HAW HAMBURG

BACHELOR THESIS
Jan Christopher Werner

A Block-Based Approach for Modeling Buildings Using the CGA Shape Grammar

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jan Christopher Werner

A Block-Based Approach for Modeling Buildings Using the CGA Shape Grammar

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Angewandte Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Philipp Jenke
Supervisor: Prof. Dr. Bettina Buth

Submitted on: 17st of June, 2024

Jan Christopher Werner

Thema der Arbeit

A Block-Based Approach for Modeling Buildings Using the CGA Shape Grammar

Stichworte

Kurzzusammenfassung

Diese Thesis präsentiert eine blockbasierte Herangehensweise für die CGA Shape Grammatik. Kern der Arbeit war die Konzeptualisierung einer visuellen Programmiersprache. Eine Prototyp Anwendung wurde entwickelt, um die praktische Anwendbarkeit des Konzepts zu demonstrieren und eine Bewertung zu ermöglichen. Der Prototyp integriert die block-basierte Sprache mit einem existierenden CGA Shape Framework, um dem Nutzer jederzeit eine 3D-Szene mit der resultierenden Struktur darstellen zu können. Dabei wurde Wert darauf gelegt, dass das Grundgerüst der blockbasierten Sprache und der visuelle Editor als eigenständige Komponente wiederverwendet werden können. Hierfür wurde die domänenspezifische Logik von der allgemeinen Struktur der visuellen Sprache getrennt gehalten. Die visuelle Sprache unterscheidet sich von typischen blockbasierten Sprachen in dem Punkt, dass sie beide verfügbaren Dimensionen nutzt, um die baumartige Struktur der Grammatik zu verdeutlichen. Die Sprache und der Prototyp wurden mit einer heuristischen Evaluierung bewertet, um die Gebrauchstauglichkeit und Funktionalität zu beurteilen.

Jan Christopher Werner

Title of Thesis

A Block-Based Approach for Modeling Buildings Using the CGA Shape Grammar

Keywords

CGA Shape Grammar, 3D Modeling, Buildings, Procedural Modeling

Abstract

This thesis presents a block-based approach for the CGA shape grammar. The core of this work is the conception of a visual programming language. A prototype application was developed to demonstrate the practical application of the visual language. The prototype integrates the language with an existing CGA shape framework, in order to render a 3D scene of the resulting structure. The visual editor of the application was developed with reusability in mind. I architecturally separated the editor from the domain specific logic in order to obtain a visual editor framework for block-based languages. The block-based language differs from typical block-based languages in the sense that it utilizes both available dimensions to reflect the tree-like structure, which the rules of the grammar form. The language and prototype were evaluated with a heuristic approach to assess its usability and functionality.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In the landscape of computer graphics, the demand for efficient and user-friendly modeling tools has been on the rise. Traditional modeling software incorporates direct visual editing of 3D models by directly manipulating them in a 3D view. But modeling individual structures this way takes a lot of human effort, especially when the goal is to model entire cities. Although grammars capable of easing the workload by procedurally generating shapes, or even buildings and cities with a rule-based approach were introduced quite some time ago, it is still common to model cities building by building.

Procedural content generation is the research field that works on reducing the effort and opening new modeling opportunities. A part of this research field is the research of shape grammars. Significant progress has been made in this field, but a recent study identified a gap between the state of the art research and the level of applicability for them to be an attractive design tool for non-experts [3].

The field originated from L-Systems. They were originally meant to model the organic structure of filamentous organisms, but they also proved useful to also generate compelling representations of much larger organic shapes like weeds, plants and even trees. Shape grammars then further built on the idea of using rules and combined them with simple shapes, to then generate more complex structures. One of these shape grammars is the CGA shape grammar. The CGA shape grammar generates buildings and urban environments in this rule-based way, while being way more efficient than the traditional 3D modeling tools. This is because the rule based approach allows the user to define the characteristics of a building rather than a specific manifestation of those characteristics.

In this thesis I want to present a block-based approach that tries to improve the usability of shape grammars especially for people novice to this concept.

1.2 Problem Formulation

Formal grammars evolved from the idea of generating strings with rules. The shape grammars added a powerful visual interpretation of these strings, but like programming languages they are completely text-based. New users have to learn the syntax and semantics of a grammar in order to use it. This is a barrier of entry especially for non-programmers which are not used to writing grammar code. Furthermore, shape grammars also usually have very specific use cases, limiting the reward of learning them in the first place. This barrier of entry might be one of the reasons that limited the adoption of shape grammars.

Enabling non-programmers to code has become its own research field. There are approaches to lower the amount of necessary code which is called low-code. Other approaches avoid coding completely, those are referred to as no-code solutions. One way to avoid text-based coding entirely is to replace it with a visual approach that utilizes patterns already known by non-programmers. One such approach is visual programming. Visual programming replaces the text-based approach with visual elements that can be arranged with drag and drop. The resulting visual language is therefore referred to as a visual programming language (VPL). A VPL type that is known to be very easy to use is block-based programming, which harnesses the fact that people already know how to arrange jigsaw puzzle-like pieces.

Modeling buildings is a common process in city and urban landscape design, and the CGA Shape Grammar is a powerful tool for generating diverse structures and cities through rule-based design. However, like any text-based grammar or programming language, effective use of CGA Shape Grammar requires learning its syntax and semantics.

I want to explore to lower the barrier of entry and make it more appealing for new users to explore the possibilities of procedural building generation using the CGA shape grammar. In this thesis I therefore present an approach for simplifying the process of generating buildings or other urban 3D structures through the integration of block-based programming with the CGA shape grammar.

1.2.1 Contextualizing the Problem

End-User Development is a research field that is concerned with enabling end users to develop software systems [8]. This field is currently experiencing significant growth. No-

tably, major players like Google and Microsoft are investing heavily in low-code solutions, exemplified by products such as Microsoft PowerApps and Google AppSheet. Visual Programming is a technique in the End-User Development field that allows users to develop software systems using already well known patterns like drag and drop and their domain specific knowledge.

CGA Shape Grammar stands out as a powerful tool for generating architectural models for computer games, movies, and urban design, but like all text-based programming languages and grammars, there is a learning curve before someone can use it effectively. Specifically, students at HAW Hamburg studying applied computer sciences, who delve into computer graphics, may find it daunting to first learn how to write CGA Shape Grammar in order to experience its capabilities.

1.2.2 Wider Relevance

While visual programming languages (VPLs) weren't able to challenge text-based programming languages in professional software development, they still have become widespread in use when non-programmers are the target group. Especially in popular modeling tools like for example Blender VPLs are utilized to allow designers fast and efficient usage of powerful functions without writing any code.

In the architecture domain algorithmic design has become a major influence and architects also seem to favor visual approaches over text-based languages [16].

