important decisions were made for the two specialized rules in the application. They are described in detail in the sections 3.2.2 and 3.2.3. I decided to implement them in a way, so the two rules can simply be described as 'shape-generators' because this is the main purpose they serve. The shapes are the basic-elements of everything in the system, see section 4.2.4 for a description on the shapes. The two rules just create a lot of shapes, not only simple shapes, but also polygon-shapes and path-shapes described later. By implementing the two rules this way, it is also possible to modify all the resulting shapes by using further production-rules. For example it is possible to split up the roof-parts if this is wanted. Another possibility is the subdivision of the railings of stairs so that they can be defined to result in a more detailed object. There are a lot of possibilities of how it is possible to modify the results of the two specialized rules that were implemented. See figure 3.2 for two examples of a modified roof and a modified stairs. More details on the rules can be found in the section 4.11.

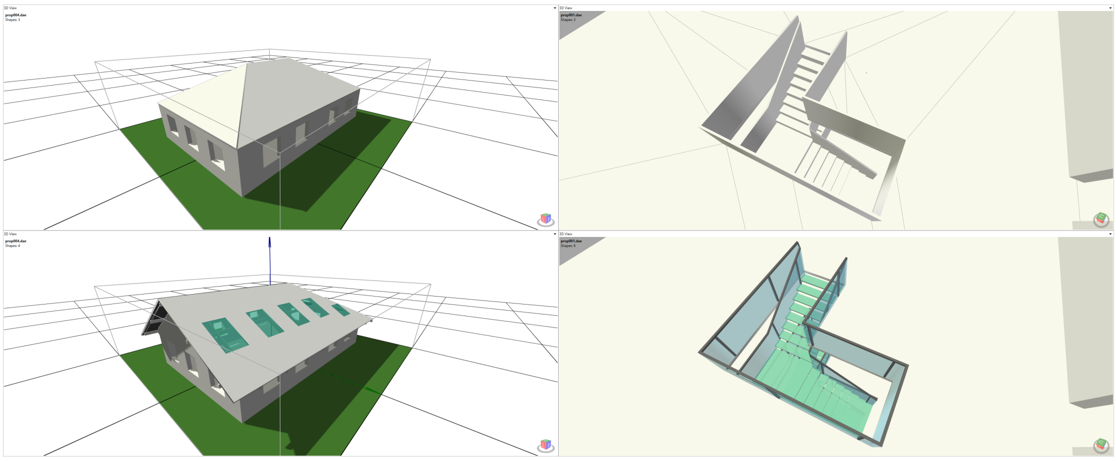

Figure 3.2: Comparison of a building with different definitions for some of the generated roof-shapes on the left side. Comparison of a building with different definitions for some of the generated stairs-shapes on the right side.

3.1.3 Extensibility of the Application

The developed application is designed and implemented in a way, so that additional rules can be created for it in the future. An explanation of this process can be found in section 4.12. By providing a base-class for all the relevant parts of the application, it is possible to add new rules, which should be available to use in the application, quite easily.

Not only can there be added more procedural generation rules, it is also possible to modify the whole generation process easily. The generation process of a building is described in detail on page 71 in section 4.6 and consists only of a few calls of methods. It is possible to change the floor-planning algorithm for example by simply replacing the respective function-call with a call to a similar function-call, which produces the needed outputs. Changing the algorithm was not tested because it would have been a lot of effort to implement a second floor-planning algorithm.

3.1.4 The Visual Rule-Editor

The application is defined to be useable without the need to write a single line of code. A visual rule-editor to create the rule nodes is used to achieve the goal to create complex procedural systems without writing any kind of program. It facilitates the creation of all building-elements and rules. The editor is the main interaction-point between the user and the procedural generator and therefore one of the most important parts of the application. A screen-shot of an example of a procedural system defined in the visual rule-editor can be seen in figure 3.3.

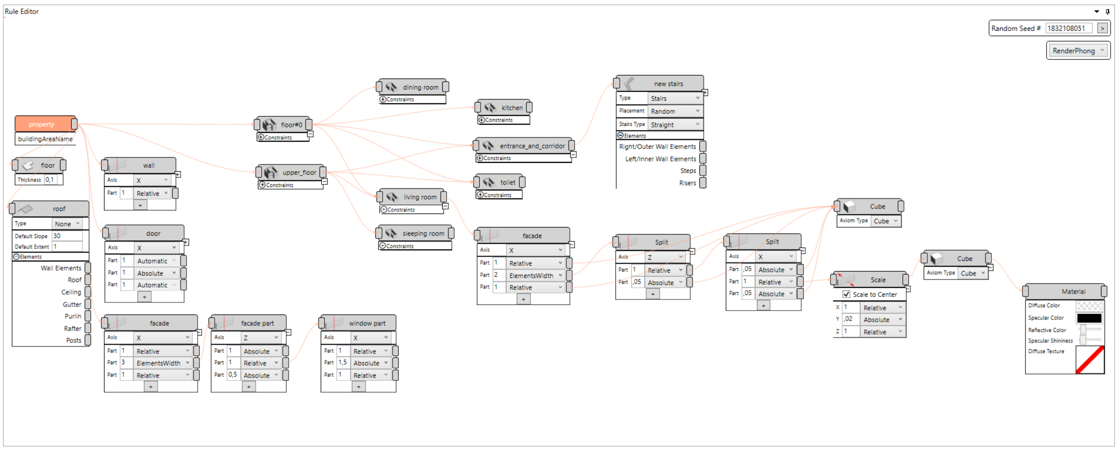

Figure 3.3: Screen-shot of the visual rule-editor. There are many defined rules in the procedural system, which represents a medium-complex example. Not all defined existing rules in the procedural system are shown in the screen-shot. They can be hidden to maintain a better overview.

In the following sections the structure of the different parts of the application, which work together when the user works with the editor is presented. An overview of the main functions of the editor, which were needed to simplify the creation of the complex procedural systems is discussed next.

Design of the Editor

The visual rule-editor is based on an existing custom WPF[2]-control published in [Dav12]. The existing code is already structured using the MVVM design-pattern, which was introduced in section 3.1.1, and did not need to be changed. More information about how the editor is structured into the individual classes and how they work together can be found in section 4.3.2.

Features of the Editor

The editor needed some broadly used and well-known features to interact with the nodes to be really useful and easy to use. The nodes are displayed in the visual rule-editor. Since this editor is one of the main parts of this application, a lot of effort went into defining and implementing the needed features, which were continuously identified throughout the implementation-process. A short list of possible additions to the features of the rule-editor can be found in section 6.3.1 and shows how the editor could be enhanced in the future.

Add and delete nodes It was clear from the beginning that the editor has to be able to create and delete the nodes in a graphical and actually useful manner. The creation and the removal of those nodes of the visual rule-editor is designed to be as easy and straight-forward as possible.

Adding a node to the procedural system is possible by using the right mouse-button and then selecting the desired node-type. Another possibility is to simply start dragging a connection out from a node-connector using the mouse. If the start-node e.g. is a room-defining node, i.e. a room or a room-collection, and there is no other node present at the position, where the user releases the mouse-button again, another new room-node is attached to the start-node automatically. If the start-node is a different type of node, the best fitting options to attach a new node are displayed automatically when the mouse-button is released. More details on how the implementation of this feature works can be found in section 4.3.3.

It is also possible to easily remove nodes that are not needed in the system anymore. The decision to support and use keyboard-input in the visual rule-editor where it is useful makes it possible to remove the selected nodes by simply hitting the ***Del***-key.

Move nodes To create a really useful editor it is also necessary to be able to move existing nodes, so that they can be grouped together visually in an easy way. This feature is actually one of the most important implemented features of the editor because it would be impossible to keep an overview over the created procedural system without the support to move the nodes. When the created procedural system gets bigger and more

[2] **W**indows-**P**resentation-**F**oundation

complex because there are many rules in the system, moving connected rules next to each other drastically improves the overview over the system. Moving the nodes is performed with the mouse. One or more nodes, i.e. room-nodes or rule-nodes, are selected in the editor at first. When one or more nodes are selected, they can easily be moved by using a drag-operation with the mouse. When the drag-operation starts, the cursor of the mouse needs to be located above one of the selected nodes to work.

Hide nodes Keeping an overview over the procedural-system is a critical requirement for the editor. The possibility to hide nodes is very important and can help keep the procedural system visually simple. When a building gets more and more complex and detailed, more rules, i.e. more nodes, are needed to define these details. Not all parts of the procedural-system need to be visible at every time throughout the definition process because if the user e.g. is working on the definitions of the doors in the building, there is no need to display all other nodes representing the other production-rules. This means that it is possible to hide parts of the procedural system by simply clicking a toggle-button, which is present at every node that has attached child-nodes.

Multiple selection of nodes is possible in the rule-editor. This feature allows a fast and easy arrangement of the nodes in the editor as well as the ability to easily delete the selected nodes. This feature is important in combination with the possibility to hide subsets of nodes, i.e. the child-nodes. If a selected node that contains invisible child-nodes should be moved in the rule-editor, not only the selected parent-node is moved, but all hidden child- and descendent-nodes are selected and moved along the selected parent-node as well. The selection of multiple nodes is performed by clicking with the left mouse-button on a node in the visual rule-editor. A colored border around the node is shown to visualize the 'selected'-state of the node. It is also possible to click on a node while holding down the ***Shift***-key on the keyboard. In this case not only the current selected node is selected, but also all child- and descendent-nodes are selected, too. This feature is used to easily move around a whole subset of nodes at once, which helps to structure the procedural-system and to maintain an overview. Another possibility to select multiple nodes is available by holding down the ***Ctrl***-key while drawing a rectangle around nodes in the visual rule-editor. All nodes that are positioned completely inside the drawn rectangle are then selected and can be moved around or copied.

Pan and zoom the visible area of the editor. This is a main feature to keep an overview over the created procedural-system. It allows a fast and easy change of the visible part of the complete procedural-system. With this feature it is possible to use an "infinite" canvas to place the rule-nodes. The feature is very important to easily add more and more rule-nodes into the system. The panning feature is performed by left-clicking over an empty space with the mouse. While not releasing the mouse-button, but instead dragging the mouse around the visual rule-editor, the visible area of the visualization of the procedural-system is updated accordingly. The moving of the area is in fact created by inversely moving around all nodes of the procedural-system. The zoom-feature on the other hand is very useful when a subset of the system should be edited and modified. Zooming into and out of the visualization of the procedural-system can be performed by using the mouse-wheel. The zooming-operation takes the position of the mouse on the visual rule-editor into account, so it is possible to zoom to the focused nodes in the rule-editor, i.e. the zooming-operation is always relative to the current mouse-position.

Fit all displayed, i.e. not hidden nodes of the created procedural-system into the visible area of the visual rule-editor. As previously mentioned, it is possible to zoom into the procedural-system. Zooming into the procedural-system is useful, but when only a part of the system is shown in the visual rule-editor, it is hard to maintain an overview over the complete system. To be able to switch back to a view, which provides a good overview of the procedural-system is therefore really useful. This feature can be used by using the keyboard-shortcut ***Ctrl+F***. The feature fits all visible nodes in the area of the visual rule-editor.

Copy and paste is a useful feature that almost everyone uses on a PC regularly. The visual rule-editor allows the users to simply copy and paste nodes and complete parts of the procedural system, so parts of an already created system can be reused and then modified. Selected nodes in the visual rule-editor can be copied by the use of the well-known shortcut ***Ctrl+C*** and can be pasted in again with the shortcut ***Ctrl+V***. The nodes are placed at the current mouse-position and are exact copies of the previously copied rule-nodes. All connections that connect the selected nodes are also copied and inserted between the newly pasted rule-nodes.

3.2 The Building

A simple structural subdivision was developed for the buildings. The generated result of the procedural generator consists of three main parts. It follows a short description about those three parts of the buildings and then there will be a much more detailed description on how the parts work together in the generation process.

The first part of those three parts is the floor. A floor is defining a space, where the rooms are placed and distributed. A building may contain only one floor in total, but can also contain many different defined floors.

The second important parts of buildings are the floor-connecting elements, namely the elevators and stairs. They connect adjacent floors in the building. Depending on the definition of the floors, the vertical-connections, which are described in more detail in section 3.2.2, are only connecting two floors, or if the floors are all defined equally, they range form the first to the last floor.

The last and also very important main part of the buildings is their roof. They form the top-most elements of the buildings and may have a different look and type. The most significant types of roofs are the simple flat roofs, the pent roof and the hipped roof.

3.2.1 Floors and Rooms

The most important element of a building is the floor. This floor contains at least one room, most of the time there will be a lot more rooms contained, though. If only one room exists in a floor, the size of the room equals the size of the floor in this case.

To be able to generate a big range of types of buildings like a residential building or an office building, a general approach for defining the rooms inside a floor had to be created and developed. The ability to define all the many different sized rooms in a residential building and using this same system to e.g. define many equally sized and offices with only a few differently defined other rooms was a desired feature of the application.

When searching for a solution to this problem it became clear that some kind of hierarchy of the definitions of the rooms and room-collections is a simple solution to the problem. I will describe how the system works in the following section and describe the benefits of using such a hierarchical system.

The Room-Hierarchy

When a floor in the application is defined, it describes the whole available area for rooms, which should be placed in this floor, i.e. inside the space it is assigned to. It is possible to create a separate node for each room in the visual rule-editor. This feature is often useful for irregularly planned floors like they occur in residential buildings.

If more than one room or room-collection is attached to a floor, the first attached room becomes a 'connector-room'. This means that all sibling-rooms, i.e. all rooms or room-collections that are located within the same parent's sub-space, to this connector-room will be connected through an opening, e.g. with a door, to it. By using this approach, it is always ensured that every defined room is connected to each other. The main benefit of this approach though, is that is is not only possible to define 'flat' room-hierarchies, but also deeper one's which are described next.

Example One If e.g. a residential building should be created and the sleeping-room should be connected to the corridor, but also to a 'private toilet', which is only accessible through the sleeping-room, this is easily achievable. In this case a new 'private' room needs to be attached to the corridor, then the sleeping-room and a toilet to this new room-collection need to be attached. The room-collections are just 'containers' for their child-rooms.The sleeping-room is the 'connector-room' for this new room-collection including the private toilet. The sleeping-room has indeed two room-connections in this example. It is connected to the corridor and it is also connected to the private toilet. This is also a good example on why the connections between rooms are restricted to only a subset of the neighboring rooms. An example of the result of a definition of rooms as described can be seen in figure 3.4.

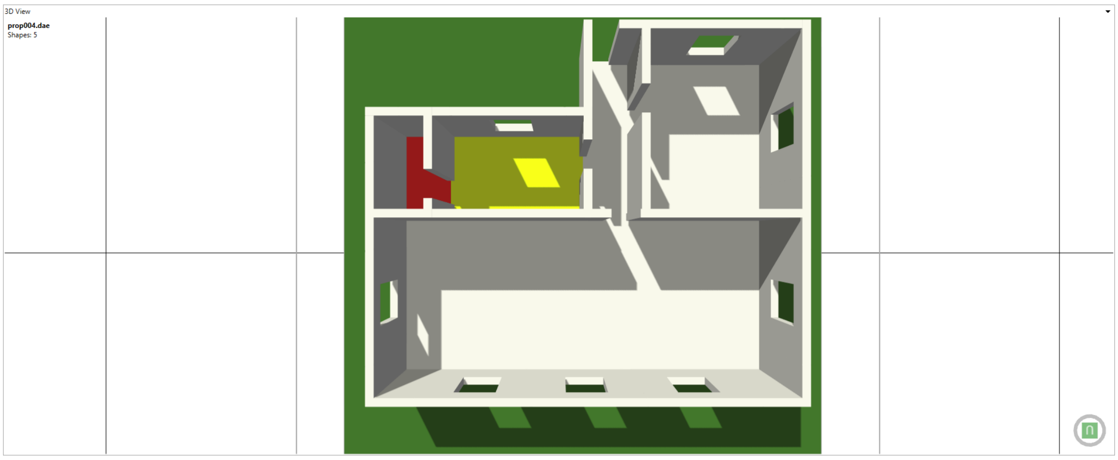

Figure 3.4: Screen-shot of the rooms of a building from above. The private rooms are highlighted with a colorful floor-definition. The sleeping room's floor appears yellow, the floor of the private toilet is colored red. This is an example of the benefits of using a hierarchy for the definition of rooms, rather than just a flat structure and randomly chose connections between the defined rooms.

Example Two Another example of using these deep hierarchies in the definition-process of the floors is the ability e.g. to create an apartment-building. If six equally-defined apartments are desired in one floor for example this is easily possible. An equal definition means they are almost equal in size and they all should contain the same set of rooms. Two rooms need to be added to the floor. The first one represents the corridor, which connects all apartments with the outside of the building, the second room represents the apartments, the 'count' needs to be set to six in this case. All the rooms, which are included in one apartment need to be attached to the apartment-node in the visual rule-editor.

In the visualization of the second example in figure 3.5, the six defined apartments are connected by a corridor (white floor). The apartments consist of five individual rooms, namely the entrance room (light gray floor), the toilet (medium gray floor), the kitchen (dark gray floor), the living room (light yellow floor) and the sleeping room (medium yellow floor).

Example Three A similar approach is also useful for the generation of office buildings. A floor of the office-building may consist of two or more different 'regions'. For example

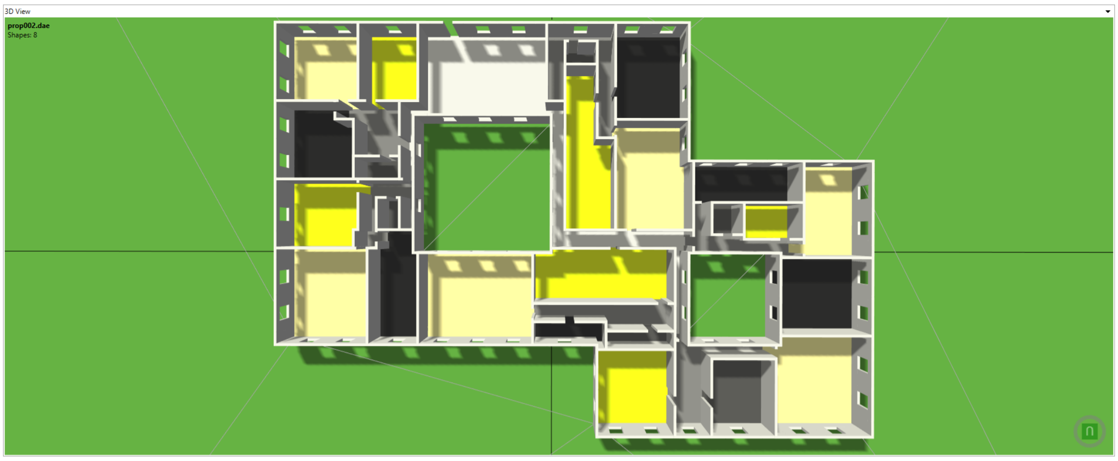

Figure 3.5: Screen-shot of a building from above. There is a corridor connecting all apartments and six equally defined apartments (they all consist of the same amount of rooms with the same size-definitions). To highlight the different rooms, all rooms have a different floor-color.

a region that includes only a few big single offices and a region, i.e. a room-collection, containing many smaller ones exist. An example of this technique can be seen in figure 3.6.

Figure 3.6: Screen-shot of a building from above. The bigger rooms are displayed with a yellow floor, the smaller ones are displayed with a dark floor-color.

The possibilities with the approach of using room-hierarchies are almost limitless and the system is really flexible, the user of the application just needs to define the hierarchies of the rooms (e.g. sleeping-room and private toilet).

The Room Connection

By using these hierarchies of rooms and room-collections, which directly correlate to the structure of room-nodes in the visual rule-editor, the connectivity is always ensured. The definition also restricts which room is connected to which other room, but this is a desired property.

Let's have a look at the example one from before (see figure 3.4). When the generation-process of the room-connections starts, there is only one room to connect to available for the private toilet. The exact position of the connection is not defined, but it is guaranteed that the private toilet will always be connected to the sleeping-room.

The room-connection generation process is started after the rooms were positioned on the floor. The exact positions of those connections are calculated randomly, but it is always defined to which other room a room connects to.

The Room-Planning

Before the room-connections are generated, there exists the generation process for the rooms themselves. I will shortly discuss which method is used to calculate the needed positions and shapes of the defined rooms in a floor. A more detailed description of the floor-planning algorithm can be found in sections 3.2.1 and 3.7.1 and a simple pseudo-code of the algorithm can be found in Appendix B.2.

I decided to keep the things as simple as possible, while creating realistic and good-looking results. To reach these goals, different possibilities to create a floor-plan were researched. A simple solution by Lopes et. al. [LTS$^+$10] uses a grid-structure to calculate and position the rooms in the floor.

The application works with a hierarchical definition of rooms, so a modification of the algorithm described in the work [LTS$^+$10] was necessary. The algorithm basically starts from the root element of the floor, which is the floor-element itself. The desired sizes for all attached room-definitions are determined and a 'start-position' inside the floor is calculated. Starting from this position all rooms grow/expand until all available space of the floor is assigned to a room. If one of the planned rooms is a room-collection, the whole process starts again, but this time just for the sub-space, this room was assigned to in the first step. By using this recursive approach of distributing the rooms inside the floor, the mentioned deep hierarchies of rooms are possible. All details are described in the section 3.7.2.

3.2.2 The Vertical Connection

The creation of the vertical connections is not an easy task. I decided to simplify the generation-process by creating the additional vertical-connection - rule. The geometry-generation is handled in the rule, so this part of the creation of the vertical connections is nicely handled. If there exist more than one vertical connection from one floor to another, the generation-process is executed for each of them separately. For a more detailed look at the design decisions please refer to section 3.5 and for an explanation of how the implementation works, please see section 4.11.12.

The real problem, which arises with the definition of vertical connections is the positioning of them inside the rooms. Since the positioning process is started after the rooms are planned on a floor in the current implementation, there sometimes exists no valid possible position and the connection cannot be placed properly. To overcome this issue, the amount of needed space for the vertical-connector is initially added to the floor-planning stages, so it is more probable for the room to be big enough to position the connector in a valid manner.