With the rise of touch-based devices, VPLs may also be better suited to provide an intuitive introduction to the CGA shape grammar, than a text-based approach could provide on these devices.

1.3 Research Goal

I want to empower individuals unfamiliar with the CGA Shape Grammar, particularly students at HAW studying applied computer sciences, to explore its possibilities and utilize it for designing structures and buildings. The goal of this thesis is to achieve this by improving the usability of the CGA shape grammar, optimizing the first-time user experience, and designing a visual programming language that comprehensively maps all of the existing features of the CGA shape grammar. Through this, I seek to contribute to

the broader discourse on the evolution and application of visual programming languages in the realm of computer graphics and urban design.

1.4 Structure of this Thesis

The thesis is structured as follows. First I cover the necessary basics my work is based on and my search for similar works in the domain. Assuming these basics are known, I then walk the reader through the concept of the VPL I designed for the CGA shape grammar. To aid the evaluation of the VPL, I implemented a prototype application. The application, its structure and the engineering process is outlined in the realization chapter. The evaluation assesses the VPL and the prototype to determine if the research goal was fulfilled and to identify areas for improvement in both the VPL design and the prototype. Finally I summarize the findings in the conclusion chapter and provide my perspective on how the prototype could be further developed and improved.

2 Basics

2.1 Introduction to Procedural Modeling and the CGA Shape Grammar

This section introduces the formal approaches that led up to the development of CGA Shape Grammar.

Procedural modeling, also referred to as generative modeling, is the research field in computer graphics where models and textures are generated using algorithms rather than manual editing. It has been developed in order to be able to generate highly complex objects based on a set of formal rules [6].

The idea is that manually modeling takes a lot of time and most visual objects share similarities that can be generalized by rules. Today, procedural modeling plays a huge role in urban design, the film industry, and video games. Procedural modeling is capable of creating compelling visual representations of plants, textures, buildings, cities, landscapes, planets and even star systems. An example of state of the art procedural modeling is the procedural content generation in Unreal Engine 5. It allows artists to combine visual modeling with procedural modeling [17].

The origins of generative modeling lie in the 1950s. Back then the linguist Noam Chomsky attempted to give a precise characterization of the structure of natural languages. The work he undertook initiated a new research field called formal language theory. Formal grammars are a central part of procedural modeling, as they are the foundation for the rule-based approach. Although formal grammars were not able to describe natural languages, they have since been widely employed to describe the syntax of programming languages [4].

In the following subsection I want to show you a special formal grammar, which describes the growth of plants. This work laid the foundation for using formal grammars to procedurally generate visual structures with string rewriting rules.

2.1.1 L-Systems

Lindenmayer Systems (generally referred to as L-Systems) are a type of string rewrite mechanism introduced by Astrid Lindenmayer in 1968. Lindenmayer was a biologist and his intention was to develop a mathematical formalism of how filamentous organisms grow and branch cells [9]. The theory proved not only useful for its intended purpose of modeling the neighborhood relations between cells in small organisms like for example algae, but also for bigger ones like weeds and trees. A popular method to interpret the strings created by L-systems in a visual sense is the turtle geometry. Using turtle geometry together with L-systems results in compelling two and three dimensional visual representations of plants [18].

L-systems can be categorized into different classes. In the following two sections, I am going to describe to you the key concepts of how L-systems work to then define the DOL-system class, the simplest form of an L-system. Understanding the concept of how L-systems work is key to understanding how shape grammars work.

Description

To understand L-systems one needs to know the terminology and key concepts first. Like all formal grammars, L-systems operate on a string of letters and apply an algorithm to generate a new string. Lindenmayer refers to these strings as words. L-systems are divided into different classes of systems. The simplest class are those systems which are deterministic and context-free. This class is referred to as DOL-systems. The D stands for deterministic, which means the result will always be the same.

I will now give an informal description of the terminology and of how L-systems operate.

The set of all letters, which can appear in such a string is often referred to as the alphabet. In this context the alphabet can also include other symbols like for example square brackets, they still get referred to as letters. L-systems require an initial string of these letters, this string is referred to as the axiom. L-systems then consist of a set of

production rules. A production rule is a mechanism that describes how a symbol should be replaced. They are able to replace a letter with a string. L-systems apply these production rules in iterative steps. During an iterative step, every production rule gets applied to every character of the string. After this step, the L-systems generated a new string. We refer to these generated strings by naming them after the amount of iterative steps that were needed to generate them.

Next I provide a more formal definition of the DOL-system class.

Definition of DOL-systems

Prusinkiewicz und Lindenmayer defines the DOL-system class in the book 'The algorithmic Beauty of Plants' as follows. [19].

A string OL-system is an ordered triplet.

$$G = (V, w, P) \tag{2.1}$$

where

- V is a collection of letters the system operates on called the alphabet.
- w is a non-empty string made out of letters from the alphabet. This is the starting word of the system which is referred to as the axiom.
- P is a finite set of productions. Productions are rules which define how the letters of a string are going to be rewritten.

A Production is defined as following:

$$a \to \chi \tag{2.2}$$

where

- a is a letter from the alphabet V
- χ is a string made of letters from the alphabet. χ can be empty.

The rule says the letter a is rewritten by the string χ.

Every OL-system that has not more than one rule for each letter in the alphabet is deterministic and is referred to as a DOL-system. If there is no rule for a letter the identity rule is applied which replaces the letter by itself which ultimately changes nothing.

A derivation is defined as following:

$$\mu \Rightarrow \upsilon \tag{2.3}$$

which reads υ is directly derived from μ.

where

- μ is an arbitrary word made out of letters from the alphabet.
- υ is a word that is the result of the concatenation of the applied productions to each letter in μ. Special to L-Systems compared to Chomsky grammars is that productions are applied in parallel.

The derivation can again be applied to v.

Deriving w results in a derivation of length 1. Deriving the result again results in a derivation of length 2. This way it is possible to refer to a derived string by how many derivations were necessary to generate it from the axiom w

Turtle Interpretation of Strings created by L-Systems

The letters of L-systems can be interpreted in different ways. A sophisticated but easy to understand method to interpret them is the turtle interpretation. To understand the idea of turtle interpretation, the turtle can be visually figured as a metaphor. The turtle has a state, which is defined by a position and a heading of where the head of the turtle is facing. Given a specified step size and angle increment, the turtle can respond to commands represented by letters of a defined alphabet. The alphabet consists of four letters. The letter 'F' means to move forward a step while drawing a line as a trail. 'f' means to move forward without drawing. '+' means to turn left by the specified angle and '−' means to turn right vise versa.

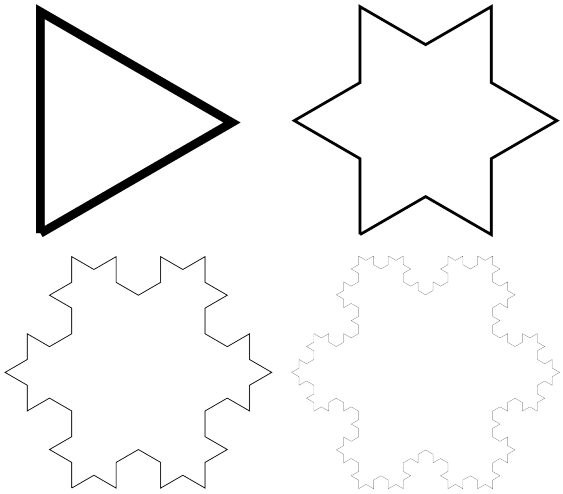

Figure 2.1: Koch snowflakes generated using Inkscape's L-system tool.

Given an initial state of the turtle, it can interpret a word of the alphabet letter by letter to draw a set of lines.

This enables L-systems to draw fractal curves like for example the Koch snowflake. The Koch snowflake can be defined by the following L-system using a single production rule.

$$w : F - -F - -F \tag{2.4}$$

$$p : F \Rightarrow F + F - -F + F \tag{2.5}$$

Interpreting the words using turtle graphics interpretation results in a snowflake-like curve, when using an angle of 60 degrees. Figure 2.1 shows this interpretation of the axiom and the generated words after three iterations.

2.1.2 Shape Grammar

Shape grammars were introduced in 1971 by Stiny und Gips. The paper defined shape grammars as sets of rules for generating shapes. These grammars used painting rules to paint the areas contained in a shape and sculpting rules to modify shapes. The shape grammars were meant to use formal, generative techniques to generate art and to develop an understanding of what makes good art objects [20].

2.1.3 Split Grammar

In 2003 Wonka et al. introduced split grammars as a novel method to generate buildings. The presented split grammar was based on shape grammars, but focused on the idea of splitting existing shapes into smaller ones in order to create details. This approach was very successful at generating city-like facades for buildings.

For this they used two grammars that worked in conjunction. A design grammar referred to as split grammar and a control grammar. The split grammar only allowed a certain type of rules, while the control grammar handled spatial distribution of the design ideas like for example setting different attributes for different floors of a building. Users of the grammar can influence the generation process by directly modifying both the split grammar and the control grammar, or by modifying attribute values that influence the generation process [21].

2.1.4 CGA Shape Grammar

The CGA shape grammar was presented by Müller et al. as a combination of the ideas of the previous work by Parish und Müller and Wonka et al.. Parish und Müller had presented an approach to build urban environments by placing simple mass models and using shaders to add details, while the split grammar presented by Wonka et al. was able to generate geometric details for an individual building. The CGA shape grammar is able to generate complex mass models with detailed surfaces. It does this by starting with a building lot. Mass models can then be added to the lot using rules like split, extrude, rotate or translate. These rules allow switching between a 2D and 3D context which enables creating complex mass models. To then allow for a consistent detailing of these mass models, the grammar has context sensitive rules like the repeated-split, which allow placement of doors and windows without intersecting walls. [13].

2.2 Introduction to Visual Programming Languages

Kuhail et al. combined the widely known taxonomies in order to get a taxonomy that is able to characterize every VPL while also avoiding similar categories. They came up with four different categories. In this section I give a short description of these categories so that I will later be able to explain why I chose a block-based approach [7].

2.2.1 Form-Based Programming

Form-based programming is a method to create and manipulate software applications by configuring a form. Users are able to create and configure cells in that form and add actions and triggers using drop-down menus or drag and drop. This approach is for example often used in the smart home context, where users can specify what devices should do when an event happens.

2.2.2 Icon-Based Programming

Icon-based languages capitalize on the use of icons to represent objects or actions. There are different types of icons. Elementary icons represent objects or actions. Complex icons are compositions of these elementary icons and allow for creation of visual sentences [1].

2.2.3 Diagram-Based Programming

Kuhail et al. categorize languages as diagram-based if they allow users to place graphical objects on a canvas using drag and drop. The user can then connect the objects with arrows or lines to create relations between the objects. Some of these languages are also generally referred to as flow-based languages, because the lines represent a directed data flow between these objects where the output of one object serves as the data input for others. Diagram-based languages are very versatile because they allow users a lot of freedom in placing objects and deciding how to connect them. A problem with this approach is that they are not able to provide a clear visual hint which connections are valid.

2.2.4 Block-Based Languages

The main characterization of block-based languages is that they allow the user to drag and drop blocks out of a predefined list into a workspace. Then the user is able to use these blocks to piece together a program kind of like a jigsaw puzzle which was the main inspiration for this type of VPL. The idea is that the user can build upon the well known concept of jigsaw puzzles and therefore focus on the concept of the programming language rather than the syntax. In fact the paradigm is able to completely prevent syntax errors. Popular block-based languages are Scratch [12] and Google Blockly [2].

2.3 Related Work

Lipp et al. introduced an interactive visual editing approach to the CGA shape grammar. The Approach introduced a solution for selecting components in the 3D view or in a tree-view. These components can then be modified using buttons to add rules. Rule parameters can be modified in a separate form using sliders, checkboxes and text fields [10]. This approach's main difference to the one introduced in this paper is that it does not utilize a VPL in any way.

Another approach was conducted by Patow. He identified that users need to traverse grammars similar to the CGA shape grammar as a graph. The author then introduced a graph-based method to enable user-friendly procedural modeling of buildings, which uses a grammar that loosely follows the notation of the CGA shape grammar. The result is a visual language for editing grammars that could be categorized as diagram-based programming [15].

3 Concept

In this chapter I lay out the conception of the VPL. I already set the research goal in the Introduction, which was to improve the usability of the CGA shape grammar in order to make it easier for users unfamiliar with the CGA shape grammar to get started learning its capabilities to build urban structures.

My approach to create the VPL was to find a suitable visual representation capable of mapping the essence of the CGA shape grammar to objects and connections.

The first step to do this was to decide which VPL approach would be the most suitable for mapping the grammar. I then list design principles that I used to guide the design and implementation process to ensure that the visual language has a good usability for novice users. These principles gave a necessary foundation to make user-friendly design decisions. After that section I go through the requirements for the VPL. In the final section of this chapter I conceptualized the VPL by identifying the concepts of the CGA shape grammar which I then transferred into the VPL.

3.1 Decision on the type of VPL

In this section I explain why I chose the block-based approach over other possible VPLs.

The VPL should be able to map all aspects of the CGA shape grammar but the focus is to maximize the first time user experience for new users not familiar with the grammar.

When writing multiple rules for the CGA shape grammar they form a tree-like structure. The outcome of a rule is largely influenced on where the rule is located in that tree. Rules that are located near the axiom shape will have a greater impact on the overall shape of the end result than those located way down the chain of rules. This is why I consider this tree-structure a very important thing I want the VPL to express.