The main reason for the positioning-problem of the vertical-connectors is not the size though. Since there exists the additional requirement for each room to be connected to another room, there exists a validity check. The room to connect to is implicitly defined for each room as mentioned in section 3.2.1. This test checks if there is enough space for each room to connect, so that for each of those rooms to connect, a room-connection can be created in the following generation-step. If a potentially valid position for a vertical-connector inside a room is found with regards to the spatial circumstances, this 'connection-check' is performed and may fail. In this case, another possible solution for the placement is searched and tested, until a valid position is found, or none of the checks were successful. In the latter case, the vertical-connection is not created.

Types of Connections

Two types of vertical-connections exist in the application. The first type is the stairs, which may have different predefined shapes. It is possible to create straight, L-Shaped, U-Shaped and spiral stairs. A more detailed description about the stairs can be found in section 3.5.2.

The elevators are the second type of vertical-connections. They are described in more depth in the section 3.5.1.

3.2.3 The Roof

The roof of the building is the last of the very important elements. A special rule for the handling of the generation of the roof, like for the vertical-connectors mentioned before, is defined and implemented. The creation of the roofs should be possible in an easy way while maintaining a maximum of flexibility, which is possible with the implemented rule.

Since the generation of the roof is the last step of the building generation, all defined floors and rooms, including all the connections, and even the façade-elements already exist. The system is defined in a way, so that the generation of the roof does not interfere with any other elements of the building.

Roof Types

A requirement for the application is to be as flexible as possible, which means that it should be possible to create different looking roofs for the buildings. It is possible to create simple flat roofs, pent roofs and hipped roofs. The most complex roof-type are the hipped roofs because there is more than one roof-part present in the resulting geometry and the shapes of those parts of the roof need to be calculated and generated.

As described in section 3.7.3 in detail, a straight-skeleton algorithm is used to generate the individual parts of the hipped roofs. The pseudo-code of the implementation can be found in Appendix B.3. The algorithm is modified and adapted to the needs of a roof-generation application. It works for all building-shapes that only consist of straight lines and it is possible to handle courtyards of the building correctly for example. The individual parts of the generated hipped roof may have different slopes defined as well as different 'roof-part-extents'.

The straight-skeleton algorithm is modified in a way, so that it allows the handling of different slopes of the individual roof-parts. More realistic roof-definitions can be created this way and another (a fourth) type of roofs is made possible: the saddle roof. It is just necessary to assign a slope of 90° to the respective roof-parts, for more information on how assigning different slopes to roof-parts is working, please see 3.6.3.

The handling of the roof-extent is handled in a post-generation-step of the roof-parts. The points defining the eaves of all the individual roof-parts are modified in their position and height.

When a building contains a definition of a pent- or a hipped-roof, it is possible to add more detail to the roof. One aspect is the automatic handling of the defined 90° roof-parts, which results in additional wall-parts of the building, i.e. not only roof-parts themselves are generated in the roof-generation step. It is possible to use the roof-rule to create a lot more shapes than just the mentioned roof-parts themselves. These parts are the purlin-elements and the rafter-elements as well as an additional 'ceiling' element. With the last mentioned element it is possible to define a vertical closure for all the rooms that are positioned in the top-most generated floor. The other two mentioned elements are part of the roof's substructure and lead to a much more detailed visual appearance of the roof, by adding a lot more geometry to it. The generation of the additional and optional elements is handled by the use of other existing basic CGA-rules in the system, described in section 3.6.3.

3.3 How the Rules Work

The whole procedural system is based on the mentioned production-rules. These rules are used to modify and generate the geometry of the building, it's walls, roof-parts, stairs, etc. . The rules are defined in the visual rule-editor and they use the defined settings to alter a shape.

Each of the developed rules in the system implements a function which is executed for every production-step in the generation-process. The signature of the function is the same for all the rules and always uses a "Shape", a "PolygonShape" or a "PathShape" as an argument and returns an Array of Shapes as a result. More information about the shapes can be found in the section 4.2.4.

In each generation step, a pre-defined or pre-calculated scope or Shape with its size, rotation and translation is used with the rules. After the application of a rule to a Shape, the execution yields one or more new and modified Shapes as a result. Those results are then used as input for the next production-rules that are defined in the procedural-system.

3.4 The Specialized Rules

An introduction of two new rules that are implemented in the application follows. Those two rules are more specialized than the default CGA-rules used in procedural generators. The usage of both of those two rules result in the simple generation of complex geometry.

Apart from the standard CGA-rules like split, re-size, translate, rotate, etc. I decided to implement two more specialized rules that simplify the generation process of 3-dimensional buildings drastically. The first rule is used to connect two or more floors of the building and are named 'vertical-connectors'. The second new rule is used for the generation of the roofs of the buildings are named 'roof-rule'.

Both rules do not derive from the basic rule view-model and are not handled like the other implemented rules because they are not applied to a shape. The assumption that a building can only have one roof is made. This is a simplification, but it works for most buildings very well. The generation of the roof is executed after the geometry of the rooms and façades is calculated. As described in section 3.2, the generation of the roofs is in fact the last step of the individual generation steps in the current state of the application. The vertical-connectors are also treated specially because the room-planning partly depends on the presence of a vertical-connector inside a room. The connectors might need a big area of the room that should contain it and the connectivity to the rooms that are adjacent to the vertical-connector-containing room has to be preserved, see section 3.5.2 for a description.

The simpel idea behind the two introduced rules is that they just serve the purpose of shape generation. This means that the resulting geometry is generated exactly the same way like all other shapes and elements are generated. The most important reason and the biggest benefit of using just the three implemented shape-types described in section 4.2.4 as a result of the two production-rules is that they are all modifiable by the use of the other implemented CGA-rules. It e.g. is possible to split up the railing of a stairs into many more shapes to be able to add a lot more detail to it. The modified railing is not just a big and complex path-shape, but instead it is split up into more different-looking path-shapes. Each one of those created elements can be seen e.g. as a section of the railing consisting of a glass panel and a handrail.

To be able to modify these generated elements, they have to be able to be connected to other CGA-rules. I decided to implement 'elements' for the two rules. The elements are different for the two rules, but they work in exactly the same way. For a description of the elements, please refer to figure 3.7. All implemented elements are represented in the user-interface by a label and a connector. The connector is used to modify the elements by assigning other production-rules to it, i.e. by connecting them.

I implement several different elements for the vertical-connections as well as for the roofs. If the user adds one of these rules, some rules will automatically be attached to the vertical-connector rule and to the roof-rule. By changing e.g. the type of the roof in the visual-rule-editor, the elements will change to fit the type of the selected roof. The elements are ready for modification by attaching other rules immediately.

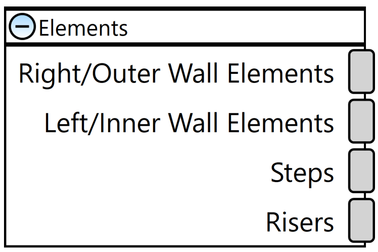

Figure 3.7: The implemented elements of stairs. All elements are grouped at the bottom of the visual representation of the stairs-rule. Each element is a combination of a label and a connector. The individual shapes of the elements are calculated automatically and can be modified by attaching rules to the connectors.

3.5 The Vertical Connector / Stairs Rule

The vertical connectors are the first specialized rule which are implemented in the application. It serves the purpose of connecting two or more floors of a building, thus the name of the rule. The most commonly realized types of vertical connectors were chosen and implemented. The connectors can be used to create elevators, which are the most simple vertical connectors, and stairs, which are much more complex. If a stairs should be generated to connect two vertically adjacent floors, four different types of stairs can be selected. The default 'straight' stairs, a 'U-shaped' stairs, the 'L-shaped' stairs and at last there are the 'spiral' stairs are implemented in the application. The stairs can be seen in figure 3.8. An example for a straight stairs is shown in the top left region, a U-shaped stairs in the top right region, a L-shaped stairs in the bottom left region and a spiral stairs in the bottom right region of the image.

Types of vertical connectors Some different types of vertical connectors exist in the system. In the following sections, a description of all their special elements and behavior is given.

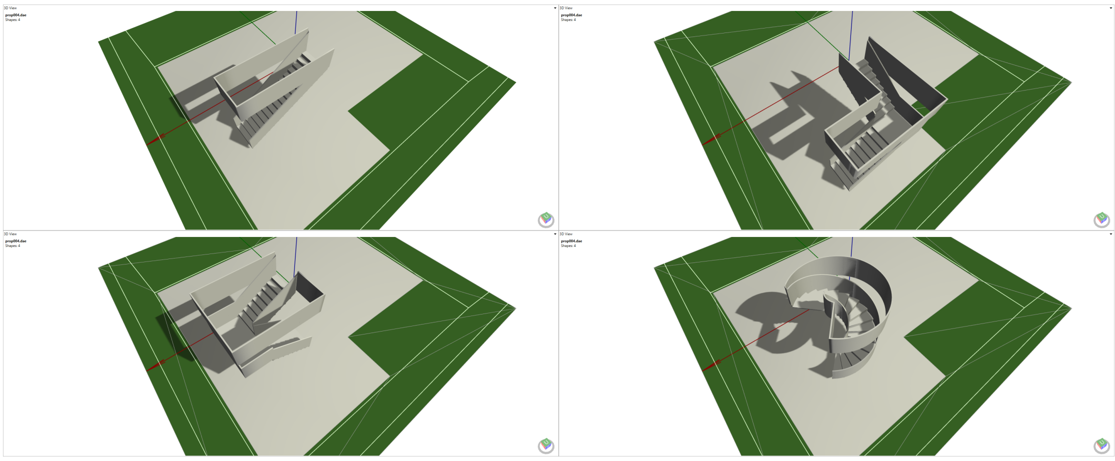

Figure 3.8: Top left: example of a straight stairs. Top right: L-shaped stairs. Bottom left: U-shaped stairs. Bottom right: spiral stairs. Only the default stairs-definitions are used in the visualizations.

3.5.1 The Elevator

The elevator is the most simple vertical connector present in the system. They occur in a wide variety of different buildings so it was decided to implement them in the procedural generator. Elevators may be located in an apartment building, but they are usually also present in office buildings, where there are a lot of floors. The positioning of these elevators is relatively simple and almost always succeeds, compared to the stairs because an elevator uses less space in a room, which makes the positioning easier. Finding a valid position inside a room is easy because it is more probable for the adjacent rooms to still be accessible, even when the elevator is positioned at a wall that contains the desired room-connection.

When the elevator is positioned by the application, only it's position has to be defined. The 'rotation' of the elevator is determined after a good position for the elevator is found. The position is valid, when the needed space of the elevator lies completely inside the room-defining outline and when there exists enough space for all rooms that need to be connected to the elevator-containing room.

After a valid position of the elevator is found, the sides of the elevator-doors are calculated. The side of the doors may be different for the individual connected floors because the layout of the connected floors may be different. An example of this case can be seen in figure3.9. The elevator-door is typically headed to the room-center. If the room-center is positioned in such a way that the elevator-door would be positioned at a wall, another fitting side is selected for the elevator-door.

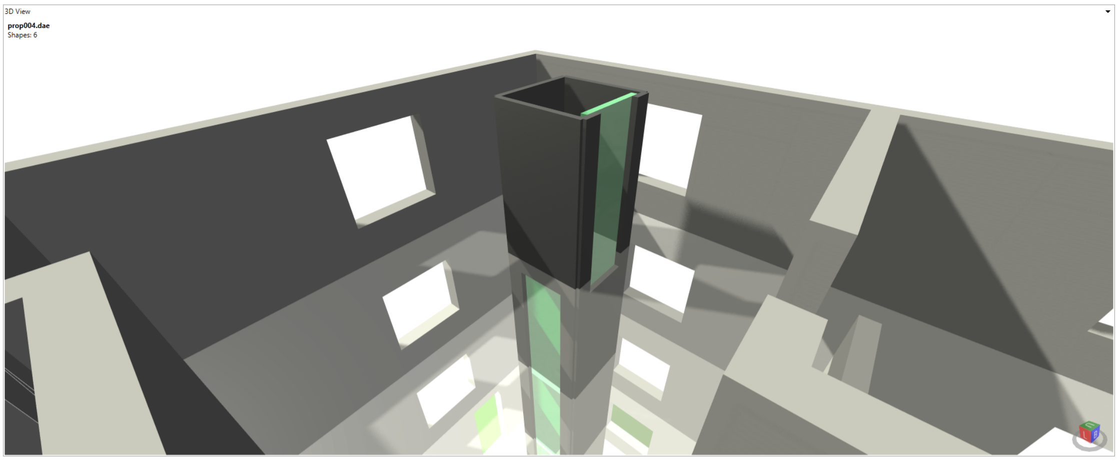

Figure 3.9: Visualization of an elevator. The floor-setup at the top-most floor is different than the ones below. The doors of the elevator are positioned at another direction of the elevator because of the different shapes of the rooms.

The Elements of an Elevator

When a vertical-connection is added and the type is changed to an elevator, some elements that can be modified and worked with, are created. The default setting including the attached element-rules consists of four rules.

The first element is the 'Wall elements' - named element. It represents all the shapes that are created for the walls that surround the elevator. There exists a scale-rule named 'Wall elements' attached to the elevator-element to scale the wall elements to the needed sizes because by default all simple shapes have a side-length of 1m. It scales the shapes in a way, so that the walls of the elevator are 0.1m thick. There is another rule attached to this scale-rule. It is used to actually generate the geometry for the created shape-scopes.

The second and last element present in an elevator vertical-connector are the 'Entrance elements'. As the name suggests, they are used to generate the shapes for the wall-side of the elevator that contains the elevator-doors. A simple split-rule is attached to the element and leads to two simple shapes at the side of the door. Those side-elements are connected to the definition of the other wall elements for the elevator, so they will be generated the same way. The middle element is not connected to anything by default, but any user-defined rule or set of rules may be attached to model and generate the actual doors of the elevator. Figure 3.9 is an example for additional modeled elevator doors (green).

3.5.2 The Stairs

The second type of the vertical-connectors are the stairs. There are four implemented types of stairs in the system to provide flexibility to the generation of the buildings.

The process of the positioning of the stairs is a lot more complicated than the positioning of the elevators. The main reason is the bigger space-consumption of the stairs. While an elevator only have a constant space requirement, regardless the height of the rooms, the stairs space requirement rise as the height of the floors rises. The taller the floor, the more size is required by the stairs that connects the floor with it's top adjacent floor. The reason behind this is that more steps are needed to connect the floors.

The amount of steps needed to create a stairs is dependent on the floor-height and the rules that define the maximum step-height and minimum step-width are taken from the ÖNORM. This approach leads to realistic stairs, which are not too steep or flat, and also implements a realistic size-constraint to the stairs. The restrictions taken from the ÖNORM are used to ensure the stairs consists of enough steps. A maximum height of 0.18m per step is used in the system. The steps also need to be wide enough, each step has a length of at least 0.27m, measured 0.45m from the 'inner' side of the step. The last constraint that is implemented in the system is that, according to the ÖNORM, the average human step-length is measured at 0.63m. The formula $2 * h + b = steplength$ is the reason for the maximum of 0.18m height and minimum of 0.27m length of a step $(2 * 0.18 + 0.27 = 0.63)$.

An additional aspect of the positioning of stairs is that it must be ensured that people can actually 'access' it. This means that there is the need for an additional free space at the start and at the end of each stairs. This space is added to the stairs, when the positioning process starts for the vertical-connector. For simplification reasons, this space requirement at the start and at the end is assumed to match the width of the stairs and is a square. The figure 3.10 is a visualization of the additional space needed.

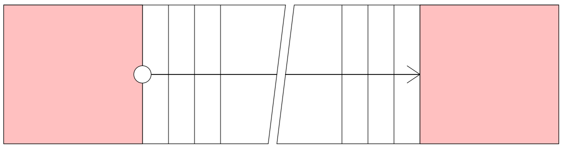

Figure 3.10: Visualization of the additional space-requirements for placing a straight stairs inside a room. The reddish areas are the added spaces that also need to fit in the room. Between those areas, the actual stairs is positioned. It is cut-off in the visualization.

The Straight Stairs

The straight stairs are the simplest type of the stairs, but a lot more complex than the elevator. Figure 3.8 (top left) visualizes an example of a straight stairs. The outer shape of a straight stairs is a rectangle and all steps are aligned in one direction.

This circumstance leads to a big space-requirement in one direction of the stairs, i.e. the stairs is quite long. If we assume a floor height of 3m and apply the previously mentioned rules taken from the ÖNORM, 17 steps are needed in total. This number is true for all different types of stairs, as described later. The 17 steps needed to connect the two imagined floors 'consume' a length of at least 4.32m ($(number\_of\_steps - 1) * 0.27m$), which may be a problem for the positioning of this stairs in a small room. In fact the length is not only 4.32m but a little bit more because of the step-length rule, where $2 * height\_of\_step + length\_of\_step$ always equals 0.63m to ensure a comfortable usage of the stairs.

The total size needed to be able to position this straight stairs is 1.5m (default stairs-width, not changeable in the system) by 5.94m (4.44m plus 1.5m because of the additional space to access the stairs).

The L-Shaped Stairs

The next and slightly more complex type of stairs is the 'L-shaped' stairs. The main difference to the straight stairs is that this type of stairs is not just heading in one direction, but it has a turn in the middle, which is demonstrated in the top right area of figure 3.8. In the system it is assumed for both parts, before the turn, and after the turn, to be the same size and to have the same amount of steps, therefore an additional step is generated for an odd number of minimum steps.

The U-Shaped Stairs

The same is true for the U-shaped stairs. If there is an odd number of steps that are needed to connect two floors, an additional step is generated. This is done to simplify the generation of the geometry, as well as a simplification for the stairs-positioning because the outline of the stairs remains a rectangle and is therefore easier to position inside a room.

A U-shaped stairs is again a slightly more complex type of stairs than the previous mentioned L-shaped stairs. It has not only one, but two turns into the same direction, forming a 'U'-shape. The two turns, i.e. the platform, do not contain any steps, which also is a simplification in the system, and the required additional space for 'entering' and 'exiting' the stairs are added again, when the system tries to position the U-shaped stairs inside a generated room. An example of a U-shaped stairs can be seen in the bottom left area of figure 3.8.

The Spiral Stairs

The last and most complex type of stairs in the system are the spiral stairs, which can be seen in the bottom right area of figure 3.8. The positioning of those stairs is not more complicated than the previously mentioned types of stairs, but the shapes that are generated using this type of stairs are a lot more complex than the shapes that are created for the other stairs. Spiral stairs, like the other types of stairs, also take the height of the floor into account when the needed size is calculated. If e.g. a floor is 3m tall, the diameter of the spiral stairs is smaller than when a floor is 5m tall. See figure 3.11 for a comparison of the space needed to create a spiral stairs for the different floor-heights mentioned.

One example for the more complex shapes are the steps. For the other types of stairs, the generated step-geometries are simple shapes because the shape of those steps are simple cuboids that can be modified. The geometry of steps in a spiral stairs are polygon-shapes, though. The polygon for each step looks like a part of a ring-shaped geometry. Other complex shapes are the railings that follow path around at the inner side as well as the outer side of the spiral stairs.

Additionally to the more complex geometries that are generated for the spiral stairs, also their positioning and rotations are determined in a more complex manner. The steps and risers have to be positioned around the center of the spiral stairs and rotated accordingly.

While the geometries and the transformations of the individual elements of the spiral stairs are more complex than they are for all the other stairs and the elevator, it is still possible to modify each and every generated shape. This fact is the main reason for the the vertical-connectors be designed and implemented as shape-generators. The shapes are always either simple shapes, polygon-shapes or path-shapes and can therefore be modified in all ways by using the CGA-rules.

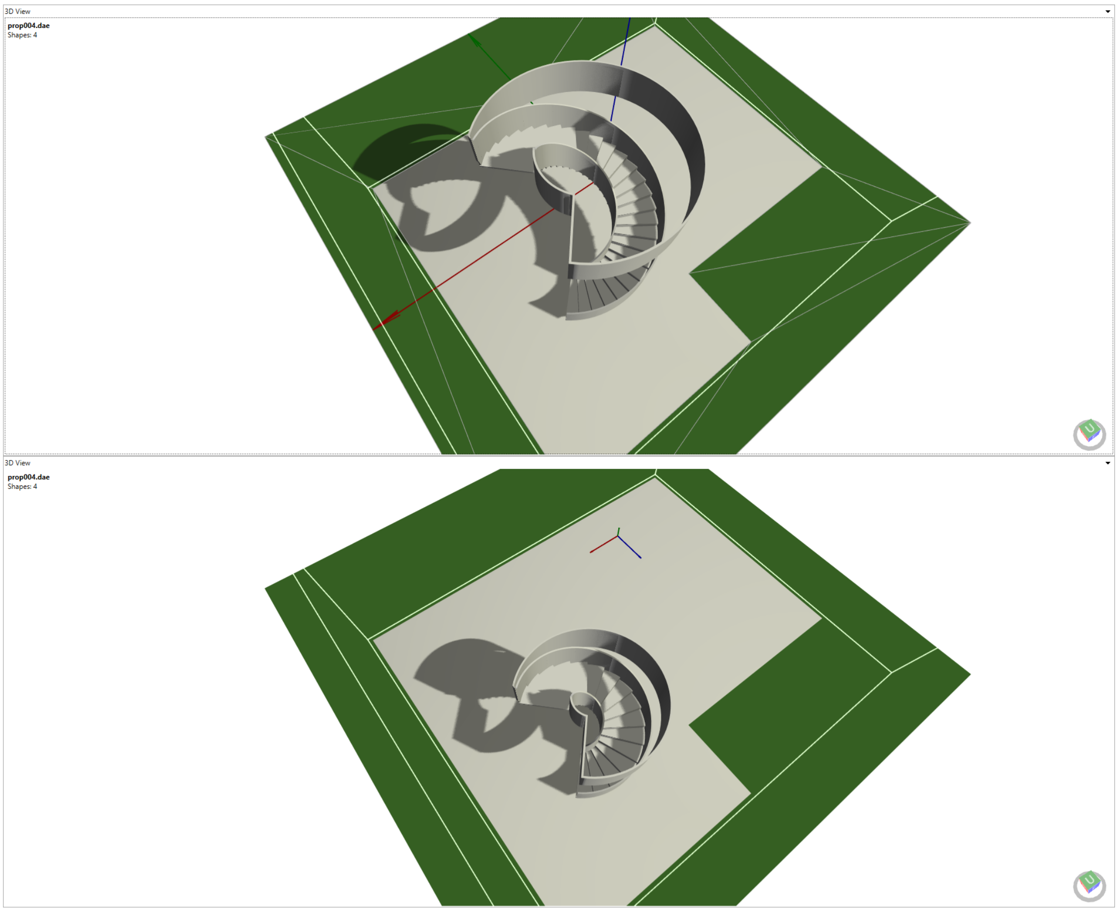

Figure 3.11: Comparison of the size and the space needed of a spiral stairs, which are created for a floor-height of 3m (bottom) and for a floor-height of 5m (top).