There are two different classes of VPLs that excel at visualizing tree-like structures, the diagram-based and the block-based VPLs.

Diagram-based VPLs consist of visual objects and lines or arrows. The objects can be freely arranged on a canvas and the lines or arrows express the relation between the objects. This makes them very flexible to adapt to a lot of different domains. They are able to represent a tree-like structure, but they are also able to visualize other more complex structures. This flexibility and power of expression makes them a popular choice in a lot of projects.

Block-based VPLs consist of visual blocks. These blocks feature different connector types that express which blocks are compatible with each other and in what manner. They are also able to express tree-like structures. Popular Block-based languages target a novice audience, teaching them the key concepts of programming, without messing with syntax.

The flexibility of diagram-based approaches comes with added complexity to the user. The user is responsible for arranging the objects and then adding connections. It requires the user to repeatedly rearrange objects in order to keep them neatly arranged and avoid lines crossing other elements. Diagram-based VPLs are not able to visually express which connections are valid like block-based languages are able to with their connectors. Block-based VPLs excel at this. This makes connecting elements really easy. The block-based approach also allows the user to add new blocks directly to existing ones, which is simpler than adding the elements and then connecting and rearranging them afterwards. The blocks clearly express their compatibility through the shape of their connectors. Although the block-based approach is more limited than the diagram-based, the capabilities are certainly enough to enable non-expert users to generate structures. The limitation being that context sensitive blocks might not be possible. Block-based programming usually avoids context-sensitive structures in order to maximize usability for non-expert users.

Since the block-based approach has unique features targeted at reducing the cognitive load of the user, I choose this one over the diagram-based approach. The diagram-based approach provides more flexibility but at the cost of being more complex to configure. It requires the user to first place the objects to then create the necessary connections. The block-based approach is able to clearly express the compatibility of blocks through the connectors with each other using the connector shapes. This will ensure that the resulting visual language will be as easy as possible to use.

3.2 Design Principles

Design principles are a great way to avoid mistakes other people already have made and learned from. They provide guidance during the design process and a good foundation to base design decisions on. In the book "User-Centered Design" Lowdermilk elaborated on a number of important design principles. I will mainly use the following principles to justify my design decisions. [11]

Principle of Proximity

The principle of proximity states that we perceive relationships between objects that are closer together. This principle was useful to have in mind especially when deciding how and where to handle parameter editing.

Visibility, Visual Feedback and Visual Prominence

Visibility is when the application provides a visual focus to an element.This can be done using color, different sizes or typefaces. It can also be done using a loading indicator. Visibility is there to guide the user. Visibility can also be used to provide feedback to the user. For example in a form color can be used to signal an illegal status of a text field. This principle for example influenced my decision on how to communicate if an object has valid parameters.

Mental Models and Metaphors

A new user of an application will use his existing knowledge of how the application might work. During the design a developer should consider this and choose icons, language and behavioral patterns for the application that do not confuse but help the user understand the application. This principle is always important to keep in mind, when designing user interfaces.

Progressive Disclosure

It can be helpful to hide things in order to reduce the users' cognitive load. An example is to gray out options in a menu. I used this principle to avoid overloading the objects with input fields that would sometimes have no effect depending on the specific configuration situation of the parameters.

Consistency

The principle of consistency says that users learn and understand an application more easily when it is consistent with what they already know. This principle influenced the decision which VPL should be best suited for a novice user to the CGA shape grammar.

Affordance and Constraints

Real world objects are often designed in a way that constraints the user to use them improperly. Designing software in a way that limits the options or makes it impossible to do the wrong thing avoids confusion and frustration. This principle also influenced the decision of which VPL would be best suited and how to design it.

Hick's Law

The user will need exponentially more time to make a decision the more options the system presents to him. This is good to keep in mind when populating a menu with items. The more items there are the longer the user needs to find the right item. I kept this in mind, when deciding which and how many input fields would be acceptable.

Fitt's Law

The farther the user must travel between two objects, the less precise the user will be when reaching his target. This law can help determine the necessary size of an UI element. This principle influenced the size and margins of the VPL objects so that the user would be able to easily grab them using drag and drop.

3.3 Functional Design Requirements for the VPL

To design the VPL for the CGA shape grammar, I define requirements that the VPL concept then must fulfill in order to be able to maintain the capabilities of the grammar.

In this project I use user stories to set the focus and scope of the prototype but I also have set the goal to design a VPL that is capable of most or even all of the features that the CGA shape grammar has. In order to do that I need to analyze the existing grammar and make sure that the VPL has the same capabilities. For this I analyze the existing grammar and formulate design requirements that the VPL would need to fulfill to get feature parity.

I also list additional design requirements that stem from a combination of the requirements the existing grammar has and the design principles for creating a user-friendly block-based VPL.

Table 3.1: Requirements for the Visual Programming Language

R01	For each operation and for each shape there must be a visual block.
R02	Every block should have an identifying name that clearly describes what it does.
R03	A block must be able to have children blocks. Children blocks are blocks that got attached to a socket of the block.
R04	Blocks that require a minimum number of children blocks must always provide at least this amount of sockets so that the user can see that there are missing blocks.
R05	Blocks must always provide at least one empty socket unless there is a maximum capacity of attachable blocks.
R06	Every block must have exactly one plug which can be used to connect the block to a socket of another one.
R07	The parent blocks in the block language must adjust their size in order to be big enough to match the total size of the children blocks that got attached to it.
R08	When dragging a block with the mouse, the block must move accordingly
R09	blocks that are attached to a block that gets dragged must also get dragged so their position relative to the parent block position stays intact.
R10	When dragging a block and then releasing it onto the canvas the block gets either added to the canvas or to the closest compatible socket of an existing block.
R11	sockets and plugs that share the same geometry, must be attachable to each other.
R12	Blocks must be able to be in an invalid state, when the current information provided by the user is not enough to generate a rule.
R13	Blocks that are in an invalid state should provide the user hints on what to do in order to get into an complete state.
R14	The system must ignore invalid blocks for rendering.
R15	There must be a list of blocks from which the user can drag new blocks onto the canvas.
R16	The List of blocks must not change when the user drags a block from it.
R17	There must be a way for the user to remove blocks from the workspace.
R18	Operation blocks should be visually distinguishable from shape blocks.
R19	The user must be able to set a tree as the active one that gets rendered

3.4 Non Functional Requirements

It should be way faster to build a structure using the VPL compared to writing the rules in an editor. The user interface of the system should always feel responsive even when it is busy generating the 3D visualization. The VPL should be reusable for other future projects. Ideally the VPL design should be capable of preserving all of the features the CGA shape grammar has.

3.5 Designing the VPL

In the previous section I explained why I use a block-based VPL. In order to be able to design and implement a prototype of the VPL I used these design principles and the requirements. But first I needed to identify the key concepts of the CGA shape grammar to then build the key elements of the VPL.