The Elements of the Stairs

When a stairs is added to the procedural system, some shapes are generated if the stairs is connected to the building somehow. In the visual rule-editor there exist some elements that can be modified when a stairs-rule is added. By default, the stairs-rule looks like displayed in figure 3.12. The displayed rules are the rule-set used for the generation of the spiral-stairs in figure 3.8. There exists a difference to the elevator vertical-connection. This difference is the fact that there is no axiom-rule needed for a real geometry being generated. In figure 3.12, there is no axiom rule attached.

The 'wall-elements' represent the shapes that are generated for the railings of the stairs. The wall-elements of the stairs are both handrails along the stairs unified with the outline of the stairs on the top-floor. Please refer to figure 3.11 (top) for a good example of the wall-element. The whole element consists of the inner handrail, the top part that connects

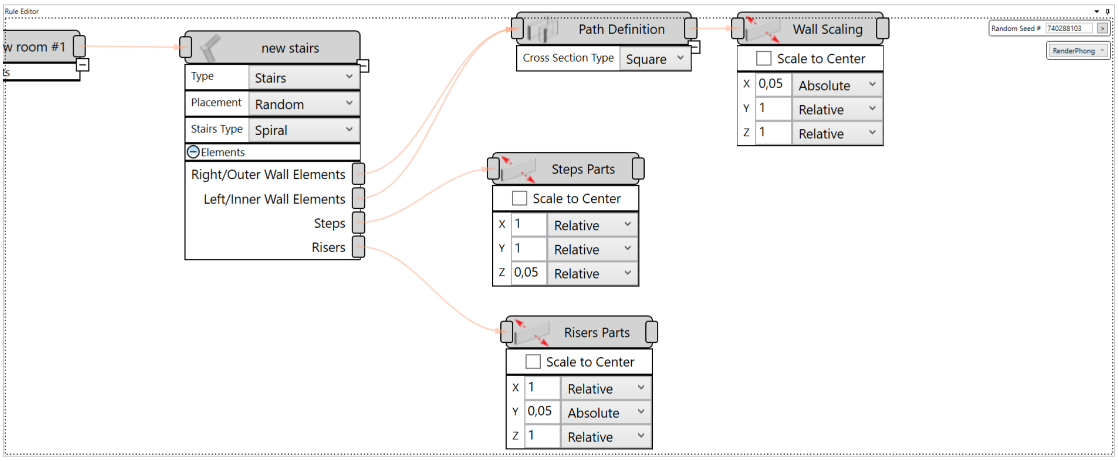

Figure 3.12: The rules that are automatically attached to the stairs-rule for each added stairs in the visual rule-editor.

the inner and the outer handrail and the outer handrail. The walls-elements are the most complex shapes that are generated for the stairs. They are always path-shapes, since the generated outline represents the path of the shape by default and the cross-section is defined as a square with the side-length of 1m. By default there is a path-rule attached to the walls-elements connector. It defines the mentioned cross-section of the railing shapes. Unscaled this would result in a 1m by 1m cross-section of the railing. To change this unrealistic size of the railings there is another rule attached to the path-rule. This scale-rule is used to bring the railing into a realistic shape by scaling the 'thickness' of the shape down.

The steps are maybe the most important elements for stairs. The steps-element is used for the geometry-generation of the shapes representing the steps. In most cases the generated shapes are simple shapes, which means that they are just cuboids. Only when a spiral stairs should be generated, the shapes generated for the steps of the stairs, are polygon-shapes. A simple cuboid does not fit the geometric needs in that case. By default there is a simple scale-rule attached to the steps-element connector. It changes the default size to a more realistic value for the thickness of the shape.

The last elements that can be changed, when working with the stairs rule, are the risers. Unlike the steps, those risers are always just simple shapes, i.e. cuboids. The risers are the step-connecting elements that do not really need to, but may exist at a stairs. The risers are by default connected to a scale-rule to bring the generated shapes into a more usable and more realistic size.

3.6 The Roof Rule

The second specialized rule introduced in this application and master thesis at hand is the roof-rule. Similarly to the vertical-connections, I tried to keep the usage of the rule as simple as possible while allowing as many manipulations as possible. These manipulations can be done by the use of the other simpler CGA-rules implemented. The generation the roof is the last step of the generation-steps performed by the application. The roof represents the top-most elements of the buildings and are therefore handled at last. All previous stages are already handled at this stage. The decision about how many floors should be generated, which rooms should be positioned in them, and how they should be connected, have been made before.

Types of Roofs Three types of roofs, which are most commonly used and which are described in the following sections are implemented in the application. The first and simplest type of the roofs is the flat roof, which can be seen in figure 3.16 (top). The flat roof basically just generates a horizontal roof that only consists of a polygon-shape and some surrounding shapes, which are used to form the boundaries of the roof. The second implemented roof-type is the pent-roof, as seen in figure 3.16 (middle). When a pent-roof should be generated there are a lot more elements available to connect other rules to and an additional value, which defines the direction of where the roof is higher and where it is lower, exists. The third and last roof-type is the hipped roof, which is the default setting, when adding a roof-rule to the visual-rule-editor, see example 3.16 (bottom) for an example. I decided to add these three roof-types because many realistic scenarios of generated buildings can be created with the usage of just these three roof-types.

3.6.1 Flat Roof

The most simple variant of the roofs that can be generated using the procedural-system are the flat roofs. These flat roofs are sometimes used in contemporary architecture for modern residential houses, but they are also the most common type of roofs for bigger buildings like factories and office buildings. Since this roof-type is commonly used, I decided to to integrate this type into the procedural system presented.

A flat roof consists of two main parts, which both can be modified by using the implemented basic CGA-rules. The first elements are the 'wall elements', which represent the possible wall elements that enclose the entire roof. After adding a flat roof to the visual rule-editor, a rule that scales the wall shapes is attached to the wall-elements connector. This scale rule is used to re-size the default-sized wall shapes to a useful and more realistic, but still changeable size. To see the actual geometry of the generated wall shapes, an additional axiom-rule is attached to the output-connector of the scale-rule.

The second element that can be used to modify the look of the roof, is the 'roof' element itself. By default there is a polygon-rule attached to the element's connector.

This polygon-rule is used to define the 'thickness' of the roof-geometry and can be changed to other values.

3.6.2 Pent Roof

Pent roofs are a popular type of roofs nowadays. They are mostly used for residential buildings and commonly used in suburban and rural areas. One advantage of the pent roofs is that they are cheaper to build and construct, than the more expensive hipped roofs and the flat roofs. When a pent roof is created in the visual rule-editor, the available elements change and another set of attached rules appear in the editor. The five elements are described in the following sections and all of them can be modified.

When a pent roof is defined in the visual rule-editor, there exists the possibility to change three main values for the roof. The first value 'default slope' controls how steep the roof should be. A smaller value, which may be more realistic, leads to a roof that is almost flat while a higher value (the maximum value is 89°) leads to very steep roofs. The second modifiable value 'default extent' controls how much the generated roof extends the building-layout, i.e. the outline of the building. The third value represents the angle of the pent roof. A value of 0 represents a roof with the lowest parts 'in the west' and the highest parts 'in the east'. It is a virtual value, which corresponds to an arrow at the origin of a coordinate-system pointing in the specified direction. A value of 'defines' an arrow that points from the origin 'to the right', 90 would result in an arrow pointing from the origin 'to the top', etc. . The imaginary arrow defines the direction of the slope of the pent roof. Of course all values are possible, not only the two mentioned ones. Values that lie between the two mentioned examples (0, 90) result in the creation of quite interesting generated roofs.

The 'wall elements' for pent roofs serve a slightly different purpose than for the flat roofs. To understand the need for a wall element, when working with pent roofs, please see figure 3.13 (top), where a visualization of the wall elements can be seen. The wall element is handled in a special manner both, for pent roofs as well as for hipped roofs. By default there are two rules attached to the wall element connector. This approach is used to be able to control the size and behavior of the complex generated shapes by just using the existing rules. The first attached rule to the wall elements is a scale-rule named 'wall added height'. This rule simply serves the purpose of being able to add a certain height to the roof. If the z-value stays zero, no height will be added to the roof, but if a higher value is inserted, the roof will be positioned at a higher height, and the wall elements will also be taller to connect the roof with the top-most generated floor. In the figure 3.13 a comparison of two different values for this wall added height can be observed. There is no added height in the top part of the figure and an added height of one meter in the bottom part of the figure.

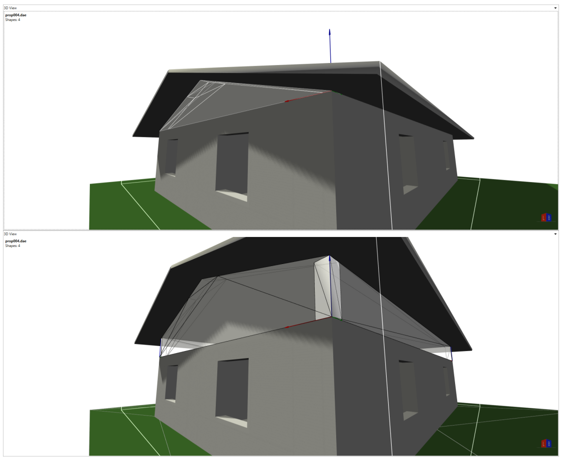

Figure 3.13: Comparison of the different values for the 'wall-added-height'-value. The top image does not have any height added to the walls, while the bottom image uses an added height of one meter. The wall-elements are highlighted. This feature could be used to create an attic for example.

The second attached rule is used for handling the wall-thickness of the generated shapes. Since the roof by default has a slope bigger than zero, the resulting wall shapes are not just simple shapes, which are represented by cuboids, but polygon-shapes and they need to be handled accordingly by using the polygon-shape.

The second element for pent roofs is the roof element. By default there is another polygon-rule attached to the connector. It defines the thickness of the generated pent roof and works similarly to the roof element of the flat roofs.

The ceiling element is the next available element to modify. By default there is no rule attached to the connector, but it is possible to attach a polygon-rule to it. When such a rule is added to the ceiling element connector, an additional ceiling will be generated. This ceiling consists of a polygon-shape which can also be modified.

The fourth element for pent roofs are the purlins. The purlins are part of the roof and are distributed automatically if a 'lines-in-polygon‘ rule is attached to the element's connector. The process of the purlin-distribution can be modified as well. For more information on how the 'lines-in-polygon'-rule works, please refer to section 4.11.9. The created shapes from this rule can then be re-sized and modified in all possible ways.

The last modifiable elements are the rafter elements. To generate the shapes representing the rafters, a lines-in-polygon - rule has to be attached to the element's connector. This element works like the purlins described before, but when they are generated, it affects the wall-added-height - value because there always needs to be a wall-element between the rafter shapes, which is an assumption that was made in the application. This decision was made to keep the resulting buildings as realistic as possible.

3.6.3 Hipped Roof

The roof-parts of the hipped-roofs are calculated by using a modified 'straight-skeleton' algorithm I implemented. An example of the individual roof-parts can be seen in figure 3.14 and a description of the implemented algorithm can be found in section 3.7.3. It allows the modification of the slope and the extent of individual roof-parts, which is very useful if there e.g. is one side or more sides of the building that should not have an eave. By setting the respective slopes of the roof-part to 90° it is possible to generate saddle roofs with the help of the hipped roof system. An example of a saddle-roof can be seen in figure 3.15.

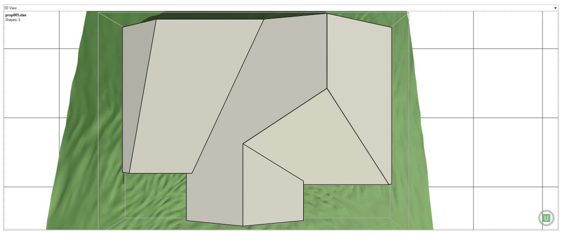

Figure 3.14: A generated hipped roof from above. Some complex roof-parts are generated due to the defined different slopes and extents of some roof-parts. The shape of the roof-parts is highlighted with a border to demonstrate the complex shapes.

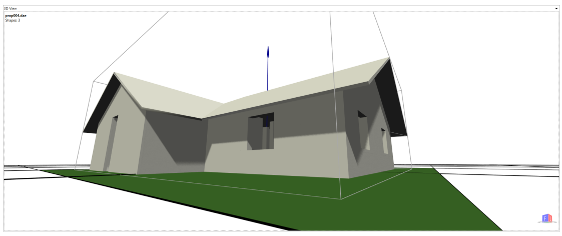

Figure 3.15: By setting the slopes of some roof-parts to 90°, it is possible to create saddle-like roofs for the buildings.

It is possible to set two default values for a hipped roof. The two values control the default slope and the default extent of all roof parts. Since there exist more than just one roof part, when a hipped roof is generated, the default values are assigned to all the generated roof parts. The angle, i.e. the pent direction, is not needed here and is implicitly controlled by the 'base-points' of the roof parts. The two points of the roof-part that have the lowest height and define the eave.

The elements that can be modified work exactly like they work with the pent roofs. A description about them can be found in the previous section 3.5.2. One exception are the purlin shapes. Those generated shapes are calculated for every roof part and since adjacent roof-parts share common 'outlines' two purlins would be created at the exact same position if no special handling would be performed. Purlins which are generated for a single roof-part are always located 'at the outline' of the roof-part-defining polygon. One exception are roof-parts with an extent bigger, than zero. In the case of a roof part with a certain extent, which is bigger than zero, the lowest generated purlin is not located at the lowest possible part of the roof part, but instead it is located above the building wall to ensure a realistic look of the generated roof-parts.

An addition of the hipped roofs is the possibility to change the slope- and extent-values for each roof part separately. The assignment of individual values for a roof-part is possible by double-clicking the roof-part in the 3-dimensional view. A small context-menu opens for the selected roof-part in the 3-d view. After clicking the "OK" button the values are assigned to the roof part. The selected part is calculated by a hit-test check. The values of the individual roof-parts can then be modified in the same way, i.e. by double-clicking it, or by opening the added "Roof Parts" section in the roof-rule. The possibilities that arise by defining different slopes and extents for all roof parts are enormous and result in different styles of the buildings. Combining the possibility to define those values with the possibility to define randomly distributed values gives the generated building a completely different look when a new version is generated. Please have a look at figure 3.17 to see an example of completely different roof styles, which were generated by using the exact same definitions in the procedural system.

3.7 The Most Important Algorithms Used and Integrated into the System

A short introduction into the most important used and implemented algorithms in the application is given next. There are many implemented algorithms which are modified to fit the needs and to be easily integrated into the procedural system. In the following sections, a description on how the algorithms work is provided. For details on how the presented algorithms work and on how the algorithms are implemented and integrated into the procedural system, please see sections 4.11.11 and section 4.11.12.

3.7.1 The Floor Planning Algorithm

The maybe most interesting and also most important algorithm implemented in the application is the algorithm for the floor planning. Since not only detailed façades need to be generated in the work at hand, but also the generation and distribution of all the rooms inside the floors of a building is included, this is a main part of this work.

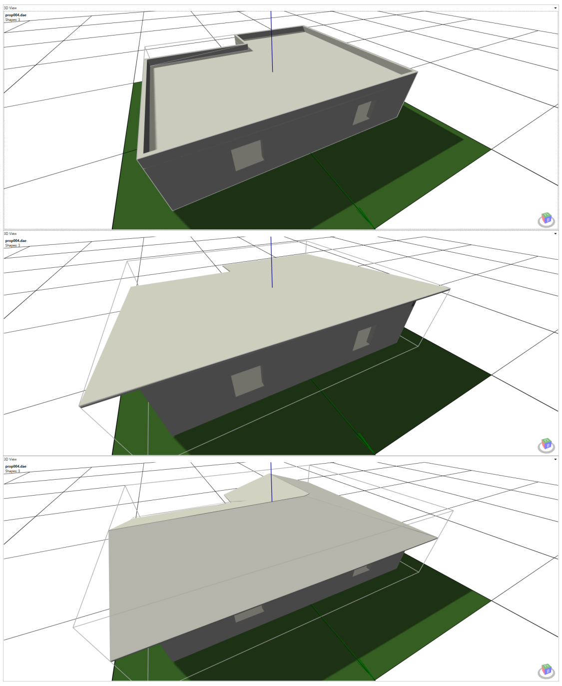

Figure 3.16: Comparison of the three available roof-types in the application. From top to bottom: flat roof, pent roof, hipped roof. No additional rules are attached to the default roof-rules.

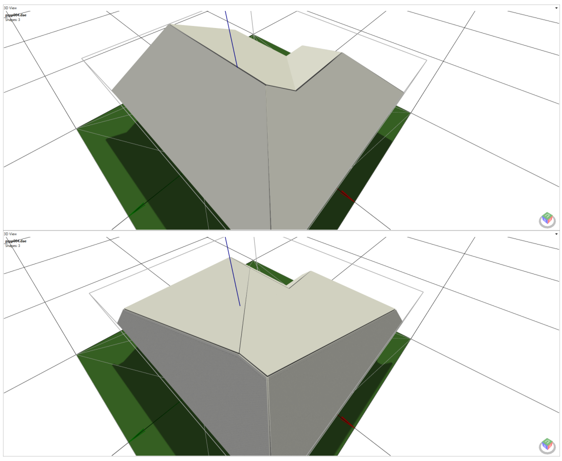

Figure 3.17: Two simple hipped roofs, created by the exact same procedural system definition. The random values for two roof-parts (slope and extent) result in very different roof shapes.

General Description

The algorithm basically works like described in the paper "A constrained growth method for procedural floor plan generation" by Lopes et. al. [LTS+10], but some modifications are made to the algorithm to perfectly work with the developed system. One of the reasons for some of the modifications in the algorithm is the ability to generate better floor-plans and provide a more realistic distribution of the defined rooms. Other parts of the presented algorithm in [LTS$^+$10] had to be adapted to fit the desired properties of the procedural system. A definition of the deep hierarchies of rooms and room-collections needs to be handled by the algorithm.

The Pseudo-Code

The pseudo-code for the subdivision-algorithm can be found in the Appendix B.2. It describes the steps that are performed in the algorithm to solve the problem of realistically distributing the defined rooms and room-collections inside the available space of the floor. A short description of the distinct steps follows below.

The Steps of the Floor Planning

The algorithm for subdividing a given floor is based on the work [LTS$^+$10] and modified and extended to fit the needs of this application. Since a hierarchical structure of rooms and room-collections is given by the user input in the visual rule-editor of the application there is no need for a distinction between 'public' and 'private' rooms for example, which is the case in the mentioned work [LTS$^+$10].

The algorithm uses a 'building-node' as input. This building-node represents the information of the complete building and contains the 'building-rule' amongst other important objects. The building-rule is always considered to be the 'root-rule' of the procedural system.

At first all floor-rules that are attached to the building-rule are used to define how often the outer-most loop of the algorithm should be executed. This means, the following steps are executed for all different floor-definitions, i.e. for each attached floor-rule.

The first step in the algorithm is the initialization step for the layouter-grid. The grid is a very useful class, which allows some simple operations, like assigning a 'room-node' to a grid-cell. From this basic operation, more elaborate possibilities to modify the grid exist.

After the grid was initialized with the fitting amount of cells, the subdivision of the provided floor-plan itself begins. The process works almost exactly like the algorithm shown in the paper [LTS$^+$10] and will be discussed in the following section 3.7.2.

When the subdivision step itself is finished, a few more steps in the generation process follow. At first all generated rooms are determined and using the defined rule-set from the visual rule-editor, all attached rules are added to the corresponding generated room-nodes.

The next step in the generation process is the generation of the 'initial' wall-objects for the current handled floor. The actual floor-nodes are created next. Depending on the amount of floors that were defined in the user-interface only one or many floor nodes are generated. All defined rules are attached to the floor-node, which means the default definitions for the doors, the façade and the floor elements themselves are attached. The floor-nodes are then attached to the building-node.

At this stage all the floors have been generated, the wall-elements exist and the rooms, the floors and the building itself contain all defined rules. The next steps of the generation algorithm is used to position the vertical-connectors inside the floors.

For each floor in the building that was generated with a 'new' floor-rule, i.e. the first or lowest of possibly many equally subdivided floors, all defined vertical-connectors are positioned. After the positioning-step, the modified objects are updated because it is possible that e.g. rooms change their shapes.

3.7.2 The Subdivision Algorithm

One imporant part of the floor planning algorithm is the subdivision of the available space in the floor. To provide an overview over the chosen solution, the basic steps of the algorithm are described in the following parts of this work.

The subdivision-algorithm is always executed after the layouter-grid was initialized. The input to the algorithm is always a room, which can also be a room-collection - if the room 'contains' more than one 'sub-space'. The initial step in the algorithm is to create the actual child-room-nodes for the generation-graph.

If the current input-room-node is a room-collection, at first the start-positions for the room-expansion on the floor are calculated. Those start-positions are located next to the outer border of the floor-outline and the distances between the start-points correspond to the relative size-definitions of the rooms. This means that two big rooms have start-positions that are located farther away from each other, than two smaller rooms because the resulting sizes of the rooms will be affected by the start-positions of the room-expansion.

After all the start-positions were calculated, the affected grid-cells, i.e. the grid-cells at the start-positions, are determined and assigned to the corresponding room. To ensure that the rooms are never too small, even if there are many rooms defined in one floor, the initial assignment not only assigns one grid-cell, but also it's neighbors to the room. The size of the 'start-area' is dependent on the room because in the application the 'connector-room' starts with an area of $2m \times 2m$, while the 'normal rooms', i.e. the connector room's siblings, start with an area of $1.5m \times 1.5m$. See figure 3.18, sub-figure 1, for a visual explanation about the start-points. This distinction of different start-areas is useful to ensure the connectivity of the rooms with the connector-room (a minimum width of $2m$ is assumed to be enough for corridors.

The next step is the expansion-step for the rooms. The expansion-strategy randomly selects a room and a direction out of the possible expansion-directions for that room to grow the rooms iteratively. See figure 3.18, sub-figure 2, for the visualization of one expansion step of a room. The possible expansion-directions for all the rooms are determined between the expansion-iterations by checking if the expanded room is already located next to another room on one of the four sides. If the room was expanded until it is adjacent to the floor-outline or another room, the affected direction is removed from the possible expansion-directions. The selection of the rooms to be expanded is done randomly, respecting to the current size of the room and the desired relative sizes of all rooms. The expansion is randomized but still respects the desired size-constraints this way. When no room has an allowed expansion-direction left, see figure 3.18, sub-figure 3 for an example, all the remaining areas are assigned to neighboring rooms, in a way that minimizes the quotient of the outline of the room and the area of the room, i.e. the rooms are as 'square' as possible, see figure 3.18, sub-figure 4 for a visualization.

When the room-expansion step is finished, all the area of the floor is assigned to one of the defined rooms in the floor. Since one of the design decisions of this application was to define the room-hierarchy in the visual rule-editor one of the child-rooms of a 'parent'-room is considered to be a 'connector-room'. This means that all 'siblings' of this room need to be connected to this room. This decision ensures connectivity throughout all rooms in a building as well as the possibility to implicitly define the hierarchy of the rooms in the visual rule-editor.

To ensure the mentioned connectivity between the 'connector'-room and it's siblings, an additional step is performed in the generation process. This step calculates the adjacency, i.e. the neighbor-information between the generated rooms. For all rooms which are not adjacent to the connector-room, i.e. there is no possibility to create a connection between the two rooms, the shortest path between the two rooms is searched. This shortest connection-path is always located at the borders between the rooms to avoid splitting a room into two parts. The shortest path is then expanded, until a minimum width is reached, which ensures the possibility to create a room-connection, and assigned to the connector-room. This step adds 'corridors' to the connector-room until all sibling-rooms can be reached from the connector-room. After this step the layout is complete and two more steps follow. Please see to figure 3.18, sub-figure 5, to see how the described connection-creation looks like in an example.

The actual layouts of the generated rooms are calculated. This layout stores information about the room's outline. The last step in the subdivision algorithm is the computation of the connection-positions. These connection-positions are later used to generate the connections between the rooms.

For all child-rooms the subdivision-algorithm is then executed again to create a further subdivision of the areas of the floor, see 3.18, sub-figure 6 for an example. The grid-cells which are assigned to the room-collection are reset before the child-rooms are distributed in the area.

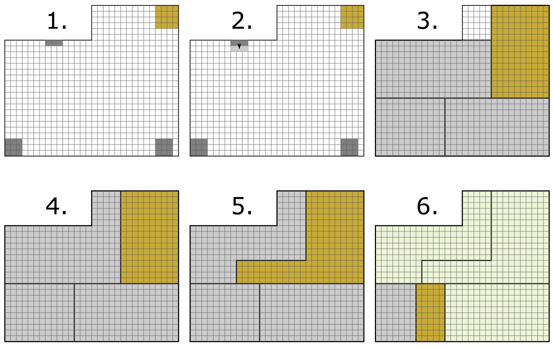

Figure 3.18: Visualization of the most important steps in the floor-planning algorithm. Yellow areas represent connector-rooms, gray areas other rooms and green areas, fixed rooms from previous subdivision steps. In this example the definition of the floor contains four rooms, one of them contains two child-rooms. Sub-figure 1 demonstrates the situation in the layouter-grid, after the start-positions of the defined rooms were calculated and expanded. One room was placed outside the actual floor due to numerical errors, but this is not a problem. Sub-figure 2 shows one step of the room-expansion. One of the rooms is selected and expanded in an allowed direction. Sub-figure 3 shows the layouter-grid after all expansion steps were performed, not all areas were assigned to a room yet. Sub-figure 4 shows the complete assignment of grid-cells to rooms after all unused areas were also assigned to adjacent rooms. Sub-figure 5 shows the room distribution, after all rooms were made accessible from the connector-room. The last sub-figure shows the subdivision of one of the previously generated rooms into two child-rooms. The room, which is connected to the first connector-room is the connector-room in this subdivision step, while the other room can therefore only be accessed through this room.

3.7.3 The Hipped Roof Generation Algorithm

The second very important algorithm that is implemented in the application is the creation of the roof, specifically the hipped roof. This roof-type results in the most complex roof-geometry and also creates more than just one roof-part. The subdivision of the area of the roof into the roof's individual segments, i.e. the individual polygons, can be achieved with a 'straight skeleton' algorithm. A description about how the algorithm works and how it was modified to fit the needs of creating actual hipped roofs is shown below.

General Description

The creation of roofs is achieved in a few steps that don't really change, no matter what type of roof should be generated. The hipped roof generation is slightly more complicated but is discussed in more detail in the following sections.

The first step in creating a roof, is to to retrieve information about the roof-shape. It equals the outline of the building and any modification to it will be added later, e.g. through the extent of roof-parts. In the next steps, the roof-part setup is performed. This setup is not really needed for flat-roofs and pent-roofs, but essential for the hipped roofs, where it is possible to assign different slopes and extents for every roof-part.

After the setup is complete, a distinction between the desired roof-type that needs to be generated, different further calculations are used to create the roof. The basic steps are the same for all types, though. At first the geometry of the ceiling of the top floor is generated if fitting rules are attached to the 'Ceiling'-element of the roof-node in the visual rule-editor. This geometry equals the actual roof-geometry if a flat roof is generated.

After generating the ceiling-geometry, the roof-geometry is generated. Pent roofs use the building outline for creating the roof outline. This roof outline can be bigger in size because a defined extent-value for the roof is taken into account in this step. The outline is then stretched according to the 'roof direction', i.e. in which direction the roof is headed, and it's slope. The higher the slope, the bigger the stretching of the base outline. By calculating the rotation of the roof by using the defined direction and slope, the generated geometry is positioned. The calculation of the geometry of hipped roofs is discussed in more detail in the following section.

Additional geometry for the roofs is generated at last. For flat roofs this can be an outer wall geometry surrounding the roof. Please see figure 3.16 (top) for an example. For the pent roofs and the hipped roofs there exist more optional geometries that can be generated. These additional roof-parts are the purlins and the rafters. The base purlins are always located exactly above the lowest parts of the roof or roof part that is positioned inside the building-outline. Pent roofs only have one base purlin and one top purlin located at the top-most parts of the roof above the building outline. Purlins in between are distributed according to the attached rules. The base purlins and top purlins for hipped roofs works exactly in the same way, but there are additional purlins created for all roof parts at locations where the roof-part is adjacent to another roof-part. This means that, except for the base purlins, the outline of a roof part is surrounded by purlins, which leads to realistic structures below the roof. See the black highlights in figure 3.19 for a complex example of the purlins created. The rafters of the roofs are positioned perpendicular to the purlins and are distributed across the width of the roof-part according to the attached production-rules and are highlighted in red in the same figure.

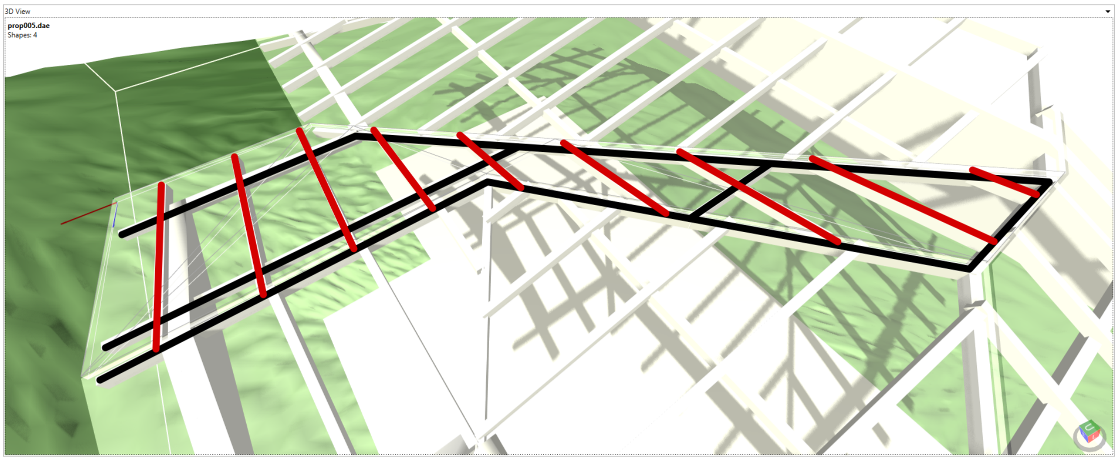

Figure 3.19: Complex additional roof geometries. The purlins are highlighted in black for one roof-part, the rafters are colored red.

Straight Skeleton

The straight skeleton algorithm was developed to create a subdivision for a polygon, which only contains straight boundary-elements. This subdivision, compared to Voronoi-diagrams, do not use a distance metric, but uses a shrinking process to calculate the polygons. See [AAAG95] for a description. This shrinking process contracts the boundary of the polygon by moving the vertices of the boundary along the angular bisector of its incident edges. This shrinking process continues until a change in the topology of the boundary occurs.

One modification in the application, compared to the original algorithm, is the use of different directions for the vertex-movements in the shrinking process. If two adjacent roof-part's slopes of a boundary vertex are equal, the calculated direction of the vertex movement, i.e. the 'ridge-direction', equals the angular bisector, but if the two roof-parts have different defined slopes, the ridge direction is different. Figure 3.20 demonstrates the effect, different slopes defined for adjacent roof-parts have on the direction of a ridge of the roof.

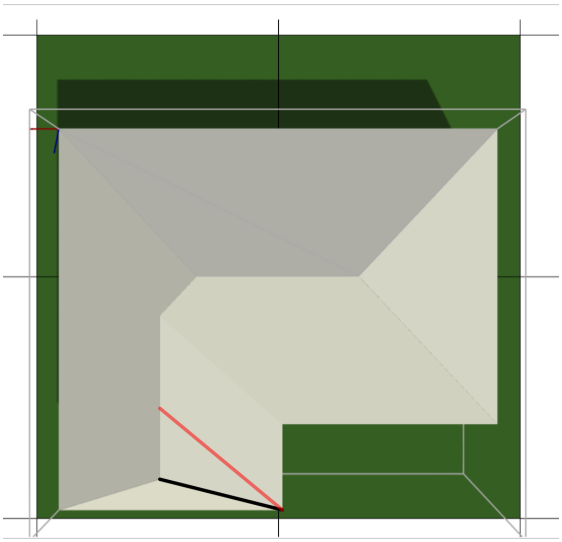

Figure 3.20: Different slopes result in a ridge direction (black line), which is not equal to the angular bisector for the boundary vertex. The angular bisector is laid over for a better comparison (red line).

Two types of possible events that cause a change in topology exist when shrinking the boundary of the building-outline. The first type of event is a so-called 'edge-event'. Edge events occur when an edge of the shrunk boundary shrinks to length zero, meaning it does not exist for further shrinking operations any more. This means that the two neighboring edges of the affected edge become adjacent themselves. The second event that can occur are the 'split events'. A split event is the event, when a reflex-vertex 'splits' an edge, thus splitting the polygon into two or more sub-polygons.

The straight-skeleton algorithm is useful only for roofs with equal slopes defined for all roof parts. The application allows different slopes for all roof parts to improve the abilities, though. A change in how the algorithm works was developed to account for the different slope-definitions. The generation of the roof-parts in the application starts with the calculation of the actual ridge directions. These ridge directions are accounting for the different slopes of the adjacent roof parts. The directions are represented by two-dimensional vectors and the length of those vectors are different, depending on the slopes. The length of the ridge direction vectors represent the needed projected two-dimensional movement of the corresponding boundary-vertex along the ridge, so that the boundary vertex of the shrunk boundary would be positioned 1m above the previous boundary. This means that the values of the ridge direction vectors are bigger if the slopes are smaller and vice versa.

The calculation of the straight skeleton in the application is explained next,please also refer to Appendix B.3 for a the pseudo-code of the algorithm. The input for the calculations is the building object, which contains the building layout, i.e. the boundary for the straight skeleton algorithm, and all defined values for the individual roof-parts, i.e. the slopes and extents.

The algorithm starts by iterating through the steps discussed next and stops when no shrunk layout, i.e. no boundary, exists any more. Each iteration starts by choosing the next boundary to shrink in the current iteration of the calculation.

After the roof parts are set up and the ridge directions were calculated, all events for the current boundary are calculated. This calculation is performed in two steps. At first all edge events are calculated and saved, including the height of their occurrence. The height of the occurrence is important later to be able to sort the events and the height calculation is easily performed due to the special ridge direction vector properties. The second part calculates the split events for the current boundary and is a little bit more complex. Split events can only occur for reflex vertices, i.e. for vertices that cause the boundary to be non convex, so at first a check for 'reflexivity' is performed. The reflex vertex is then tested for an intersection with all non-adjacent edges of the boundary and the possible positions and heights are again stored as possible split-events. Since the application allows roof parts with a defined slope of 90°, this special case leads to another small rise in the complexity for the split-event calculations. After all events were calculated, they are all sorted with respect to the height of their occurrence from lowest to highest.

The events with the lowest value of the height of occurrence are used in the current iteration of calculation. Those events are stored in the roof part - objects for the generation of the geometry later. The height of the active events is then used to shrink the boundary using the ridge directions. This can also easily be done because of the special ridge direction vector properties.

The next step to calculate the straight skeleton for the building is to actually handle all the active events, i.e. all events of the current boundary that occur at the same minimal height. At first the split events are handled by splitting up the current layout into more separate layouts if necessary. The separate layouts all share one common vertex, the vertex where the split-event occurs. At last the active edge-events are handled. Two vertices of the shrunk boundary become one vertex for the further iterations of the calculation.

All remaining shrunk boundaries that have an area of zero are removed from the set of boundaries that need to be handled in the next loop-iteration. This is done because all calculations for those boundaries were performed already and no more events can occur, i.e. the roof-parts reached the maximum height in that building-area.

After all the events were calculated and stored for all roof parts, the actual generation of the roof part geometries starts. This geometry generation basically works by creating a polygon for each roof part by using the calculated events and then stretching the two-dimensional representation of the resulting geometry according to the slope of the roof part. This stretched polygon is then used to create the final polygon-shapes, which are then rendered to the screen.

CHAPTER 4

Implementation

This chapter focuses on the created application and especially the procedural system itself. I will describe some important properties of the system as well as the steps that are needed to create a nice building from the start, i.e. from the base-floor to the end, i.e. the roof and additional details. The following descriptions start with an overview of how the application is organized.

Some file-formats that are used for importing a 'scene', i.e. a property, which describes the space conditions for the building-generation or a previously created building followed by the output of the application are described next. There exists the already mentioned possibility to save a 'scene', completely with the property and the whole created procedural system. A brief description of how the application can also export the generated building in a file-format which is widely used is also given.

After the file-formats section, an explanation of the main and most-important data-structures follows. It gives an overview of how the data is structured and how it is used in the procedural-generation process.

In the next section some details about the implementation of the user interface are outlined. An explanation of how the created visual elements relate to the rules of the procedural system and how some of the main features regarding the visual rule-editor and other user-interface parts is presented.

A description about the procedural system and how it is used in the application follows. The derivation-process as well as the hierarchical nature of the system is discussed also.

4.1 Application Organization

The application consists of many small parts that need to work together well, to provide the desired results. To avoid a big and hard to manage application, the individual parts were split into several projects. The grouping of the many classes into those projects was achieved by grouping together classes that serve a similar purpose and work together.

There are ten different projects in the solution, not counting the external libraries, which are used to perform special computations. Those ten projects are again grouped to four main application-parts. Each application-part contains only projects, which make use of the other contained projects a lot.

4.1.1 Core Program

The core program consists of five projects, one of them being the main project, which is compiled to an executable file. The other projects are working together with the main project a lot and are described in the following sections.

Constraints

This project is not really used widely throughout the application. One exception are the RoomViewModels, which contain a set of "RoomConstraints". The RoomConstraint class is used to create the individual elements for RoomNodes, which can be edited in the visual rule-editor. Examples of constraints is e.g. the size of the rooms, the amount of rooms to generate and the indicator if the room is a "ConnectorRoom". Since RoomViewModels are also used to create the "FloorNodes" some other constraints are attached to the rule when it is directly attached to the property node. The constraints have an "Argument" of a certain type. There exist some basic types like the "BooleanArgument", the "NumericArgument" or the "RangeArgument". All the argument classes are implemented in the Constraints project.

Generator

This project implements all the different classes that perform generative calculations in the application. Some very important types of classes are grouped in this project and are discussed below.

The node classes are used to automatically build and iterate through the procedural system. There exists an abstract base class "GeneratorNode", which basically only defines some properties that should be usable, no matter what real sub-class the procedural system is actually dealing with. The different sub-classes of the base-class extend the available properties that are needed for the special types of nodes and implement some helper functions, which are e.g. needed for positioning purposes.

When the generation of a part of the building is started, the needed parts of the visual representation of the procedural system, i.e. the ViewModels, are used to create the actual GeneratorNodes, which are then used to process the procedural system. The creation of the floor planning is a good example for this approach. In the defined procedural system, which contains a property rule, some floor- and room-rules and some other rules that affect the results, only the floor- and the room- nodes are attached to the building node. This approach creates a smaller tree than if all attached rules, i.e. including the CGA rules, were added to the room rule at the beginning of the generation process. In later generation steps, fitting rules like a new façade definition, or an individual floor definition are attached to the floor- and room-nodes.

All the CGA-rules are handled with the RuleNode objects. This rule-node class defines some additional properties, which are necessary e.g. to be able to access the rule it represents, and an “ApplyRule“ function. This function basically represents the procedural generator because it takes a shape as an input and applies it’s rule to it. The function is then recursively called for all generated shapes that were created by the application of the rule to the shape, which represents the iteration through the procedural system. All generated shapes from the recursive function calls are added to the renderer and to the rule node at last if an axiom rule is attached somewhere in the generation process.

The floor planning classes, which are used to generate the layout the floor and position the vertical connectors are also implemented in this project of the application. The most important class of this type is the ”GeneratorManager“, which is the main generator class. It is used to start the building generation process and calls the individual functions which handle the different parts of the generation of the building. It uses all the other classes that are part of the floor planning classes and uses other generators as well.

At first it uses the ”BuildingLayouter“ to generate the distribution and base geometries of the rooms and floors of the building. The next step is the placement of the room connections and the modification of the affected geometry, followed by the application of all façade manipulations and the generation of the roof.

Manipulators and geometry generators. The geometry generation is executed in some different stages of the building generation process. The first generation of geometry is performed after the floor planning and the positioning of the vertical connectors is finished. Those initial geometries are refined and modified in the later steps of the generation process and those changes and modifications of the geometry are performed by the rest of the classes in the generator project.

Models

The classes in this project are used for handling the data for several different types of objects used in the generator project. The "WallModel" is used for all basic generated walls along the outline of the rooms for example. The "RoomModel" holds data needed for the creation and modification of the rooms and the "LayoutModel" and "LayoutViewModel" classes are used to store the layout information for the rooms and provide some useful functions and properties.

Procedural Buildings

This is the main project of the application and is compiled to an executable file. The definition of the look and the behavior of the main window of the application is defined in this project as well as the "RenderViewModel" and the "ViewportViewModel".

The MainWindow classes define the user interface of the application, the menu and event handler for all user interactions. The ViewportViewModel is the base class for the RenderViewModel and it is used to store information about the scene of the 3-dimensional view by defining some needed properties. The RenderViewModel is the sub-class and again extends the base class with many additional properties like the collection of the 3-dimensional objects that should be rendered. The RenderViewModel is used in the MainViewModel and is used as the DataContext for many of the individual parts of the MainViewModel like the 3-dimensional view. It implements a lot of functions, which are needed e.g. to add or remove a shape to or from the visual output.

Rules

The last project in the Core Program part is also one of the most important one's. The classes implemented in this project are all directly related to the production rules of the procedural system. The three types of shapes are implemented in the shape file and are used in the generation processes of the application.

The condition classes are not used in the application yet. As previously mentioned in the work of Prusinkiewicz and Lindenmayer [PL90], there exist "conditional production rules" that check if the rule is applicable to the shape before it is executed. The two implemented classes demonstrate how an implementation of this functionality could be used in future releases of the application. In the abstract base rule class the conditional is checked at first when the "ApplyRule" function is called.

The rule classes are the most important classes, which are implemented in this project. They are described in more detail in the sections 4.7.3 and 4.8.

4.1.2 Helper

The helper application part contains only one project, which is also called helper.

Helper

The helper project is used to define some of the widely used base classes and includes some static classes that are used to implement extension functions for several classes as well. The helper project has no dependencies on any other implemented project, except external libraries, which are used for some calculations.

Base classes like the "ValueElement" are defined here. These objects are used in the user interface as well as in the rule classes for example. Another example is the RNG, i.e. the RandomNumberGenerator, which is a static class used in many different other classes. The "BaseViewModel" is implemented in the project as well as all special enumerations and value converters, which are used in the user interface.

Extensions are also defined and implemented in this project to support the use throughout the application. There are four classes that implement extensions for e.g. the external "ClipperLibrary", the "HelixToolkit's" meshes, converters for colors and other extensions that e.g. add functionality to the HelixToolkit vectors.