3.5.1 Key concepts of the CGA shape grammar

The key concept of the CGA shape grammar is the rule. The grammar starts with an initial shape referred to as the Axiom. Rules are then used to alter existing shapes and generate new ones. The rule itself is made out of multiple components I will go into now. Here you can see the structure of the rule, square brackets are not actually part of the rule but I use them to mark optional parts.

```
predecessor-id [: condition] --> action [action ...] [: probability]
```

- **predecessor-id** is a string of characters which refers to the shape the rule is executed on.
- **condition** is an optional precondition that must be met in order for the rule to be able to be executed.
- **action** refers to the action to perform when the rule is executed.
- **action ...** means that a rule can perform more than one action.

- **probability** a probability can be used in order to declare multiple different rules that refer to the same predecessor-id. The probabilities of those rules must add up to 1.

Like the rule actions are furthermore made of a sum of parts.

```
name( [argument, ...] ) {successor-id, ...}
```

- **name** The name of the action
- **argument** A comma separated list of arguments.
- **successor-id** A list of ids for the successor shapes

In order to design the VPL I need to identify the two most important concepts of CGA shape grammar and look at how to turn them into the basic elements of a block-based language. Next I used this information to create the key elements of the block-based VPL.

3.5.2 VPL Elements

Blocks

The fundamental elements of the VPL are the blocks. Therefore I first identified which types of blocks the VPL should have. A VPL with feature parity would definitely require a Rule block. But I do not think the concepts of rules fit well to the mental model a new user has. The visual feedback principle states the importance of providing immediate visual feedback. I want the user to have immediate visual feedback in the 3D model he creates. A rule block would not be able to be used to generate any output until an action would be attached to it. Also in my testing of how the grammar behaves most rules actually only housed a single action. So in order to create a VPL that is as intuitive to use as possible, I identified the actions as the most important concept that is able to generate immediate results in the 3D view.

The other very important concept is the shape. The shapes are what the actions operate on. They provide scope and context for the actions and the final shapes are also directly visible in the 3D view. The user is expected to use them to identify where to apply an action in order to change the resulting model. Actions of the CGA shape grammar also

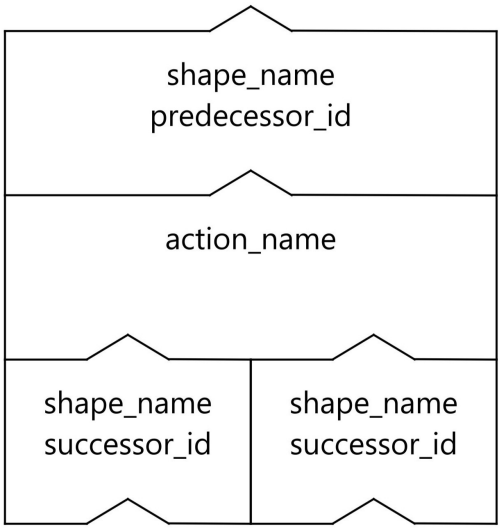

Figure 3.1: Example connection between a shape block with a single action block attached. The action block has multiple shape blocks attached to it which configure the number of successor shapes..

determine their behavior based on the amount of resulting shapes. I want the user to be able to add shape blocks to an action block to increase the amount of resulting shapes. A very good example of this is the split action. The grammar determines the amount of splits based on the amount of resulting shape ids.

These two block types, the action blocks and the shape blocks are what I chose to be the key blocks of the VPL. Using shapes and actions I want the user to be able to have a strong mental connection between the structure of the blocks and the resulting 3D view. Not having rule blocks creates a limitation, but that fits to the affordance and constraint principle. The user will still be able to generate more than enough structures. In case this limitation would not be considered worth it in the future, it would still be possible to introduce a rule block in the future to enable having multiple actions in a single rule and unlocking the full capabilities of the CGA shape grammar. The other principle I applied was the mental model, when I considered which concepts of the grammar should be reflected as block elements in the VPL.

The user will be able to arrange these block types like in the following draft.

Action blocks provide the ability to add multiple shape blocks so that the user can increase the number of successor shapes. On the other hand, shape blocks provide the

optional ability to add multiple action blocks to fulfill the probability part of the CGA shape grammar rule. When a probability is added to a rule, at least one other rule needs to exist which targets the same shape. In the visual language the equivalent is to add multiple actions to the same shape and give them a probability parameter. I will draft the parameter handling later on in more detail.

Block Connectors

Block connectors define which blocks are able to get connected. They can communicate these limitations to the user through their shape. It is important to only use visually distinct shapes. There is only a limited amount of visually distinct connector types one can imagine without confusing the user. This is a big limitation for block-based languages in general. It is not a problem to use distinct connectors to avoid the user being able to connect shape blocks with other shape blocks and action blocks with other action blocks. Trying to then also find distinct connectors for every shape the grammar offers is problematic, because having dedicated connectors for every single shape would require a large amount of different connectors. According to Hick's Law this results in the user requiring a large amount of time in order to find the correct block.

The compatibility of shapes and actions is determined by the constraints of the CGA shape actions. Table 3.2 is a matrix that displays the compatibility of actions with shapes.

Table 3.2: Action-Shape Compatibility Matrix

		Shape								
		detailShape	roofShape	dormerShape	gableShape	gableOrnamentShape	prismShape	subdivShape	geometryShape	polygonShape
Action	detail									X
	extrude									X
	gable									X
	gable_ornament									X
	dormer									X
	remove	X	X	X	X	X	X	X	X	X
	subdiv						X			
	roof									X
	transformation	X	X	X	X	X	X	X	X	X
	expand									X
	split						X			X
	repeated_split						X			X
	component_split		X	X	X		X			

I used Table 3.2 to search for patterns which shapes really need different connectors. From the matrix it became clear that the most important shapes for the actions are the polygon shape and the prism shape. To keep the number of blocks and the number of connectors manageable, I decided to limit the VPL to only these two shapes since they are enough to use most actions. I consider this limitation to be acceptable to evaluate the suitability of a block-based VPL for the CGA shape grammar.

Figure 3.2 shows the connectors I decided to use. There are two different types of connector shapes, one for 2D polygon shapes and another one for 3D prism shapes. For each of these types I then created two different variants. These variants ensure that shape blocks are not compatible with other shape blocks. Instead they are only compatible with action blocks and vice versa. The two variants are similar but easily distinguishable. This

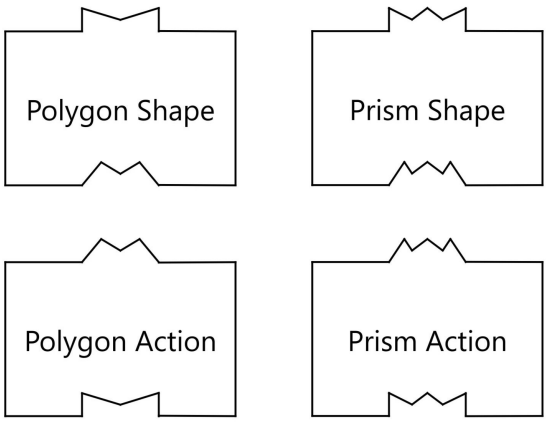

Figure 3.2: Different block variants showcasing the connectors

provides a visual hint to distinguish shape blocks from action blocks. Another visual hint I added is that connectors for 2D shapes have two corners, while connectors for 3D shapes have three corners. The actions capable of operating on both shape types get two blocks, one for every different shape type.