4.1.3 IO

The IO[1] part of the application currently consists of only one project. This project is used to save and load the scene information to and from XML-files.

ProjectIO

The ProjectIO enables the application to load and save project files. The "SaveProject" function in the static IOCore class saves all the scene information and all defined rules including their settings and connections to the specified file. The scene information consists of the current RandomNumber of the scene and the information about the camera position and direction it is pointing to. The procedural system is saved by calling the "SaveToXML" function, which every node implements and at last saving the connectors and nodes of each connection, so they can be restored when loading the file again. The "LoadProject" function makes use of reflection techniques to create the different types of ViewModels from the saved information. At first the scene information is loaded and the information about the ViewModels and the connections afterwards.

4.1.4 NetworkView

The network view part of the application consists of three very important projects, which work closely together to create the visual rule-editor. This network view is based on the work [Dav12] by Ashley Davis. It provides the basic mechanisms of the visual rule-editor and was extended and modified to fit the needs of the developed application.

[1] **I**nput-**O**utput

A visualization of the hierarchy of the ViewModels in this project, augmented with the BaseViewModel from the Helper project, can be found in figure 4.1.

NetworkModel

The network model project consists of a number of classes that all implement the view model component of all the MVVM elements in the application. The "NetworkViewModel" class implements the viewmodel for the visual rule-editor and contains collections of all the ViewModels, i.e. the room ViewModels, the rule ViewModels, etc. , that form the procedural system. The other ViewModels that directly derive from the base viewmodel define the logic for some additional user controls. The "NodeViewModel" also derives from the "BaseViewModel"-class and it defines several variables and properties that are used by the sub-classes and also the procedural generator. The NodeViewModel classes are visualized in the top right of the figure 4.1.

The most important sub-class of the NodeViewModel class is the "RuleViewModel" - class, which itself is the base-class for all other rule-related view models, which are grouped together at the bottom right of the figure. The rule ViewModels additionally contain logic, which makes it possible to interact with the actual rules that are used in the procedural generation steps.

NetworkUI

Some predefined classes are found in this project. The appearance of the connections between the nodes of the visual rule-editor is defined in the "Arrow" class for example but also the nodes appearance is defined here. The classes defined in the "Views" folder all visually define the user controls mentioned in the NetworkModel project. The ViewModels serve as the DataContexts for the user controls and the DataBindings are defined in the View classes in this project. By using DataBindings it is possible to automatically update a value in a viewmodel for example. The views defined in this sub-folder are used to represent the ViewModels in the visual rule-editor.

NetworkUtils

This is an assisting project for the two other mentioned network-projects before. One class to mention is the "ImpObserveableCollection", which defines a collection that makes it possible to add and remove a whole range of elements for example. It automatically notifies all EventHandlers about the changes of the collection and allows the proper handling of these events. Another important helper class is the "WpfUtils" class implemented here. It makes it possible to retrieve information about the visual parent of a child element and to perform a hit testing for the nodes in the visual rule-editor. This hit test is used for the selection of nodes in the visual rule-editor for example.

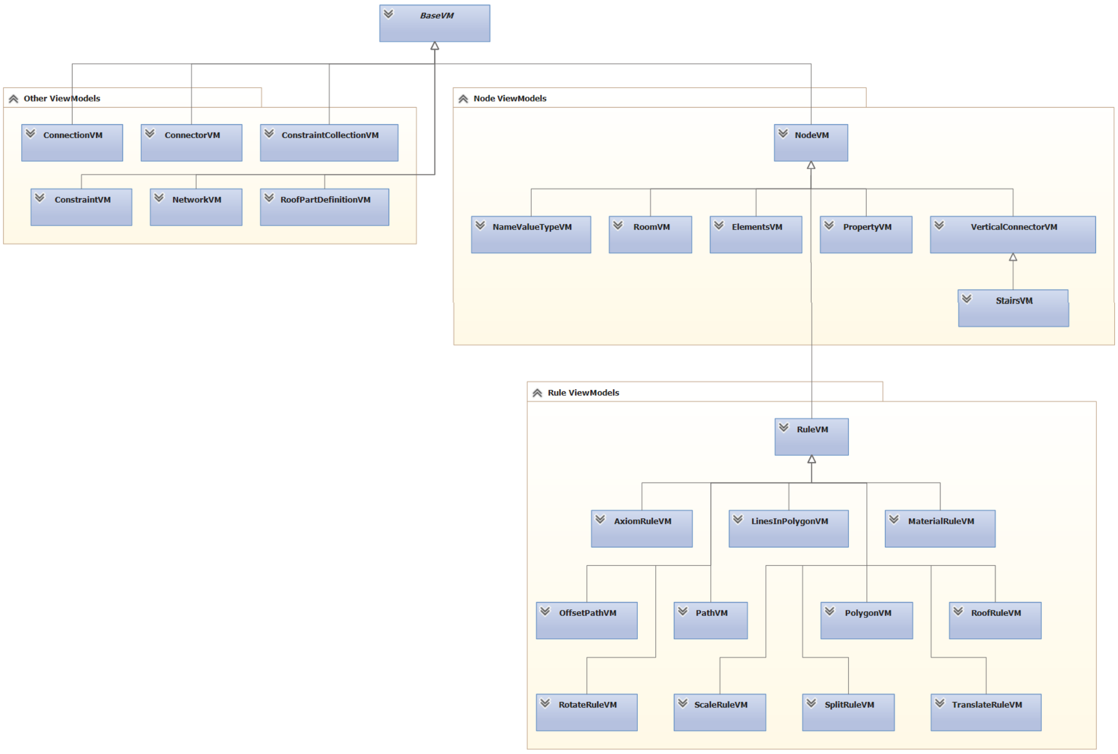

Figure 4.1: Visualization of the hierarchy of the viewmodel-classes implemented in the system. The base class is located at the top of the figure. Some classes that directly derive from it are grouped together at the left side of the figure. The group at the top right are the node ViewModels which are all represented visually in the visual rule-editor. Another derived class from the node viewmodel is the RuleViewModel. It is the base class for all other rule ViewModels and the classes are grouped together at the bottom right of the figure.

4.2 Data Formats

In the implemented application a variety of file- and data-structures are used. Some of the most important data-formats used are described in the following sections. At first a section about all the used file-formats in the application and then an overview over the most important data-structures is given. Some of the most helpful features and properties of those data-structures are discussed as well.

4.2.1 Property Input File Format

Before the application can be used to generate buildings, there is the need to load a previously created 'scene', i.e. the property that should contain the building. This input is created by one of the many available 3d modeling tools and applications. For testing the application I used the tool 'Cinema4D' from 'MAXON', but any other application that is able to save a DAE-file is working. Since the 'Collada' file-format is open-source there are a lot of free tools available to create such a property file for the application.

There are two important assumptions that were made to simplify the import of those DAE-files into the application. The first one is that there must be two objects present in the import-file. One object must be the child of the other object. The child element defines the area of the building while the other element represents the surrounding. The second assumption is that the positive Y-direction of the imported objects has to point downwards.

If both of those assumptions and simplifications are met, there is no limit on the shape or size of the scene, some rather big scenes were tested.

4.2.2 Scene File for Saving and Loading Projects

To be able to store a created procedural system completely, a simple XML-file format is used. This file only saves the properties of the elements shown in the visual rule editor, which is equal to saving the procedural system itself because it is defined by the elements in the visual rule-editor.

At the beginning of the XML-document there is the basic information about the project. This basic information consists only of the value of the random number generator at the moment of saving the scene and the information about the camera, i.e. it's position and view-direction. After the basic scene-information, all the 'nodes' in the visual rule-editor follow. These nodes represent everything that was defined previously. There is a node for every rule in the procedural system and additionally there exists a property-node, which does not relate to a rule in the procedural system. The room-nodes are also treated in a special manner. After the description of all the nodes in the system, a list of 'connections' follows. These connections are very important because they define the hierarchy of the nodes. A connection is defined by the information about the two nodes it connects, the source node and it's connector-element as well as the destination node and it's connector-element. The source node relates to the predecessor and the destination node relates to the successor of a textual production rule.

The XML-file was developed in a way it makes it easily readable by a human. When such a file is loaded into the application at first the random-number-generator is set with the defined "SeedNumber“ and then all nodes are inserted. By loading the property-node the defined scene-file (a Collada-file) is also loaded automatically.

4.2.3 3d Output File

It is also possible to export the generated building to a 3d object-file to save the generated geometry. I wanted to be able to work with the files in a wide variety of tools, therefore it was decided to use the simple and well-supported OBJ-file format. Since I use a framework that already implements such an exporter this was an obvious decision, but some of the code had to be rewritten to actually work.

4.2.4 Important Data Structures

In this section I will discuss some details of how the most important data-structures are working and what the benefits of those discussed data-structures is. I will start with how the grid for the room-planning part of the application is structured and how it works. Afterwards a description of how the nodes in the visual rule editor are implemented and step though all details down to the rules.

The Grid Data-Structure for the Room-Planning

The room-planning algorithm is one of and most important implementations in the master thesis at hand. It determines how realistic the distribution of the rooms in the building and the floors is. For the algorithm "A constrained growth method for procedural floor plan generation" developed by Lopes et. al. [LTS+10] to work properly, a special data-structure had be be implemented. Since the room-planning algorithm is a modified version and therefore it is grid-based, I chose to implement the grid itself by using an array of grid-cells. A description can be found in section 3.7.1 and 3.7.2. The decision to use the mentioned technique was made to keep things as simple as possible and to allow the definition of a modified **indexer** for the grid-cells inside the grid. This modified indexer uses two indices, one for the x-index of the grid-cell that should be read or set and one for the y-index. It ensures that all indices stay in a valid range from zero to the maximum allowed value. The range of valid values is determined at run-time and depends on the size of the building-layout.

Some other important features of the grid itself are some special variables, mentioned in section 4.13, and the fact that the grid itself is responsible for the assignment of the rooms to the grid-cells. The algorithm is described in detail in section 3.7.2.

The 'grid-cell' is the basic data-structure which is used in the 'grid'. Each cell of the grid holds information about the assigned room, i.e. the room it the cell belongs to, if there is already a room assigned at all or if the grid-cell is even 'inside' the layout at all. It is possible that the grid-cell is inside the 'axis-aligned bounding-box', without being inside the building because not only rectangular building-layouts are allowed. Some more information about the grid-cell is stored. One additional example is if the grid-cell is part of a room-border or if the grid-cell is part of the grid-border or both. The grid-cell is part of the room-border if there is an adjacent grid-cell in the grid that does not belong to the same room as the current grid-cell.

The most important feature of the grid-cell other than holding the previously mentioned data, is the 'Neighbors4' method. It returns the '4-neighborhood' of the current grid-cell, which means that the result is a list of usually four grid-cells. The default behavior is to return the top, right, bottom and left neighbor if there is one (in this order). If the optional argument for the function-call is set to true, only the diagonal neighbors are returned which makes it possible to check the complete '8-neighborhood', i.e. all eight adjacent grid-cells, of the current grid-cell.

The Nodes, from the User-Interface to the Data-Layer

As already mentioned in section 3.1.1, I decided to implement the MVVM design pattern to be able to structure the classes and their data in an easy to manage manner.

The nodes, i.e. the elements that are visible in the visual rule-editor, are basically the view-models of the depending views. This view is created on the fly by the theming-system of the WPF. It basically works by predefining a so called 'UserControl' in the XAML language [Smi09]. In this XAML-file it is possible to define all data-bindings for values and other elements displayed in the UserControl. By using data-binding, every time a value is changed by code, the element that is bound to that value is updated automatically. The other way works also, so if the user changes a value by inserting another value-string or by changing the selection of a combo-box, the values of the node are used to update the procedural system automatically. If the user changes a value, which defines a size in the procedural system and the node is actually connected to the procedural system, which is not necessarily the case, the generation process will be started again, and the changed value triggers an update of the 3-dimensional view as well. By using this technique, it is always ensured that the two views of the procedural system are synchronized, i.e. the view of the rules of the procedural system and the 3d-view of the generated building.

Each view-model of all the available nodes implement an interface which defines the two basic methods for loading and saving the view-model. All view-models are arranged in a hierarchical manner, which means that there is a base view-model class for the nodes of the visual rule-editor. This base class is also responsible to save and load the basic information of each view-model like the node's name and it's position on the 2d-canvas of the visual rule-editor. All other values of the derived view-models are handled by the respective view-model classes. The view-models themselves are the main parts of the data-side of all the nodes. They provide all the properties, which are needed for the view to create a data-binding to.

For those view-models that are used for defining the production-rules of the procedural system, there exists a derived class from the base view-model class that serves the purpose to represent the base-class for all explicitly implemented rule view-models. It provides a few additional variables and properties like the rule that is handled by this view-model and a list of shapes that were generated by applying this rule in the generation process. This list of shapes helps with the re-generation of the building when a value-change in this rule occurs.

The Rules in the Procedural System

The rules that are used in all derived rule view-models form the models in for the MVVM-design-pattern. All values and settings that are set by the user are stored there. The rules themselves are also arranged in a hierarchical manner, which means that there exists a base rule class that defines the basic information of a rule in the procedural system. This base-class also defines a virtual function that is used to apply the defined rule to a given shape. All deriving classes of rules implement this function and handle the shapes differently, i.e. split the shape, transform the shape, add additional attributes to the shape, etc. .

The Shapes

The shapes that are used in the derivation steps by applying a rule to a shape are also defined as a base-class and derivations of this class. The base-class is only a wrapper for a 'GeometryModel3D' object that was already implemented in the HelixToolkit, see section 4.9 for more information. The basic shape class is used for most of the 3-dimensional objects that are generated and is geometrically also the most simple one. The shapes are just container-objects at the beginning of the generation process. It contains a transformation matrix and initial settings for it's appearance, no geometry-information is added by default. If the previously 'empty' shape get's assigned to an axiom-rule, which adds the actual geometry to the shape, the result is a cuboid, which is positioned somewhere in the scene.

Since the simple shapes are not sufficient for all building-elements, the so-called 'PolygonShape' class is implemented, which derives from the base 'Shape' class. They are needed in the system because not only cuboids, but also more complex geometries occur in buildings. A PolygonShape uses two additional informations of the objects to generate, namely an "PolygonOutline“ and a "Thickness“. The outline is used to store information about the shape of the object to generate. The thickness is used for object-thickness. With just these two additional values is is possible e.g. to define the geometry needed for roof-parts and other non-cuboid elements and objects of building-parts like the room-floors which are not simple cuboids in most cases. Section 3.6.3 and figure 3.14 contain for more information about the hipped roofs.

The third implemented type of shapes are the 'PathShape'-objects, which are derived from the PolygonShape class. A PathShape again introduces a new attribute, which is used to generate a much more complex geometry than it is possible with the previously described shape-types. The PathShape-objects are mostly used by, and were introduced for, the stairs of the buildings. See section 3.5.2 for more information about the stairs. The additional attribute is called 'Path'. A given cross-section 'follows' this path, thereby creating complex geometries. The cross-section of the object is defined using the PolygonOutline - property from it's base-class and a list of 3d vectors that form the "Path" of this PathShape.

4.3 The User Interface

The user interface is designed and implemented by using the WPF and enables us to influence a lot of elements of the UI[2] using the mouse and the keyboard. All the individual windows and window-parts are dockable and the layout of the parts of the user-interface is customizable.

4.3.1 The Menu

The menu makes it possible to interact with the program directly. The most important commands that are available in the main-menu is the loading of scenes and properties and the saving of created 3-dimensional buildings and the created scenes. Additionally it is possible to change some values which influence the render-output of the scene and the generated building in the 3d-viewer. One example is the possibility to change the amount of samples and the size of the PCF[3]-filtering function, which is used to improve the shadow mapping. The intensity of the shadows can also be adjusted and is another example of the modifiable values.

4.3.2 Node Editor / Visual Rule-Editor

The visual node-editor is implemented in a way it let's the user change the procedural system in easy ways. Adding a production-rule to the system is done by right-clicking on a free space in the node-editor and selecting the desired rule, but it can also be added directly to an existing node just by dragging a new connection out of a connector of an existing rule for example. The following sections give a broader and more detailed look on how the interactions with the user-interface are implemented and realized.

Nodes

The nodes in the visual rule-editor are templated elements, which are rendered inside a WPF-canvas object. Through data-binding and the use of data-templates they are automatically created, when a new rule is added to the procedural-system. The look of the nodes is defined in the "View"-classes which are created for each rule of the procedural system. Section 4.8 provides an explanation of how the rules work and how they are implemented. Each rule is created and modified through it's corresponding view- and viewmodel-classes. This structure is used to respect the MVVM-pattern, which is described in sections 3.1.1 and 4.7.

[2]**U**ser-**I**nterface
[3]**P**ercentage-**C**loser-**F**iltering

Connections

The connections in the visual rule-editor serve an important task. They represent and define the hierarchy of the procedural system and are defined by the start- and end-connectors they connect. Those two connectors always belong to two different nodes. Each node defines one or more connectors, which can be used to connect the node to other nodes. The system is designed so that only one following-, i.e. the "child", node attached to each connector is allowed. If a new connection should be created between two nodes, some checks are performed to ensure the procedural system is valid and the derivation steps don't result in an infinite loop. The connections are represented by arrows in the visual rule-editor. The start of the arrow is visually always located at the middle of the "source"-connector and the end of the arrow is always located at the middle of the "destination"-connector. The connection-line itself is implemented using a Bezier-curve through the WPF. A double-click on a connection removes it from the procedural generator and therefore possibly changes the created procedural system. A change only occurs if the connection connects nodes, which are part of the procedural-system, i.e. nodes that are connected to the property-node.

4.3.3 Suggestions for Adding New Rules

When the user drags out a new connection from an existing connector-element in the rule-editor and later releases the mouse over an empty space in the rule-editor, a new context-window opens up. Dragging out a new connection is done by pressing the left mouse-button and holding it down while dragging the mouse. Empty space in the rule-editor means no 'destination'-connector is found near the mouse position when the button is released.

The context window is created dynamically. This dynamic creation of a user-control works in a few stages. At first the type of the source-node is used to create a list of "Tuples" containing two String objects. A short description of a Tuple is given in Appendix A.1.1. The first String of the Tuple contains the name of the button that is generated for the selection of the rule that should be added and the second String of the Tuple contains a function-name which is called when the button is clicked.

The list of Tuples is then used for the creation of the context-menu-object. By using 'reflection'-techniques for the creation of the context-menu, the second String, i.e. the function-name of the Tuple is the used to to defined the function, which is executed when the user clicks on the button. A 'click'-event-handler is also set up to actually handle the user-interaction.

When the context-window is closed, a check distinguishes between the click on a rule-creation-button and the abortion of the action, which is possible by pressing the ***Esc***-key on the keyboard or by clicking the 'Cancel'-button of the context menu. If a rule-creation-button is clicked, the attached information about the function which should be executed is restored and then used to actually create the selected rule at the current mouse position.

The last step in the creation-process is the removal of the context menu from the user interface again and actually connecting the source-node to the newly created node, which represents the new rule in the procedural-system.

4.4 The 3-Dimensional View

The result of the generated building and the loaded scene can be viewed in a 3d viewport. This viewport is implemented using the HelixToolkit described briefly in section 4.9. It allows the easy navigation in the created scene and features a fast and robust 3d-renderer. The handling of the separate render passes as well as performing all needed render-calls are handled by the renderer. Some modifications to the default shaders were made to ensure a good visual quality of the resulting buildings and scenes. The geometry drawn in the 3-dimensional view is precomputed, i.e. geometries with the same attached material are merged together to reduce the number of render-calls, to ensure a good performance.

4.5 Graphs

A procedural system can contain many production rules. Those connected rules form a graph, in this application the graph is a tree because there are no loops allowed. The nodes in the visual rule-editor represent the graph that is implicitly defined by connecting the nodes and is one of the "tree"-graphs present in the program. The second tree-graph is generated out of the defined nodes and is used for the generation process of the procedural system. The second tree is more complex than the tree of nodes mentioned first because e.g. a shape is split into more smaller shapes, which results in many more leaves in the graph. The result of the splitting-operation are many new shapes and there might be a connection from the resulting shapes to another rule in the visual rule-editor. For each generated shape the next attached rule is executed, which means that the simple representation of the depending rule in the rule-editor is applied and therefore attached to each shape. This second tree is used to perform the actual procedural generation and the generated intermediate shapes correspond to the different stages and nodes defined in the visual rule-editor. The tree is iteratively created step by step in each generation step of the procedural generation.

4.6 Step by Step Building Generation

In this section I want to discuss how the application is able to generate buildings that fit to the given floor-plan and produce a realistic distribution of rooms on each floor. At first a base-"scene" containing a property and a defined building-area is needed.

After loading a previously created scene, the render-view displays the loaded 3-dimensional data of the property as well as the area, where the building should be placed. The visual rule editor automatically adds a "property"-node and attaches and connects some default production-rules to it. An example of how the application may look like after loading a property-file is shown in figure 4.2.

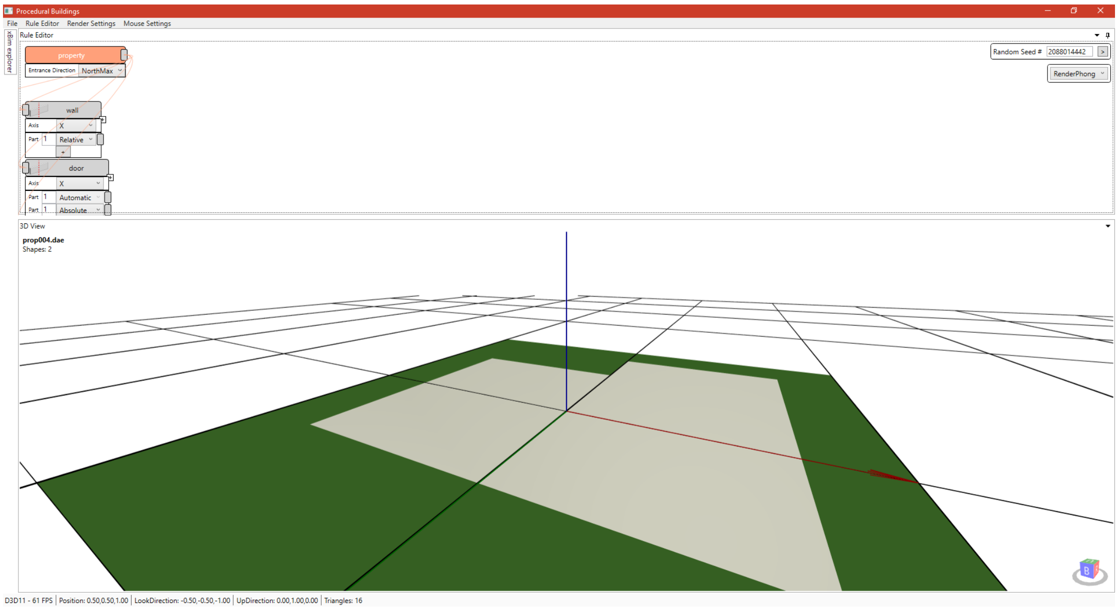

Figure 4.2: Screen-shot of the application when a previously modeled property-file is loaded.

When the loading of a property was successful, the application is ready to add more and more nodes to the procedural-system. At first a definition of the building itself is needed. This definition consists of the floors and the rooms that are placed inside the defined floors. "Vertical-connections" can be added to connect the defined floors and a roof-rule defines the top-most element of the building.

When all desired floors and rooms are placed and connected, it is possible to modify the previously mentioned default-rules. Those rules are "global" and are used for all floors and walls of the building. It is possible to define specialized rules for a floor or even for an individual room by simply attaching an equally named rule to the corresponding room or floor. This opportunity to define the rules for individual parts of the buildings makes it possible e.g. to change the width of a door connecting two rooms, or it is possible to define a completely different kind of façade as previously mentioned in section 3.1.2. An example of different façade-style-definition is shown in figure 4.3.

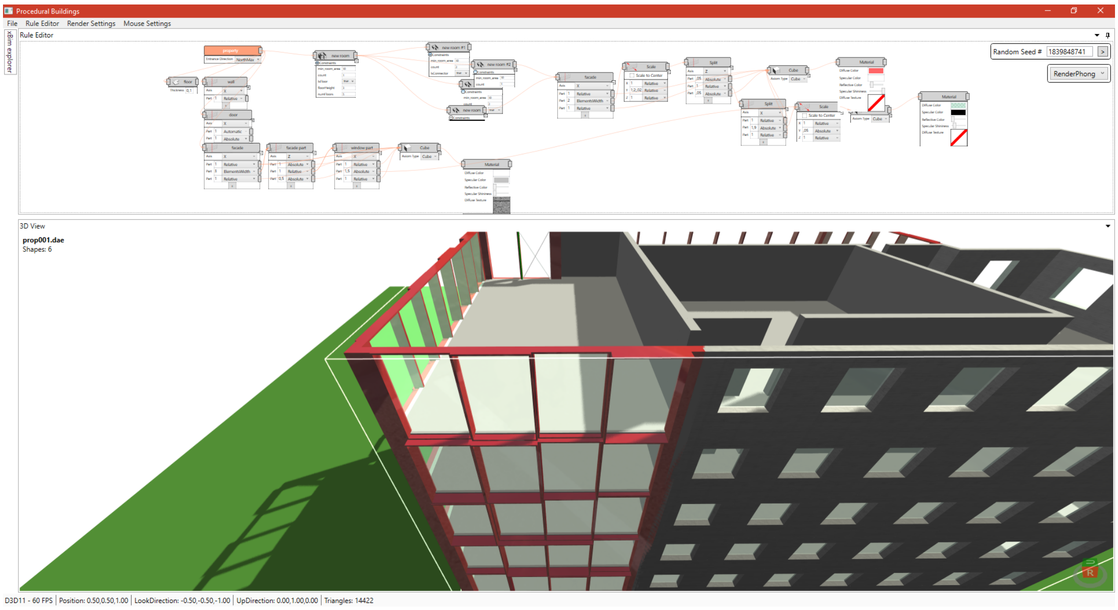

Figure 4.3: The application contains a simple scene. A building which features more than just one façade-definition is displayed. The second façade-style is emphasized by the use of a reddish material.

By adding more and more production-rules it is easily possible to enhance the level of detail of the generated building because the generation process uses all connected nodes, i.e. rules, to modify the generated geometry. All nodes in the visual rule-editor are implemented in a way, so they allow modifications through the usage of other attached rules. Two examples are the specialized rules, i.e. the roof rule and the vertical-connector-rule. Both rules create complex geometry and are easy to use. The two mentioned rules use ”elements“, which are calculated automatically. Those elements are visible in the visual rule-editor at the bottom of the rule. Each element has a name like ”Purlin“ for roofs or ”Steps“ for the vertical-connections if the vertical-connector is a stairs. The mentioned elements have a connector to make it possible to attach one of the implemented rules to them. By adding a rule, the default results of the shape-generation steps are modified. Some elements use rules which are automatically added to the element-connector at the time of creation of the rule.

4.7 The MVVM Pattern Implementation

To create a complex system like the application at hand, it is necessary to split up the project into small pieces to avoid ’couplings‘ between the individual parts of the application as much as possible. The approach used in the application is the realization of the Model-View-ViewModel design-pattern. The three parts of the pattern-name describe the three separate elements that are used. All the rules and other nodes, which can be created in the visual rule-editor are implemented using the MVVM design-pattern.

The result of using the MVVM - design-pattern is the creation of a lot of classes that work together. A single rule and the visual representation of this rule in the visual rule-editor consists of at least three classes that have to be implemented. This might seem to be unnecessarily complex, but the benefits of using this structure are bigger than the drawbacks. If only a small part of the behavior of a rule, which is implemented with the MVVM-pattern, needs to be changed, only the ViewModel-class of the rule has to be adapted. The graphical definition of the rule-node, i.e. the View, and the data of the rule, i.e. the Model, don't need to be changed.

4.7.1 The Model

The model is used to store all the data of the object and is most of the time not really an interesting class. Typically there are no functions or methods that are called in the model of the object and all access to the data stored within the class is handled through the ViewModel. The model-classes do not have any dependencies on other classes and are therefore usable, even if there exists no 'view'-class for the object. In the presented application, there mostly are no real model-classes for all the different objects because the data needed to be stored is handled directly in the 'rule'-classes, i.e. they can be seen as the model-classes.

4.7.2 The View

The view defines the visual appearance of the objects and rules in the system. In the application, there exists a '. . . View.xaml' file for all the usable nodes and rules in the visual rule-editor. A view is automatically created in the visual rule-editor whenever a viewmodel of a rule or node is added to the procedural system. This automatic creation of the views for the added elements works by the use of 'DataTemplates' that can be defined in the WPF. The ViewModel-classes do not know about the view-classes which represent the ViewModels visually in the rule-editor. The data-transfer between the views and the ViewModels is defined inside the view-classes by using the 'DataBinding' possibilities of WPF.

4.7.3 The Viewmodel

The viewmodel is the most important part of the MVVM - design-pattern. It uses the data stored in the model-class of an object and performs operations on the data when the view changes, e.g. because the user interacted with it. The ViewModel is the linking class, which uses the data from the model and the input from the view to change or update it's behavior or produce some new output. In the WPF it is easily possible to create and use the ViewModel-class because the WPF provides mechanisms like the DataBinding and the DataTemplate. Every change of a value in the user-interface, i.e. via the view class, is directly propagated to the model-class via the viewmodel. The use of DataTemplates is very comfortable because every time a new element is added to the procedural-system, a corresponding view is created for the viewmodel. Through DataBinding the view always displays the values of the object and the object holds the values defined or changed in the view.

4.8 Organization of the Rule-Classes

The following figure 4.4 shows how the different classes of the rules work together. There exists an abstract base-class 'Rule‘ that derives from the abstract base-class 'BaseView-Model‘. It defines the most important method "AppyRule“, which is only a simple basic implementation that just packs the argument Shape into a list of shapes, which it then returns. An always succeeding condition-check is used to show the extensibility of the application. It would be possible to actually use this condition-test in the future, but it is not used at the moment of writing this thesis. The hierarchy of rules makes it easy to create a list/set/tree of 'Rule'-objects and always just call the 'ApplyRule‘ method without the need to cast the Rule to the actual instance-class. Each deriving class overrides the basic method and implements the details of how the rule should work.

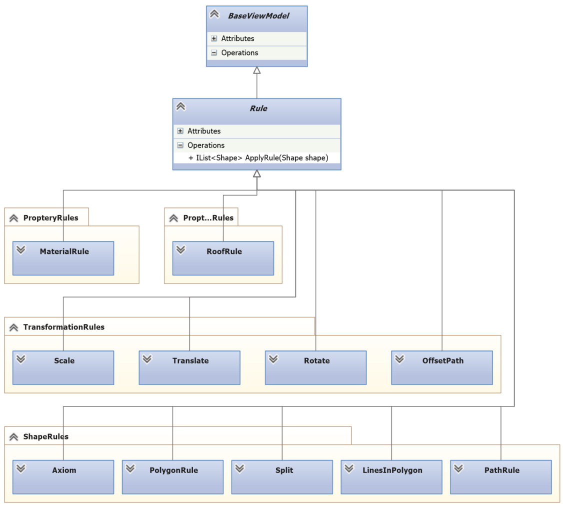

Figure 4.4: Visualization of the rules-classes. The abstract Rule class derives from the BaseViewModel class and defines a virtual method 'ApplyRule‘, which is overridden by every deriving rule-class. The derived classes are clustered to visualize the implementation-file they are implemented in.

4.8.1 How the Rules Work Together with the Other Parts of the Application

After showing how the rules are structured hierarchically, another important aspect is to understand how the rule-classes are actually used in the application. To give an overview, please see the schematic visualization in figure 4.5. When the generation process for the building is started, at first all basic building-elements are generated. These generated elements are represented by simple Shapes (see 4.2.4 and 4.10.1), PolygonShapes (4.2.4 and 4.10.2) or PathShapes (see 4.2.4 and 4.10.3), which can be modified with the procedural rules.

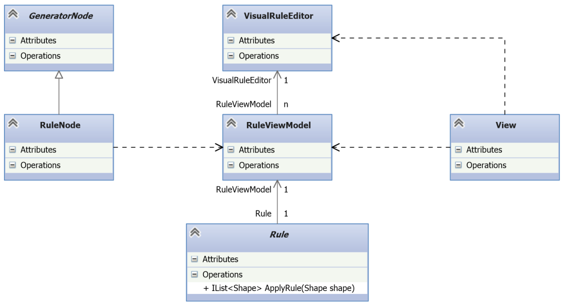

Figure 4.5: Simplified visualization of the interactions between the rule-related classes. The Rule (bottom) defines the 'ApplyRule' method, which performs changes on the Shape and returns one or more new Shapes. The RuleViewModel is used to interact with the rules through the visual rule-editor and is assigned to the RuleNodes in the generation process.

In figure 4.5 the main actors of the generation process are shown. The Rule object is created by and assigned to a RuleViewModel - object. The RuleViewModel uses the Rule as it's model. They are created in the VisualRuleEditor, where the View is automatically generated through a DataTemplate. The DataBinding between the View and the RuleViewModel ensures synchronized values for the individual settings of the Rule displayed in the View.

When the procedural generation is started for a Rule, i.e. if it is attached to e.g. a RoomNode in the VisualRuleEditor, the RuleNode is attached to it's parent node at first. Both, the RoomNode- and the RuleNode-classes are derived from the abstract GeneratorNode base-class. The class implements a method called ApplyRule using an input-shape. The method then calls the Rule's 'ApplyRule' method and stores the returned list of shapes. In the next step it is determined if there is a RuleViewModel attached to the current RuleViewModel and a new RuleNode for the attached RuleViewModel is generated if there is one. For each generated shape the AppyRule - method is called from the new RuleNode - child. This process is recursively performed until no RuleViewModel - child is attached anymore.

4.9 Helix Toolkit

The 3-dimensional viewport in the application uses the Helix Toolkit to display the generated content. The HelixToolkit [Oys12] is based on SharpDX [Mut10], which is an open-source managed wrapper for the DirectX API. The HelixToolkit implements a lot of different 3-dimensional objects and their models.

The HelixToolkit implements an easy to use WPF control which is usable by only writing a few lines of code. Some predefined objects like the grid, which is used to visualize the base-plane in the application exist. Adding and removing 3-dimensional objects is easy and interactivity is provided as well because keyboard and mouse-input is handled automatically. Setting up and updating the camera-projection matrix according to the user-input is also automatically handled by the toolkit.

A few details of the toolkit were changed because they didn't work as expected. For example, the code for saving a 3d - scene had to be updated to work properly. Some changes in the Vertex-Shader were made to improve the shadow-mapping, which is also included in the toolkit, and the method for merging more objects to one object was also updated. This feature is used to merge shape's geometries for all shapes with the same defined material.

The toolkit handles the rendering of the scene and the camera-updates, when the view changes. It is also used to load property-scenes into the application and for saving a result of the generation process, so the building can then be loaded into another application, e.g. for visualization purposes or for rendering the building.

4.10 Shape Implementation

The generated elements in the application are the shapes. The shapes are the results from the different steps in the building-generation processes as well as from the application of a rule to a previously generated shape. Shapes are both, result of a generation step, as well as the input for the following generation step. There are three different kinds of shapes implemented and used in the application.

4.10.1 Basic Shapes

The simplest type of shape is just called 'Shape' and is just a wrapper for the 'Geometry-Model3D' object implemented in the HelixToolkit. The GeometryModel3D contains a lot of information of the 3-dimensional object and is always used for polygonal objects, i.e. it has a Geometry. Besides the Geometry it contains a 'ModelMatrix', which defines the position, the orientation and the size in the scene, and methods to pop and push new transformations to and from the object. A hit-test, which is used for the picking of objects through the mouse is also implemented in the GeometryModel3D.

4.10.2 PolygonShapes

The PolygonShape directly derives from the shape-class and adds two properties to the object. It is used whenever a 3-dimensional object has a 'base-shape' and a thickness like roof-parts or the floor geometries. The first additional information is the 'PolygonOutline' which is used to define the shape of the 3d object. The second property is the 'Thickness' value. It defines the height of the object measured perpendicular to the base of the object. With the polygonal shapes it is possible to create a lot more complex shapes, which would be hard to model with the basic shapes mentioned before. Examples for PolygonShapes are the individual roof-parts, the floor-elements and the steps for the spiral stairs.

4.10.3 PathShapes

The most complex shapes in the application are the PathShapes. They derive from the polygonal shapes and use an additional property for the definition of the shape. The polygon shapes have a base-shape, which is extruded along a line to create the geometry, i.e. the extrusion line it it's path. The difference to the path-shapes is that the path-shapes define not only a thickness-value for the extrusion, but a complex path. This path can be very complex and the geometry-generation results in objects that are impossible to model with the previously mentioned types of shapes. The path-shape was mainly introduced for the use with spiral stairs, where round objects are needed. The path-shapes are currently only used in combination with stairs, but with new rules added to the system, more complex geometries could be generated, e.g. it would be possible to create a 'tree'-rule and use the path-shapes for the individual branches of the trees.

4.11 The Implemented Rules

A lot of information about the rules was already given. In figure 4.4 a small overview of how the rules are related to each other and in figure 4.5 the role of the rules in the procedural-generator is shown. It follows a short explanation of how the individual rules are implemented.

4.11.1 The Axiom Rule

The axiom rule is used to actually add a geometry to a shape. A shape has no attached geometry without the use of this rule by default. Exceptions exist for the more complex shape-types and for the special rules implemented. The rule checks the defined type of geometry that should be added to the input-shape. If the type is 'Cube', a cubic geometry-definition is generated using the HelixToolkit. The geometry is added to the shape and the result is returned. It is also possible to remove any attached geometry by selecting the 'Empty' - type in the visual rule-editor. An extension to cylindrical geometries would be possible, changes in some existing rules would have to be made, though. For example it has to be defined how the ’Split‘rule works in combination with a cylinder-object.

4.11.2 The Polygon Rule

The ’PolygonRule‘ only works with PolygonShapes. If any other shape is used for the rule-application, an empty shape is returned. The PolygonRule equals the Axiom rule for PolygonShapes. It just adds the defined geometry to the shape by using the defined polygon of the shape as well as the thickness.

4.11.3 The Path Rule

The ’PathRule‘ is equivalent to the both before-mentioned rules, it just works with the PathShapes. The rule uses the defined polygon as cross-section - definition and the path for the extrusion of the cross-section to generate the geometry. The actual generation is performed again by the HelixToolkit.

4.11.4 The Material Rule

The ’MaterialRule‘ is the simplest of the rules and just adds the material to the shape independent of it’s type.

4.11.5 The Translate Rule

The ’Translate‘ rule is a transformation rule, which only affects the position of the object in the scene. A translation-matrix is calculated, depending if the translation should be performed with respect to the object’s local coordinate-system, i.e. local, or if it should be relative to the scene, i.e. global. The calculated translation-matrix is then pushed onto the shape’s model-stack to update the model-matrix of the shape.

4.11.6 The Scale Rule

The ’Scale‘ rule is another transformation rule. It is used to create a different scaling of the shape. If the input-shape is a path-shape the scaling only affects the path-shape’s cross-section, i.e. the polygon definition, by assumption, but not the path of the shape. If another shape should be affected by the Scale-rule, a scaling matrix is calculated with the HelixToolkit and is then applied to the Shape. If the Scaling should be performed with respect to the center of the Shape, additional calculations for the translation-matrices are used to achieve the desired result.

4.11.7 The Rotate Rule

The ’Rotate‘ rule is the last rule that only affects the model-matrix of the shape. A rotation-matrix is calculated and applied to the shape to create a global or a local transformation. The rotation is always performed relative to a defined rotation-center.

By respecting a rotation-center it is possible to rotate an object around it's origin, or around a special point, e.g. the center of the shape. This possibility to define the rotation point is very useful e.g. for the creation of doors or windows.

4.11.8 The Split Rule

The 'Split' rule is one of the most complex rules implemented in the application. It distinguishes between the three different types of shapes to create a meaningful result. If the input shape is a basic shape, only a few calculations are performed. Since it is possible to define absolute or relative values for the individual parts that should be generated from the input shape, the resulting sizes are determined at first. For each definition of a split-part it's final size is calculated and the new shapes are positioned next to the other split-parts. If one split definition defines more than one resulting shape, e.g. by selecting to create five equally-sized resulting shapes, the calculations are automatically adapted.

If a PolygonShape is split, a distinction between 'split the ground-shape' and 'split along the extrusion' is done to create the desired results. If the shape is split in the direction of the the extrusion, i.e. split along the axis 'Z', some new polygon shapes are created by using the same polygon-outline as the input shape and changing only the thickness-value and the model-matrices of the resulting shapes. If the polygon-outline should be split on the other hand, a very useful clipping-library [Joh10] is used. It performs the polygon-clipping operations and is used to split up the polygon-outline into the defined sub-polygons. The calculated sub-polygons and the existing thickness-value from the input shape are then used to create the new split-part objects that are then returned.

Another distinction is made if a PathShape is split. If the cross-section of the path-shape should be split, calculations similar to the polygon-shape - split are performed to create the new split parts. The path of the path-shape remains the same for all generated split-parts, but the PolygonOutline, which defines the cross-section, is split up according to the split-definitions. If the path should be split, a few other steps have to be performed to create the wanted results. All the newly created split-parts are in turn again path-shapes and share the same cross-section as the input shape. In this case, the path of the path-shape is split into more parts. The new path-shapes are generated by using the definitions of the split-parts to split the path into several parts. To be able to split up a 3-dimensional open polygon, i.e. the path of the PathShape it is necessary to calculate the length of the path first. Additional points at the defined lengths are calculated on the existing path to create the paths for the split-parts with the exact defined length.

4.11.9 The LinesInPolygon Rule

The 'LinesInPolygon' rule is used to create new basic shapes that are positioned along imaginary lines splitting a polygon. This rule is best used with PolygonShapes because with this combination it is possible to create a lot of detail for complex elements like the rafter- and the purlin-elements of a roof. It works similar to the above Split rule, but this rule does not create 'whole' splits of the polygon but only the split-lines, hence the name. The clipper-library [Joh10] is used in combination with an extension method to perform the split-operations and calculate the lines inside the polygon. In fact the lines are pre-calculated at their final positions, but they are too long and may be longer thnt the desired result, and are then clipped against the polygon to shorten them. At last the clipped lines which are positioned inside the polygon are then used to create new shapes that range from the start-point to the end-point of each line.

4.11.10 The OffsetPath Rule

The 'OffsetPath‘ rule is only used to offset the path of a path-shape. This rule behaves like a 'geometry-aware' translation of the cross-section of a path-shape. In fact the rule does not move the polygon-outline of the path-shape, but recalculates the path with translated positions of the path-defining points. This operation is only defined for path-shapes, so for all other types of input shapes an empty result is returned. The rule is relatively specialized because in the current state of the application, it is only useful in combination with the pre-calculated elements of the stairs. The only existing path-shapes in the building. One important aspect of this rule is that using the OffsetPath rule always keeps the original shape and additionally creates a second shape.

As mentioned before, this rule only affects the path of a path-shape. To be able to transform the path, the normals, tangents and the binormals are calculated before the changes are made. The offset of the path is defined always in the direction of the normal of the path. The clipper-library mentioned in the previous rules is able to perform exactly this operation, but not in the 3-dimensional room. Before actually translating the points of the path, a check is performed to avoid 'overlapping paths‘ at reflex-vertices when the offset is too big.

The angles between the previous and the next line-segments are calculated for each path point. This angle-value is used for the generation of the new path-shapes with the 'Cut‘ or 'Round‘ corner types defined. An additional use for the angle-values is the stretching of the cross-sections at the current point of the path. In figure 4.6 a visualization of the stretching of the cross-sections is shown and why this stretching of the cross-sections is needed to ensure a constant cross-section - size along the path of the path-shape.

If the offset should be performed with an 'OffsetCornerType' of type 'Simple‘, the normals and the remaining points of the path are used to create the offset path and the new resulting shape. A last step in the generation process is the removal of path points which are positioned near to each other.