Parameters

The CGA shape grammar has a number of different parameters, which the VPL must be able to handle. There are different approaches to allowing the user access to the parameters of the grammar rules. Some block-based languages use blocks to let the user configure the parameters, but having dedicated blocks for expressions, variables and operations would dramatically increase the complexity of the VPL. Another solution would be to have a panel either as a popup or beside the workspace where the user could edit and configure the block parameters of the currently selected block. I consider having a popup interrupting the user not suitable for good usability. A dedicated panel would offer a lot of space to use for configuration. The downside of that is that the parameters are only visible for a single block at a time. Another option would be to add the panel for editing parameters into the blocks themselves. I want to try this approach since it allows the user quick access to all the parameters. It also ensures that the information of a block is close to the block itself, helping the user perceive a close relationship between the parameters of a block and the block itself as the Principle of Proximity states. Something to be careful about is to keep the number of options to a minimum. The user could feel overwhelmed by options and the size of blocks could become very large.

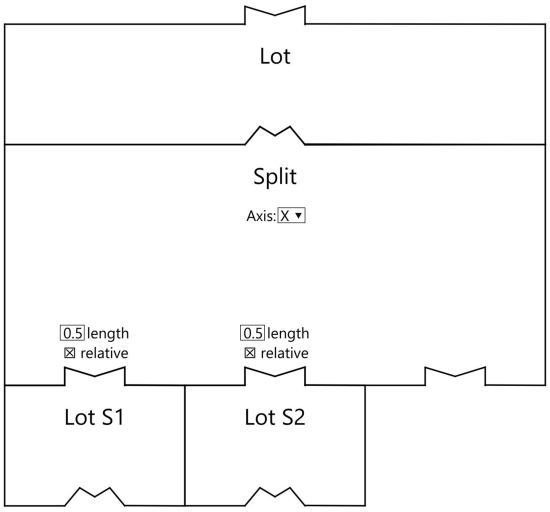

Figure 3.3: Split block draft showcasing a configuration parameter and two length parameters specific to the added successor shapes.

Shape blocks are the simplest. They only require an id parameter for the shape they represent.

Action blocks are more complicated. They need different kinds of parameters. They need to distinguish between parameters that configure the action in general and parameters that relate to successor shapes. I arranged configurational parameters in a vertical list inside the action block. Parameters that relate to successor shapes are arranged above the socket of the successor shape they influence. An example of these two different kinds of parameters can be seen in Figure 3.3.

There are a number of parameter types. There are numbers and predefined strings. Numbers can be absolute or relative. I decided that the user can change between relative or absolute interpretation by ticking a box next to the number which then enables relative interpretation when available. When only one interpretation is allowed the field is grayed out. For the selection of the predefined strings I decided on drop down selection menus. The advantage of these is that the user is not able to input invalid values and also he does not need to memorize the values. In some actions the selected option in the drop down menu influences the number of parameters of the action. Switching between these options hides the parameters that are not relevant in this case.

Since I decided against a dedicated rule block I also decided against adding the rule specific precondition or probability parameters. These are advanced concepts, which would be best housed inside a dedicated rule block in a possible future expansion of the VPL.

VPL Editor

The VPL editor consists of a block palette, a canvas and a 3D view. The palette houses all the blocks available to the user. The user can drag and drop new blocks from the palette onto the canvas. On the canvas blocks can get connected to existing ones in order to build a tree of blocks. I decided that it should be possible to build multiple trees of blocks. This enables maximum flexibility to not get in the way of how the user wants to arrange blocks. The 3D View is where the user is able to see the resulting model. Because of how the existing CGA shape grammar pipeline works the 3D view is only capable of rendering one tree of blocks at a time. This makes it necessary to distinguish between active and inactive block trees. The VPL is therefore able to highlight the active tree using color. The active tree is the one which got changed most recently.

4 Realization of the Prototype

This chapter presents the development and the implementation of this prototype.

With the Concept done the next task was to engineer and implement a prototype capable of showcasing the capabilities of the VPL design. Implementing the full concept for every action of the CGA shape grammar would have been too big of a task. I broke down the task into smaller chunks using user stories. I then arranged these in a user story map to define the scope of what the prototype should achieve. This was necessary to ensure that the prototype would be best fitted to support the evaluation of the block-based language. To develop the software I utilized a number of tools and design patterns. I will shortly introduce these first before then presenting the architecture for the prototype. The last section of this chapter focuses on the implementation of the prototype's individual components.

4.1 User Stories

To develop a prototype application for the block-based language, I first needed to set goals for what the prototype should achieve. I used an agile software development approach to develop the prototype. In agile development the application is developed in small iterative steps known as sprints. Sprints have a fixed timeline, at the end of which a new deployable version of the application should be ready. To define the goals for each sprint, I used user stories. User stories describe features from a users point of view and attach the value those features provide to the user. This approach helped me break down the overall task into smaller, manageable chunks which could be completed during a sprint. The value-oriented nature of user stories allowed for prioritizing features based on user value. To organize the user stories, I created a user story map.

Figure 4.1 shows the user story map. In this map I defined two sets of features for the prototype. A minimum set of features the prototype must provide in order to be evaluable

Figure 4.1: User Story Map showing the scope of the prototype application

later on and a set of features that would be nice to have but considered optional in case the time would not allow to implement all of them.

4.2 Development Environment and Tools

In this section, I go through the technical environment my project is based in. The prototype was developed to be part of the procedural content generation (PCG) project. It is a shared project that houses a collection of applications related to PCG. This project enables sharing knowledge and building on top of previous work. The main reason for being part of the PCG project was that it is home to the CGA shape framework that I required to implement the VPL. This had a big influence on the development environment and tools I used. In the following subsections I will detail these tools, their functions and how they contributed to my project.

4.2.1 Programming Language

The PCG project is written in Java. At the time of writing this the recommended version to build the project was Java 17. This meant that in order to be compatible with the existing project I also used this version to build the application. Another reason to use Java 17 was that it is a long term support version, which will still receive security updates during the next few years.