If the OffsetCornerType is set to 'Cut' or 'Round' some additional path points are inserted for sharp corners. In the case of the Cut-type, only one additional point is inserted, but when the Round-type is defined in the rule, more points are added to the path, depending on the angle at the corner.

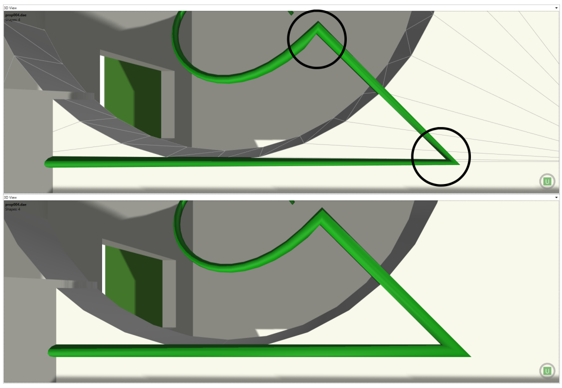

Figure 4.6: Top: the two circles mark the positions with big changes in the direction of the path. The cross-sections, which are positioned in the plane of the angle bisectors, are all exactly the same size, resulting in weird looking and distorted objects. Bottom: the cross-sections are stretched, so that the resulting object has a constant cross-section everywhere if measured perpendicular to the path.

4.11.11 The Roof Rule

The 'Roof' rule is not a classic CGA-rule because it cannot be attached to any other rule, i.e. it does not have an effect everywhere. It is only possible to attach a roof rule to the building node. This restriction is introduced because not more that one roof-rule makes sense at the current state of development. The roof generation is actually a separate generation step in the building - generation-process and the ApplyRule method does nothing, so it is safe to use it everywhere, it just won't change any shape if not attached to the building node. The roof rule stores all defined information about the roof. The default slope of the roof is stored as well as the roof angle and the default extent. Not all values are needed for all the different types of rules, only the needed one's are used in the generation process. If different values for slope and extent are defined for the individual roof-parts of a hipped roof, those additional values are also stored in the roof rule.

4.11.12 The Vertical Connection Rule

The rule for creating vertical connections is also not a rule in the classical sense. Attaching this rule to anything other than a room node in the visual rule-editor will have no effect. Only if the rule is attached to a room-node, a vertical connection will be positioned in the floor and the individual parts will be generated. This results in the fact that the rule is not deriving from the base 'Rule' class but the generation is performed by using a separate generation step in the procedural system.

4.12 How to Add a Rule

The system of the implementation of the rules might seem very complicated at first, but in a few steps it is possible to add a new rule to the system. This way is is easily possible to extend the developed system with new useful rules.

The first step of adding a rule to the system is to define it's behavior and decide a name which is meaningful and describes what the rule does in the procedural system. After the name and the behavior is defined, the possible user-interactions with the rule have to be defined as well, i.e. which values should be changeable in the view, which one's should be fixed.

In the next phase the rule-implementation starts. This implementation consists of implementing at least three classes. The most important class to implement is creating a new sub-class of the rule - base-class. This new sub-class needs to implement the 'ApplyRule' function to be actually useful and it defines what the rule does in the procedural system. The second class that needs to be implemented is a sub-class of the 'RuleViewModel' base-class. The base-class implements the needed functionality and properties. It is used to act as the view-model in the MVVM design pattern and uses input from the view class to pass to the rule class, which performs the procedural calculations. The third class is the view of the new rule. It is defined in WPF and uses the properties of the view-model class to display values and other controls. The data-binding is defined in the XAML file of the view class.

To make use of the new rule, two more steps need to be done. Since the visual rule-editor uses a collection of view-model objects to display the nodes, not the views themselves, it is necessary to define the data-template for the new view-model of the procedural system. This can be done by adding another data-template in the 'NetworkControl.xaml' - file. After the steps are finished, the rule can actually be used. The rule needs to be added to the system somehow, so the last step for adding a new rule to the procedural system is to add a new context-menu entry and a click event-handler for the context menu. This event-handler is used when the new rule is added to the procedural system by right clicking on an empty space in the visual-rule-editor and then clicking at the newly added context-menu entry. The event-handler actually creates the wanted view-model and sets some values if needed, e.g. setting up additional event-handlers.

If additional event handlers are needed for the new rule, it is also necessary to add the special event-handlers, when a scene is loaded, which contains this new rule. This is easily achieved by adding a new condition for the rule in the 'IOCore' - file's 'LoadProject' function.

4.13 Important Variables and Fixed Values

The creation of the buildings is implemented as flexible as possible, but some important values are fixed in the current stage of development. The layouter-grid class, which contains the most fixed values, contains some examples. The first fixed value present in the layouter class is the size of the grid-cell of the layouter-grid. If the value gets smaller, more grid-cells are used to calculate the floor layout. Another fixed value in the grid are the widths of the start areas for the rooms that are distributed and planned in the floor. The width of the corridor is also fixed to two meters to ensure a realistic result in the created floor layouts.

Other fixed values are found for the positioning of vertical-connectors in a room. The fixed value controls the maximum amount of tries to position the vertical connector in the room because it is possible that no valid position can be found.

4.14 Limitations of the Created Procedural System

The application is designed to be able to handle a wide variety of different possible settings like different shapes of the buildings, different floor- and room-configurations and many different roof- and vertical-connector - options. The following sections describe the most important aspects which cannot be realized with the application in it's current state of development.

4.14.1 Building Limitations

Currently there exists no possibility to create properties for the buildings inside the application. The properties have to be modeled in an external application, saved in the defined format and then loaded into the implemented application. An editor can be implemented into the application to automate not only the generation of the buildings, but also the creation of the properties that define the size and shape of the buildings.

It is possible to only generate one building at a time in the current implementation of the procedural generator. If a bigger scenery with more buildings is needed, this can only be realized in an external application, which uses the created buildings from the presented application. It is possible to overcome this limitation by modifying some parts of the program like the scene-loading and the generation-process.

The generated rooms are limited to only contain axis-aligned inner walls. This limitation is a result of the use of a grid in the room-layout process. Most real buildings only contain such axis-aligned walls, so this is not an immediately noticeable limitation, but contemporary architecture often results in the creation of buildings which include other wall-directions as well. It is possible e.g. to allow 45°- and 135°- oriented walls as well, but the room-layout algorithm has to be modified for this change. Using another room-planning algorithm is also possible if more complex wall-geometries are wanted.

Currently there is also no way to create rooms which are taller than one floor in height. Taller rooms are needed, e.g. if a factory or a mall should be modeled with the application. By modifying the existing rooms, i.e. adding a floor-height value, or adding a new rule for those special rooms, it would be possible to add the generation of higher rooms to the procedural system.

A floor is always located at one height level. This means it is not easily possible to create rooms of a floor, with a different height level, which is sometimes wanted for buildings e.g. located at a hillside.

There exist no rules for modifying the layout of a floor at the moment. Especially residential buildings and office buildings use different shapes and sizes of floors in the upper levels compared to the lower levels. Modeling a multilevel office building which has smaller and smaller floor-plans from bottom to the top is currently not possible. It is possible to implement a whole 'class‘ of new rules that affect the shape and size of the floor for example.

In relation to the last mentioned limitation, there exists another limitation regarding the roofs. If a floor would e.g. be reduced in size, some parts of the size-reduced floor could change to roof-areas. The handling of this situation is not needed at the moment and therefore it is not implemented in the system. Currently always the whole size of the floor is used to generate the roof.

The property areas are currently always empty. It is possible to modify the 'PropertyNode' to allow attaching rules to the 'yard-part‘ of the property, but additional rules need to implemented in that case.

4.14.2 Other Limitations

If the procedural system gets very complex, i.e. there exist many rule-nodes in it, the visual rule-editor becomes slower a bit when the view is dragged and panned because in fact all the nodes are inversely moved to fake this panning.

The visual output of the generated buildings has limited quality because only the direct lighting is taken into account for the rendering. More beautiful and realistic results can be achieved by exporting the generated building into a professional rendering application.

Currently there is only one export-format available, but most of the available 3D modelers and rendering applications are able to load this simple file-format.

CHAPTER 5

Results

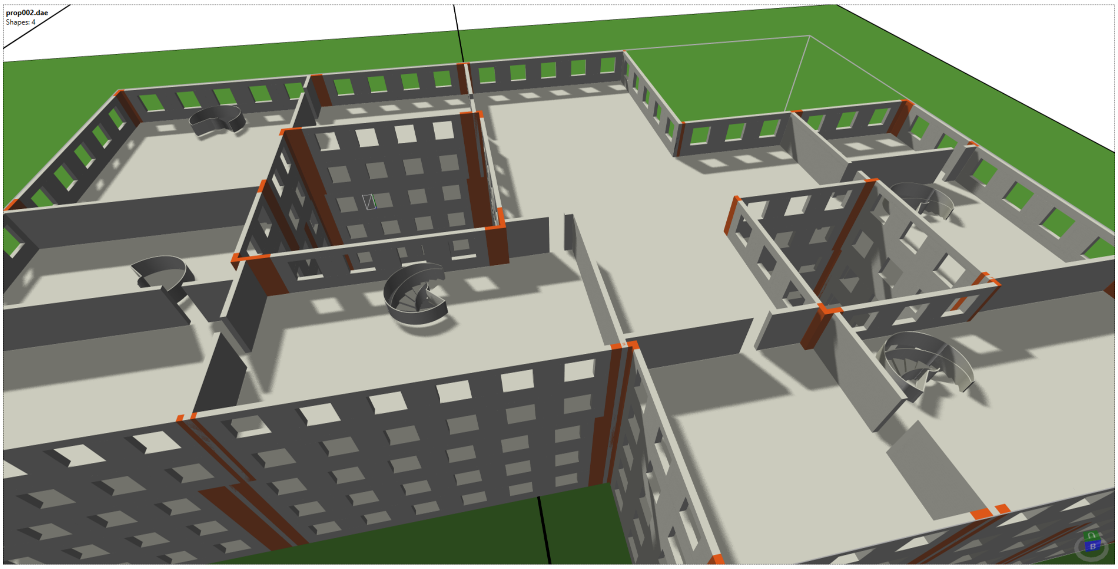

Figure 5.1: A big floor plan is subdivided into five regions. The floor plan contains two courtyards, which are taken into account in the floor planning steps. One vertical connector is defined for each of the five regions. Each region consists of two rooms. In the lower left region the differences of the two different floor plans of the building can be observed. The different orange regions indicate a different room layout in the floors below.

Figure 5.2: A big floor plan is used to create this result. The façade-definition is used to create balconies with a random size. There is no special handling of balconies, the creation is achieved by using the default production-rules. This is an example for a big apartment building.

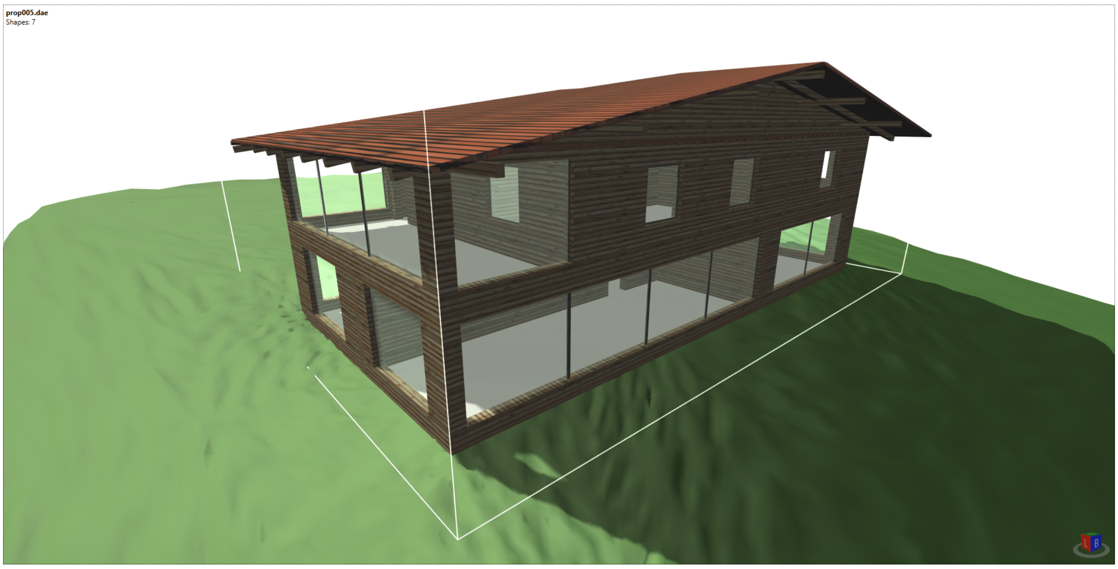

Figure 5.3: A wooden building with a complex roof consisting of many individual roof parts, some of them with different slopes.The building consists of two differently defined floors and two different façade-definitions for some of the rooms.

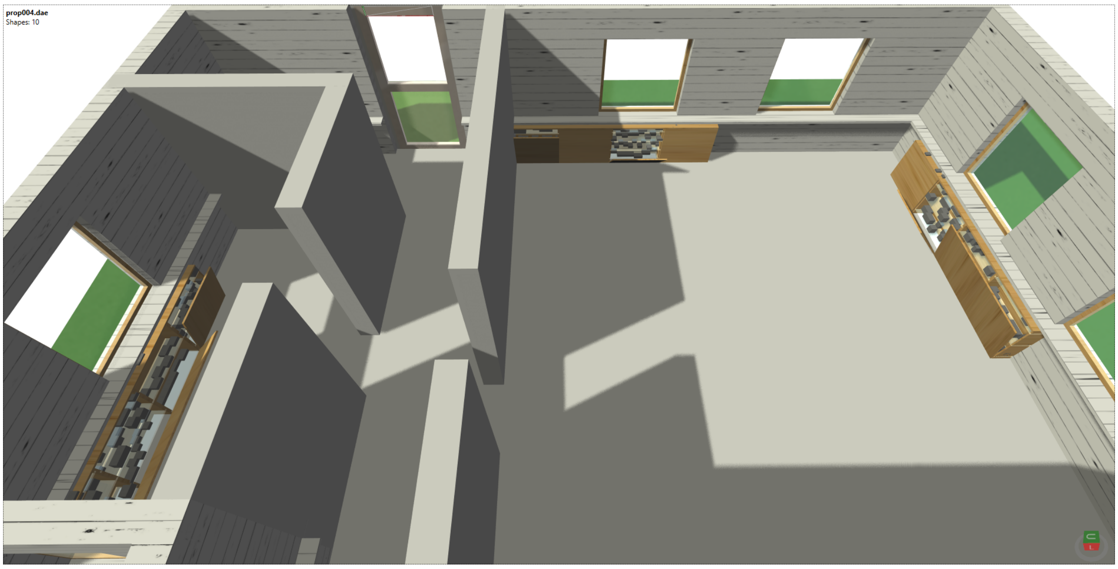

Figure 5.4: A simple bungalow-building with a corridor and two rooms visible. The CGA-rules that control the façade-generation are used to fake some furniture elements at the wall inside the rooms of the building.

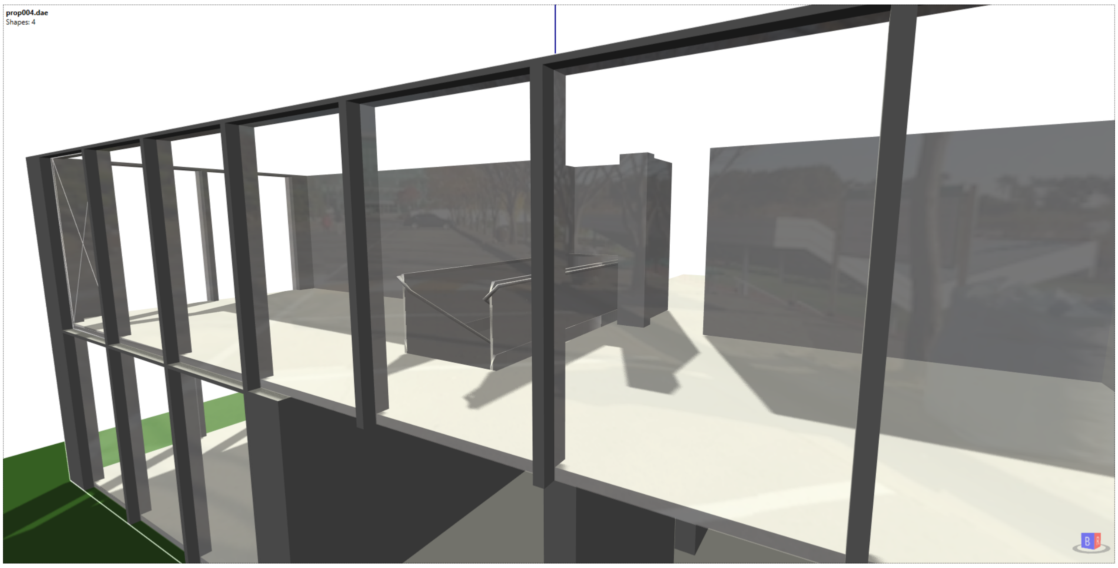

Figure 5.5: More complex building with two floors and two façade definitions. One of the two façade -types is not rendered in the screen-shot. The remaining façade has reflective glass-elements. A straight stairs connects the two floors. In this case the previously created rooms adapt to fit the positioned stairs. This fitting-of the stairs in the room can be observed in the upper floor, where the back side of the stairs ’extends‘ the room to completely fit inside the room.

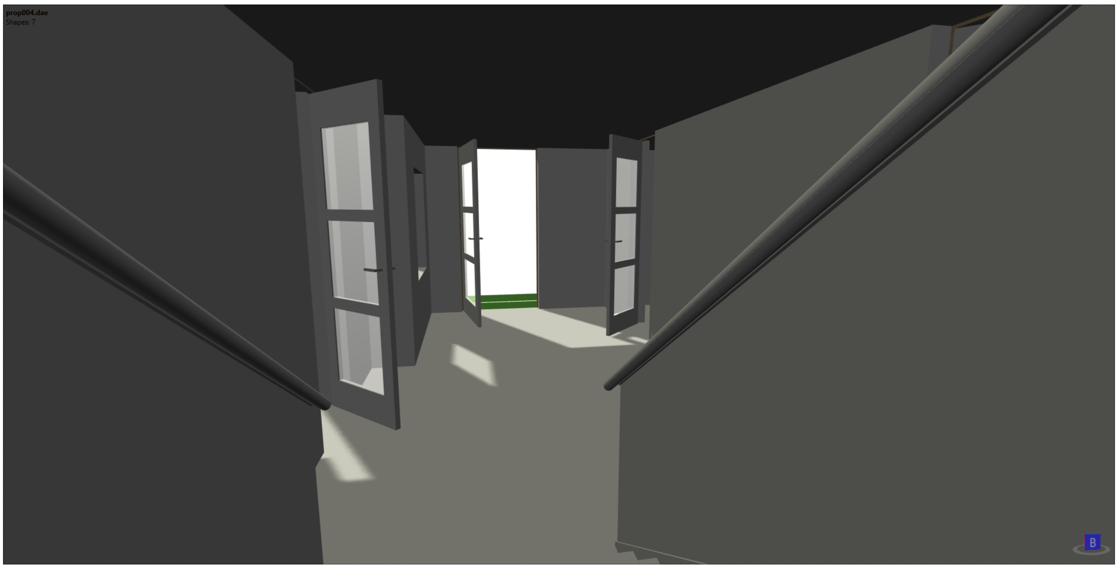

Figure 5.6: This is the same example as seen in the figure before, but with a more complex definition of the doors. The door-handles as well as the doors themselves could be defined in a more detailed way to further increase the level of detail.

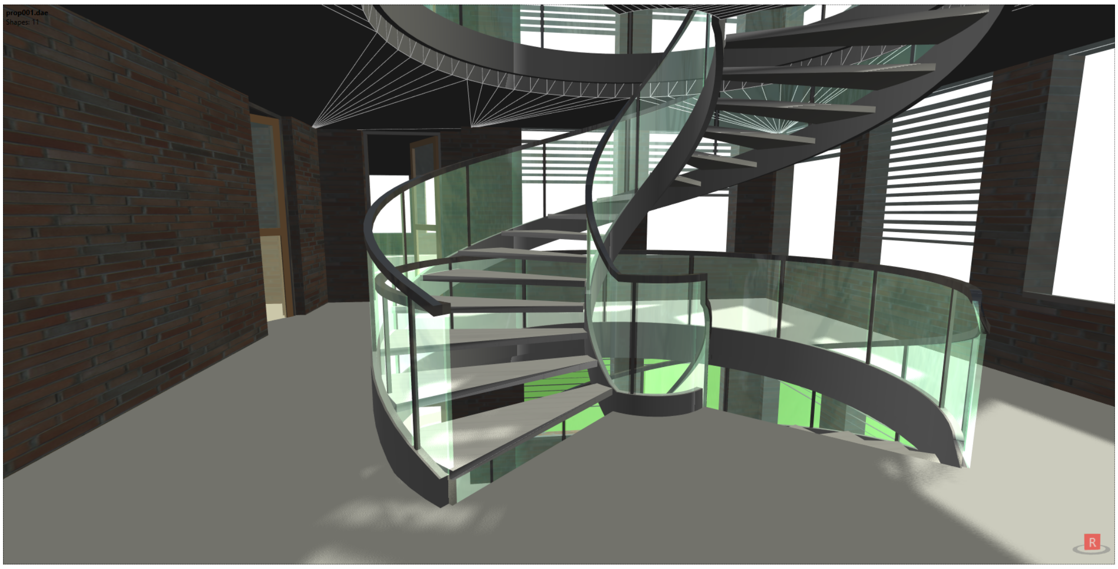

Figure 5.7: Partly textured example of the interior of a building. The geometry of the spiral stairs is generated by one of the two specialized rules introduced in this work. Some elements of the stairs use a glassy material to improve the visual quality of the generated result.

CHAPTER 6

Conclusion and Future Work

6.1 Synopsis

With the implementation of the procedural generator at hand, a small step towards the simplification of the usage of procedural-generators is made. It is now possible to create even complex procedural rule-systems for the generation of buildings without the need to be able to program a single line of code. While the procedural systems used in the CityEngine are quite complex and contain a lot of code, the process of creating the procedural system in the implemented application is almost self-explanatory Examples for the definitions of the procedural rules used in the CityEngine can be downloaded here [Cit13]. The variety of buildings which can be generated with the application is quite big, considering the fact that only the most basic CGA-rules were implemented in the system, with the exception of the vertical-connectors and the roof.

Creating a system that is easy to use and that generates believable buildings is not an easy task, but some of the used techniques, like the visual rule-editor may impact future developments of other applications. The implemented solution is far from being perfect, but it is shown that it is possible to create complex definitions of a procedural system without the need to write a single line of code. This fact is an important contribution of this work because people working in creative industries, e.g. people working in the game industry or movie industry, architects, designers, etc. , who might not be able to create such a procedural-system by writing the code for it, may be able to use applications featuring visual rule-editors in the future, which makes it possible for many more people to actually make use of these techniques.

When developing the application, some issues had to be dealt with. For example it was not easy to use the visual-rule editor. It consists of a lot of classes and getting an overview over the implementation took quite some time. I managed to implement all the functionality into the editor, but a working solution could have been accomplished with a simpler approach as well. Taking into account the time needed to understand the underlying network-view classes and the achieved overall-performance of the editor, it would have probably been a better decision to implement a visual rule-editor using the Direct2D or OpenGL APIs[1].

While the solution works quite well in most cases, there exist some potential restrictions because not all parts of the application are optimized very well. One example is the fact that the panning of the rules inside the visible area of the visual rule-editor is kind of faked by inverting the mouse-dragging and applying the delta-values to each of the node's coordinates, i.e. the X- and Y- properties, which results in many recalculations for the nodes and connections, thus slowing down the speed of the editor.

Another problematic case is the visual "highlighting" of hovered elements of the building in the 3-dimensional view, i.e. the additional wireframe visualizations of the elements. When working with more complex buildings, which consist of e.g. more than 20.000 individual geometry elements, the ray-casting algorithm, which performs the checks to identify the hit element, takes quite some time to finish it's calculations. Using e.g. an Octree would enhance the ray-casting algorithm a lot.

The application uses the HelixToolkit, which was really helpful at the beginning because the library implements a lot of functionality, which is used in the application. Some of the implemented helping functions were not finished or broken at the time of the development of the application, so a change to a few methods and functions in the HelixToolkit had to made, like the saving functionality for the building geometry and their materials. Some changes to the shaders and to the blend-functions were made as well as other small changes. Since the buildings can easily consist of thousands of simple objects, it became clear very early in the development process that the amount of render-calls had to be reduced. Instead of executing thousands of render-calls for each frame of the 3d-view of the building, all elements with the same material-settings are merged together, which immediately led to a huge speedup of the render-process.

6.2 Conclusion

The result of the master thesis is a working application, which is able to create nice-looking and realistic buildings with a procedural generator. The visual rule-editor and the 3d-renderer was not implemented, but partly changed, as part of this work, which on the one hand, saved some time for the implementation at the beginning, but created some problems on the other hand. The application is designed using the MVVM design-pattern where it made sense to use it. It is extendable, so new rules or axioms can be added quite easily.

[1] **A**pplication **P**rogramming **I**nterface

Not all useful and desired CGA-rules and optimizations were implemented in the application and the procedural-system, but a good working framework for the generation of buildings was developed and can be enhanced in the future.

6.3 Future Work

Many possibilities to improve this application exist, most of the limitations from section 4.14 can be solved by just adding some more rules to the procedural system.

6.3.1 Editor Improvements

The visual rule-editor already features some useful functionality to navigate the procedural system, add and remove rules and more. After using the application for hours it became clear that a feature to automate the distribution of the created rules would be very useful, i.e. a 'grouping'-functionality. Placing connected rules next to each other would simplify the creation of the procedural-system. An implementation may be non-trivial because the connections should minimize the number of intersections between each other to enhance the overview of the procedural-system.

A second improvement to for the visual rule-editor would be to move from the current implementation using default WPF components to e.g. an OpenGL or DirectX based system, which would make it easy to work with hundreds of rules present in the visual system without any slowdown.

Other possible extensions to the application and the editor could be the possibility to create "new" rules from a group of existing ones. A second extension for the future could be a gallery of saved rule-sets like a 2d-split rule consisting of two default split rules that can be added to the procedural system to further speed up the design process of the procedural buildings.

6.3.2 Room Layout Improvements

The implemented room-planning algorithm is limited to axis aligned inner walls. Another algorithm implementation which allows more general directions of walls would improve the quality of the resulting buildings a lot. Another important aspect to mention is the positioning of the vertical connections between the floors. In the current implementation it is not guaranteed that a vertical connection is generated. This can happen, when there is too little space to position it properly because the rooms are positioned first and they need to be accessible from the other rooms. A change in the order of the creation of the rooms and the positioning of the vertical-connectors might improve this behavior.

6.3.3 Generator Improvements

It is also possible the improve the generator itself by avoiding not necessary intermediate geometry generation steps for example. By determining the complete generation-tree before the actual geometry generation it would be possible to actually only generate geometry, which is present and rendered in the result.

Updates of the procedural system currently often result in the recalculation of the whole procedural system. By tweaking this update behavior, it would be possible to create those changes a lot faster and therefore result in an application that reacts better to the user-input.

Adding whole new classes of rules to the generator would be possible as well. There could be mesh modifying rules like 'subdivide', 'merge', a 'random point-wise transformations' of the mesh and so on.

APPENDIX A

Definitions

Some definitions which are useful for the reading-process of this master thesis are shown below. The definitions will cover some computer-graphics - related terms as well as some architectural ones.

A.1 List of Terms

For the better understanding of the master-thesis, some definitions and descriptions of terms, which are important and often-occurring in the text follows.

A.1.1 List of Used Terms

- **Building** - The main part of the master thesis. Buildings consist of one or more floors.
- **Child-Node** - The procedural system forms a hierarchy of rules. A **child-node** is therefore a 'child' of another node in this hierarchy.
- **Connection** - The **connection** between the nodes in the visual-rule-editor.
- **Connector** - The **connector** is a special room inside a room-collection or a floor. A connector always connects all the rooms in the same hierarchy-level inside a room-collection, i.e. it's siblings.
- **Connectivity** - All rooms of a building have to be connected somehow and the floor-planning algorithm ensures the **connectivity** of all rooms.
- **Design-pattern** - A **design-pattern** is a pattern that describes how to structure classes in a system. There exist many different **design-patterns** that are useful for different purposes like minimizing the dependency between classes.

- **Floor** - A **floor** is part of a building and consist of one or more rooms. A **floor** in the application is restricted to one height throughout the whole **floor**. This means all rooms in the floor start from the same height and end at the same height.
- **L-system** - **L**indenmayer-**system**, introduced in 1990 by Aristid Lindenmayer [PL90] to simulate the growth of plants.
- **Node** - A **node** is a visual element that is displayed in the visual rule-editor and represents some sort of information. This can be a property, a room-definition or a rule.
- **Procedural content generation** - The process of content generation, using procedural techniques, i.e. create the content not manually, but by defining some procedures.
- **Procedural system** - The **procedural system** consists of the ruleset and is used to generate some content procedurally.
- **Room** - A **room** is part of a floor. Many rooms can be positioned in a floor while the connectivity is always guaranteed.
- **Room connection** - This mainly refers to doors, but by changing the rules, a **room connection** could also be an completely open part of a wall as well .
- **Room-collection** - Basically it is the same as a room, but has 'sub-rooms' defined, i.e. attached to it in the visual rule-editor. A **room-collection** always consists of at least two rooms. All rooms of the room-collection are connected to each other locally. An example for a room-collection is an apartment definition inside an apartment-building.
- **Rule/Procedure** - A **rule/procedure** is part of the procedural system. It is used in the generation process and is applied to the shapes and therefore manipulating the shape. In the application the rules are visually represented by nodes in the visual rule-editor. It consists of a predecessor and a successor. The predecessor is matched in the generation process and the successor is it's replacement in the next iteration-step.
- **Rule-set** - The set of all rules in the procedural system.
- **Shape** - A **shape** is a basic element in the generation process. Rules are applied to **shapes** and modify them in a defined manner. There are three types of **shapes** implemented in the application, see section 4.2.4.
- **Tuple** - A **Tuple** is a C# object that can store more than one value. These Tuples are useful, when no special functions and methods on the stored data should be performed.

- **Vertical connection** - A **vertical connection** can be an elevator or some types of stairs. It refers to something connecting two or more floors of the building.
- **Visual rule-editor** - The implementation of the **visual rule-editor** is a contribution of the master thesis at hand. It allows the creation and modification of the available rules in a visual manner. No code has to be written and the nodes can be connected easily in the **visual rule-editor**.

APPENDIX B

Pseudo Codes

B.1 Example Code for the ”CityEngine“

The following code is taken from a scene created in the CityEngine 2015. The scene ”Philadelphia example“, downloaded here [Cit13], uses a lot of different code-files to put the whole scene together. The following code is taken from the file ”Generic Modern Facades.cga“ and represents only about 10% of the code in this single file. This small part of code is just used here to demonstrate the complexity of procedural systems. The code was re-formatted for easier reading.

B.2 The Floor-Planning Algorithm

In the following pseudo-code example the basic structure of the algorithm is shown. The ”GenerateRooms“ function is used to handle the complete generation of all rooms of a building. A loop for each floor, which contains a new room-definition is used to generate all different floor-layouts. For each floor, the layouter-grid is initialized first and used in the ”SubdivideRoom“ function. After the available space in the floor is assigned to rooms, all generated rooms are determined with the use of the layouter-grid. If procedural-rules are attached to rooms in the visual rule-editor, they are then attached to the created RoomNodes. The walls are then created before any other step of the generation-process is performed .Those next steps may change the size and positions of the created walls. The last step inside the first loop is the creation of the basic geometric objects for all floors and walls.

In the next step of the algorithm all vertical-connections are placed inside the defined rooms. Modified walls and rooms are updated at last.

The SubdivideRoom function is used to subdivide a given 'space' into sub-space. Please see figure 3.18 for a visual description of the process. If the room has child-rooms, those child-nodes are attached to the current room to be able to recursively subdivide those child-rooms if needed. If child-rooms exist, the space of the current room is then subdivided. At first the current room is completely removed from the layouter-grid to create free space in the grid. The start-positions for the child-rooms are calculated and those start-positions are then used to expand the child-rooms until no more free space is left. To ensure the defined connectivity between the rooms, the connector-room is then expanded, so that all other rooms are adjacent to the connector-room. The room-outlines are then calculated by using the assignments of the grid-cells to the rooms and the possible connection-positions between the rooms are calculated.

B.3 The Straight Skeleton Algorithm

This pseudo-code is part of the creation of the hipped roofs for the buildings. The "CalculateHippedRoofPolygons"-function uses the outline of the floor-definition to calculate a straight-skeleton. The straight-skeleton for the polygon is then used to create the individual roof-parts in the function "BuildPolygons" at the end. The straight-skeleton algorithm allows different angles of the individual roof-parts to create realistic buildings. The first step in the calculation is the calculation of the ridge-directions of the remaining layout. The remaining layout is becoming smaller in each iteration of the 'repeat'-loop. Next, all so-called 'events' are calculated, i.e. all possible edge-events and split-events, and their height of occurrence is stored with them. The next two loops only use the events with the lowest stored height. A split-event possibly splits the current remaining layout into two smaller ones, the edge-events usually just create new smaller remaining layouts. If the remaining layout does not exist anymore, the next steps in the creation-process are executed.

Algorithm B.1: Part of the code of a CityEngine example scene

```
FloorMass –>
    FloorMass(1, 2)
FloorMass(idx, n) –>
    set(floorIdx, idx) set(nFloors, n)
    comp(f){ side: FloorSide }
    Ceilings
FloorSide –>
    setupProjection(0, scope.xy, ~4, ~4, 1)
    split(y){ getWallBottom: Wall | ~1: FloorPattern | Wall_Top: Wall }
FloorPattern –>
    case scope.sx < winW+walW:
        Wall
    case front && Balconies == "On Front" || rear && Balconies == "On
    Rear":
        BalconyPattern
    case Side_Pattern != "Same as Main":
        case left || right: SidePattern else: MainPattern
    else:
        MainPattern
MainPattern –>
    case adjacentToBalconiesOnRight:
        split(x){ ~1: MainPatternDispatcher | windowsOnCorners*walW: Wall
        | balconyOnCorners*(balW+walW)/2: BalconyTile }
    case adjacentToBalconiesOnLeft:
        split(x){ balconyOnCorners*(balW+walW)/2: BalconyTile |
        windowsOnCorners*walW: Wall | ~1: MainPatternDispatcher }
    else:
        MainPatternDispatcher
MainPatternDispatcher –>
    case mainPattern == "[WO]*W":
        split(x){ { ~walW: Wall | winW: Tile }* | ~walW: Wall }
    case mainPattern == "o[WO]*Wo":
        split(x){ winW/2+walW/2: Tile | { ~walW: Wall | winW: Tile }* |
        ~walW: Wall | winW/2+walW/2: Tile }
    case mainPattern == "O[Wo]*WO":
        //split(x){ winW+walW/2: Tile | { ~walW: Wall | winW/2: Tile }* |
        ~walW: Wall | winW+walW/2: Tile }
        split(x){ winW+walW/2: Tile | { ~walW: Wall | panW: Tile }* |
        ~walW: Wall | winW+walW/2: Tile }
    case mainPattern == "wo[WO]*Wow":
        //split(x){ ~walW/2: Wall | winW/2: Tile | { ~walW: Wall | winW:
        Tile }* | ~walW: Wall | winW/2: Tile | ~walW/2: Wall }
        split(x){ ~walW/2: Wall | panW: Tile | { ~walW: Wall | winW: Tile }*
        | ~walW: Wall | panW: Tile | ~walW/2: Wall }
    case mainPattern == "Wo[WO]*WoW":
        . . .
```

Algorithm B.2: Floor-layout algorithm

```
function GenerateRooms(building object)
    Data: room-definitions, ro om-hierarchies
    Result: Floor layout, defined rooms, vertical connectors
    foreach existing floor-definition do
        Initialize the layouter-grid
        SubdivideRoom (floor)
        Get all generated rooms
        foreach generated room do
            Attach defined procedural rules to room
        end
        CreateWallsForRooms (generated rooms)
        for number of floors with same room-definition do
            Generate new floor object
            Add rules to new floor
            Add floor to building
        end
    end
    Get all floors with different room-definitions
    foreach floor with new room-definition do
        PlaceVerticalConnections (floor)
        Update the walls of the rooms where needed
        Update the room-neighbors where needed
    end
function SubdivideRoom(room)
    foreach child room do
        Attach room to current room
    end
    if room should be subdivided then
        Remove room from layouter-grid
        Calculate start-points of child-rooms inside current room
        Fill current room space with child-rooms
        Ensure connectivity to child connector-room
        Calculate room outlines
        Calculate possible room-connection positions
        foreach child room do
            SubdivideRoom(child room)
        end
    end
```

Algorithm B.3: Straight skeleton algorithm

```
function CalculateHippedRoofPolygons(building object)
    Data: building layout, roof-part definitions
    Result: roof-parts
    Get layout of the building
    repeat
        Get remaining boundary
        Create basic roof-parts from outline
        if not first iteration then
            Search for parent roof-part
            Attach current roof-part to parent
        end
        Calculate the directions of all ridges
        CalculateEventsForLayout(remaining boundary, roof-parts)
        Shrink current boundary according to minimal event height
        foreach active calculated split-event do
            Use split event to split the current layout into sub-layouts
        end
        foreach active calculated edge-event do
            Use edge event to merge the vertices and create new layout
        end
        Remove empty layouts
    until no boundary exists
    BuildPolygons(roof-parts)
```

Bibliography

[AAAG95] Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. A novel type of skeleton for polygons. *j-jucs*, 1(12):752–761, dec 1995.

[Cep10] Miguel Cepero. Procedural world. http://procworld.blogspot.co.at/, 2010.

[CEW+08] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. Interactive procedural street modeling. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 103:1–103:10, New York, NY, USA, 2008. ACM.

[Cha74] George Chaikin. An algorithm for high speed curve generation, 1974.

[Cit13] Cityengine: Philadelphia example. http://www.arcgis.com/home/item.html?id=78e67939ae7d42a294d734ee2fd427d0, 2013.

[Dav12] Ashley Davis. Networkview: A wpf custom control for visualizing and editing networks, graphs and flow-charts. http://www.codeproject.com/Articles/182683/NetworkView-A-WPF-custom-control-for-visualizing-a, 2012.

[Dou] Andrew Doull. The death of the level designer. http://njema.weebly.com/uploads/6/3/4/5/6345478/the_death_of_the_level_designer.pdf.

[Eve] Eve online. http://www.eveonline.com/.

[FKMP03] Pavol Federl, Radoslaw Karwowski, Radomir Mech, and Przemyslaw Prusinkiewicz. L-systems and beyond. In *Proceedings of ACM SIGGRAPH 2008*, 2003.

[FMLW14] Lubin Fan, Przemyslaw Musialski, Ligang Liu, and Peter Wonka. Structure completion for facade layouts. *ACM Transactions on Graphics (ACM SIGGRAPH Asia 2014)*, 33(6):210:1–210:11, nov 2014.

[FRS+12] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. Example-based synthesis of 3d object arrangements. *ACM Trans. Graph.*, 31(6):135:1–135:11, November 2012.

[HBW06] Evan Hahn, Prosenjit Bose, and Anthony Whitehead. Persistent realtime building interior generation. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, Sandbox '06, pages 179–186, New York, NY, USA, 2006. ACM.

[Hel] Hellgate london. http://www.hellgatelondon.com/.

[HWM+10] Simon Haegler, Peter Wonka, Stefan Müller, Luc Van Gool, and Pascal Müller. Grammar-based encoding of façades. *Computer Graphics Forum*, 29(4):1479–1487, 2010.

[IMAW15] Martin Ilcik, Przemyslaw Musialski, Thomas Auzinger, and Michael Wimmer. Layer-based procedural design of facades. *Computer Graphics Forum*, 34(2):2015.

[Joh10] Angus Johnson. Clipper. http://sourceforge.net/projects/polyclipping/, 2010.

[KW11] Tom Kelly and Peter Wonka. Interactive architectural modeling with procedural extrusions. *ACM Trans. Graph.*, 30(2):14:1–14:15, April 2011.

[LSWW11] Markus Lipp, Daniel Scherzer, Peter Wonka, and Michael Wimmer. Interactive modeling of city layouts using layers of procedural content. *Computer Graphics Forum (Proceedings EG 2011)*, 30(2):345–354, apr 2011.

[LTS+10] Ricardo Lopes, Tim Tutenel, Ruben M. Smelik, Klaas Jan de Kraker, and Rafael Bidarra. A constrained growth method for procedural floor plan generation. In *Proceedings of GAME-ON 2010*, Leicester, United Kingdom, nov 2010.

[LWW08] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics*, 27(3):102:1–10, aug 2008. Article No. 102.

[Mar06] Jess Martin. Procedural house generation: A method for dynamically generating floor plans. In *Symposium on Interactive 3D Graphics and Games*, 2006.

[MSK10] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. In *ACM SIGGRAPH Asia 2010 Papers*, SIGGRAPH ASIA '10, pages 181:1–181:12, New York, NY, USA, 2010. ACM.

[MSL+11] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. Interactive furniture layout using interior design guidelines. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 87:1–87:10, New York, NY, USA, 2011. ACM.

[Mut10] Alexandre Mutel. Sharpdx. http://sharpdx.org/, 2010.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *Acm Transactions On Graphics (Tog)*, 25(3):614–623, 2006.

[MWW12] Przemyslaw Musialski, Michael Wimmer, and Peter Wonka. Interactive coherence-based façade modeling. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2012)*, 31(2), 2012.

[Oys12] Bjorke Oystein. Helix toolkit. https://github.com/helix-toolkit/, 2012.

[Pea11] Matt Pearson. *Generative Art*. Manning; Auflage: Pap/Psc (28. Juni 2011), Shelter Island, NY, USA, 2011.

[Per] Markus Persson. Minecraft. https://minecraft.net/game.

[PL90] P. Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[PM01] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 301–308, New York, NY, USA, 2001. ACM.

[PYW14] Chi-Han Peng, Yong-Liang Yang, and Peter Wonka. Computing layouts with deformable templates. *ACM Trans. Graph.*, 33(4):99:1–99:11, July 2014.

[Reg] Stephen Regelous. Massive. http://www.massivesoftware.com/.

[Smi09] Josh Smith. Patterns - wpf apps with the model-view-viewmodel design pattern, 2009.

[SPD11] Nathan Sorenson, Philippe Pasquier, and Steve DiPaola. A generic approach to challenge modeling for the procedural creation of video game levels. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(3):229–244, 2011.

[Spe] Speedtree. http://www.speedtree.com/.

[Sta] S.t.a.l.k.e.r.: Shadow of chernobyl. http://soc.stalker-game.com/.

[Uni] Universe sandbox 2. http://universesandbox.com/.

[VAW+09] Carlos A. Vanegas, Daniel G. Aliaga, Peter Wonka, Pascal MÃijller, Paul Waddell, and Benjamin Watson. Modeling the appearance and behavior of urban spaces. In M. Pauly and G. Greiner, editors, *Eurographics 2009 - State of the Art Reports*. The Eurographics Association, 2009.

[WMWG09] Basil Weber, Pascal MÃijller, Peter Wonka, and Markus Gross. Interactive geometric simulation of 4d cities. *Comput. Graph. Forum*, 28(2):481–492, 2009.

[WWSR03] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics (TOG)*, 22(3):669–677, 2003.

[XFT+08] Jianxiong Xiao, Tian Fang, Ping Tan, Peng Zhao, Eyal Ofek, and Long Quan. Image-based façade modeling. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 161:1–161:10, New York, NY, USA, 2008. ACM.