4.2.2 Versioning System

Multiple people work on the PCG project at the same time. This requires a mechanism to apply changes in a controlled manner. Git provides this mechanism. It allows for everyone to commit their changes on their own branch. The diverging branches then can be merged to end up with a version that contains all the changes. It highlights when there are conflicting changes and offers mechanisms to resolve them. It also allows reverting to a previous version in case a change breaks the software. I ensured to only commit the software in a working condition, because that simplifies backtracking of bugs down to the commit that first introduced them.

4.2.3 Build Tool

The project uses Gradle as the build tool. Gradle allows to exactly specify the dependencies required to build the project. This allows other programmers to also build the application from the command line without having to worry about downloading and setting up the right dependencies first. It also enables future use of continuous integration. Continuous Integration can automatically build and test the application every time a commit is uploaded and notify the developers if there is an issue.

4.2.4 Development Environment

I decided to use Intellij IDEA as the integrated development environment (IDE). It offers a single user interface to configure the compiler, the versioning system and the build tool, while offering a source code editor with extensive linting and debugging capabilities. This simplified the setup of the development environment and offered helpful tools to increase productivity and ensure software quality.

4.2.5 Libraries and Frameworks

The most important framework I built upon was the CGA shape framework, which is part of the PCG project. This framework enabled me to focus on implementing the visual language. The CGA shape framework is able to generate the CGA shape grammar and further process it into a triangle mesh representation. To render this representation, I used the jMonkeyEngine. This engine can render the triangles generated by the CGA shape framework to present the user a 3D view of the resulting model. It also handles user input, allowing the user to change the camera view and look at the model from different angles. Another important framework I used was Java Swing, which provided the basis for the VPL and the user interface of the application. Although Java Swing has been superseded by the more capable JavaFX, I decided to use it because it is already a dependency of the PCG project and works well in conjunction with the jMonkeyEngine.

4.3 Design Patterns

Design patterns are standard solutions for common software engineering problems. It is good practice to use them whenever they provide a good fit to a problem. This improves maintainability since most software engineers know these patterns. It also enables us to talk about the software since they introduce terminology. I used several design patterns to develop the prototype. In this section I describe the most important patterns I used and how they helped me develop the prototype.

4.3.1 MVC Pattern

The Model-view-controller Pattern (MVC Pattern) is a software design pattern. It can be used on different scales, on an architectural level for the entire application, but also on a smaller scale for designing a smaller component. In the case of the prototype the pattern is used for the overall architecture of the prototype.

The pattern separates an application into three main components. These components are the model, the view and the controller. This separation used to manage complexity by dividing three main responsibilities into separate components. This positively affects reusability of the code and helps structure it in an understandable manner. It is a commonly known pattern, which further improves the maintainability when multiple developers need to understand the software.

There are different interpretations of this pattern, but all agree that the model is responsible for managing data, to encapsulate the application's state. The view is responsible to display data to the user and to receive user input. The controller is the central component that initializes all the other components and connects to them. It is responsible to handle the user input and then apply the necessary changes to the model.

4.3.2 Observer Pattern

The observer pattern works kind of similar to how newspapers get distributed. The people that write and publish the newspapers do not need to know who is reading the newspapers. They only have to provide a subscription mechanism. The people who want to receive newspapers use this subscription mechanism to announce that they want

to receive them. If someone doesn't want to receive newspapers anymore he then can unsubscribe by himself.

This can be translated to software components. There are observer and observable components. Observable components provide a subscription mechanism to subscribe and unsubscribe and they update the observer when necessary. Using Interfaces for observer and observable enables to hide any implementation details of the observers from the observables.

I used this mechanism to let the model update views directly instead of letting the controller handle both of this. This reduced the complexity of the controller while still hiding implementation details of the views from the model. It also allows adding additional observers.

4.3.3 Composite Pattern

The composite pattern allows to compose objects into trees and then operate on every possible subtree as if it was a single object. The Swing framework uses this pattern to build user interfaces by nesting components. This allows Swing to handle the resulting trees of components as if they were a single component. The framework is built and optimized around this idea, therefore I utilized this to implement the block-based VPL in a way that is easier to handle, flexible and efficient to render.

4.4 Software Architecture

4.4.1 External Components

As I mentioned before when describing the libraries and frameworks, I used other software libraries and frameworks to develop the prototype. I will now describe how the prototype connects to these external components.

As you can see in Figure 4.2, I decided to base my application on the CG3DApplication class. This is an implementation for handling 3D scenes with the jMonkeyEngine. It is used by many other applications in the PCG project. This aligns the behavior of how the 3D scene is displayed and how user input specific to it is handled. This means the application also uses Swing and the jMonkeyEngine. Swing is used for the user

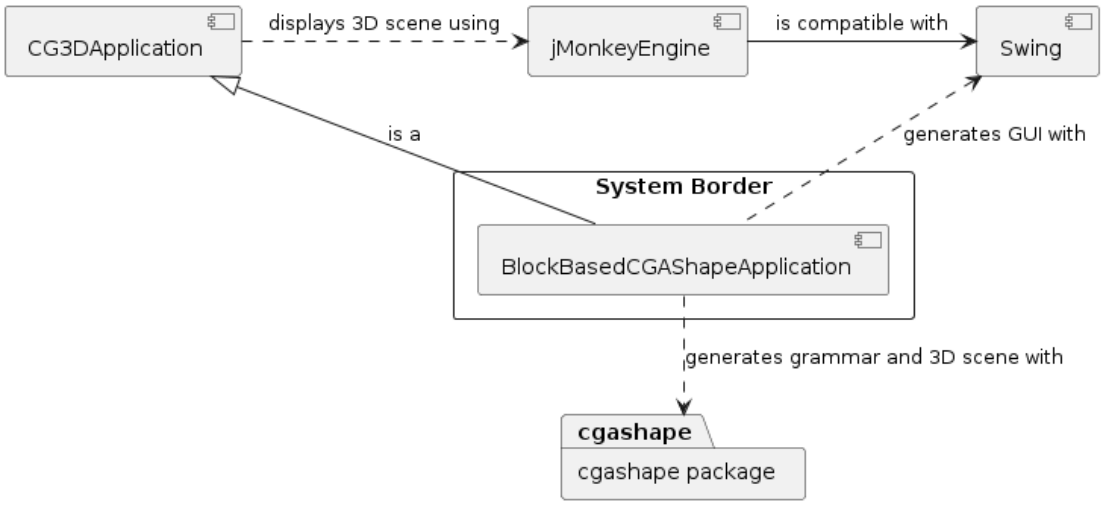

Figure 4.2: Component diagram describing the relation of the software system to its neighboring systems.

interface. Since Swing is only capable of handling 2D content, the jMonkeyEngine is used to display the 3D scene. The jMonkeyEngine is designed in such a way that the scene can be embedded inside a Swing component. The application uses the cgashape library to build the grammar and generate the 3D scene.

4.4.2 Internal Component Structure

For the internal structure of the prototype I separated the block-based editor into an independent component. This is because I want the block-based editor to be reusable as a framework for other applications that might need a block-based editor. The editor provides the abstract structure and behavior of how blocks look and behave, but it does not know the blocks' semantics. The semantics are added by the application that uses the editor. In my case the application extends the abstract blocks with CGA shape specific ones and adds the grammar builder as the component that specifies the semantics. The GrammarBuilder observes the canvas and translates a tree of blocks to CGA shape grammar using the cgashape framework.

The block-based editor uses the MVC pattern to separate presentation related logic from editor and block behavior and state. The pattern divided the editor into three components, the model, the view and the controller. The model contains the state of the editor. It does not know of any of the other components, which makes it easier to design

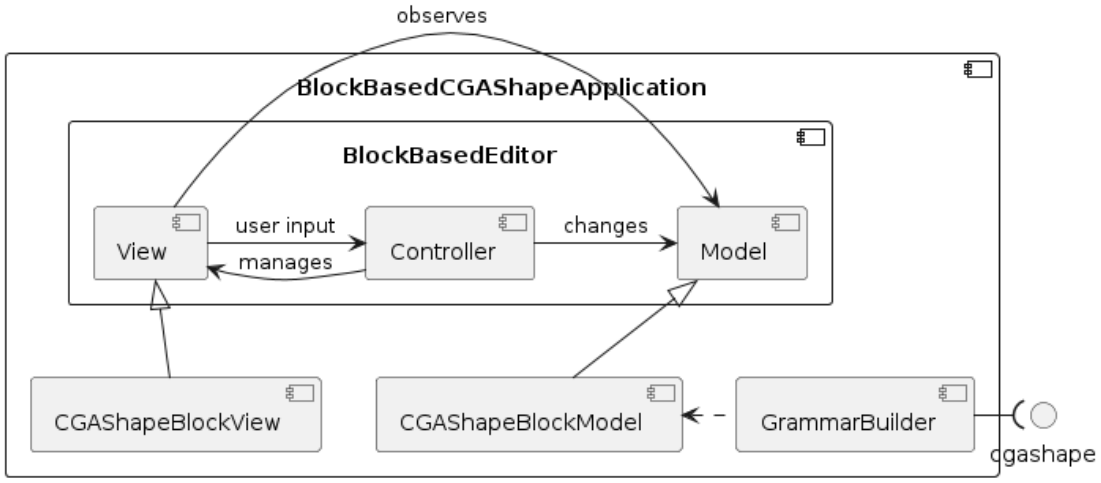

Figure 4.3: Component diagram describing the internal architecture of the prototype.

and test. I combined this MVC architecture with the observer pattern to make the model observable and further reduce the responsibilities of the controller. I did this because it makes it easier to add application specific models without modifying the controller. It also added the possibility for the application to observe the model to get notified when the model changes. I used this mechanism to connect the grammar builder component to the model.

MVC Architecture

I chose a Model-view-controller (MVC) architectural design pattern for the block-editor framework to separate the responsibilities of the application into three main components, model, view and controller. This makes it easier for other developers to understand the structure of the code, which improves maintainability and reusability. As you can see in Figure 4.3 the architecture of the framework also affects the architecture for the overall application, since the application extends the framework with logic related to the CGA shape grammar.

I accompanied the MVC pattern with an observer pattern for the model, to further reduce the responsibilities of the controller. The intention was to enable applications to define specific views and models while keeping the controller application independent. By making the model observable I not only ensured views could update as soon as the model changed, I also provided a mechanism for the application to hook into for adding

the application logic. The builder component uses this mechanism to get notified when the model changes to then build and set a new grammar.

I implemented the observer pattern with interfaces. One defines the observer behavior and another one the observable behavior. These interfaces hide observer's specifics from the model, which is why the application can hook in additional observers. But there is a cost attached to the observer pattern. When using it one needs to be careful to make sure that views correctly unsubscribe to avoid issues like memory leaks or null pointer errors. The life cycles of block views and models are controlled by the controller component of the block framework. I did not want applications having to extend the controller to encapsulate the subscribing and unsubscribing part of the pattern. Applications can subscribe to the canvas model instead, which does not require unsubscribing as long as the editor and the subscribing component have the same life cycle as the application. In the case of the prototype this is the builder component, which uses the canvas model to get a tree of blocks and translate it into the CGA shape grammar.

4.5 Implementation

4.5.1 Implementation of the View Component

I used Swing to implement the user interface. The framework provided me with the tools and components to build most of the interface, but it provided no high level components specific to visual editors. I had to implement custom components for the canvas and the blocks. To implement both of these I basically had two options. One option would have been to use a custom component for the canvas view and override its draw method to then build the block views based on custom drawing objects. Option two was to use custom Swing components for the blocks instead.

While the argument could be made that it's more efficient to implement a lightweight way to draw custom, lightweight blocks on the canvas, that option would require a lot of work, since the blocks have a lot of requirements to fulfill. Instead, I also used Swing components for the block views. This enabled both nesting them inside other Swing components and also nesting Swing components inside the block views, which made the blocks flexible to use for different contexts.

The canvas view was quite simple to implement, it is just a JPanel without a layoutmanager. The layoutmanager is missing because the blocks need to be able to increase their size and set their position themselves.

The block view was more complicated. Swing is capable of finding the deepest component at a specific position and it is also able to check if components are intersecting. These are very important functionalities for a VPL and utilizing Swing's built in functionalities to fulfill them meant that I was able to keep the model way simpler.

In Swing user interfaces are built by nesting components, which results in a tree of components. It then utilizes the composite pattern to make sure that every subtree of components can be treated like a single one. The framework is optimized around this idea. It is very efficient at drawing and moving nested trees of components.

For the block-based language I also needed the capability to move trees of block views. By extending the JComponent class, block views gained the same advantage as long as attached blocks would be added as subcomponents. This way Swing can treat a tree of blocks just like a single block, which makes dragging a tree of blocks just as efficient as dragging a single one.

But block views also had to be able to correctly position attached block views. Therefore I extended the JPanel instead. JPanels are Swing components that use a layout manager to set the size and position of nested components. What the user will perceive as a block will internally actually only be a small part of what the block view actually consists of. The rest of the space is reserved for the attached blocks.

The problem with Swings layout managers was that they are designed with a top down approach in mind. Components have to stay within their bounds. Therefore parent components tell the child components to fit into a maximum dimension. The problem with this is that block-based languages work the other way around. The size of a block is determined by the sizes of the attached blocks. To achieve this behavior I had to write a custom layout manager. It determines the width of a block based on the necessary width to fit all attached blocks. It then positions both the visible block and the attached ones inside these dimensions. Because of the top down behavior of Swing I had to trigger the layouting process twice when attaching a block to another one. First the block that got a new one attached, needed to be laid out to determine its required dimensions. With this information set, the root block then had to be laid out in order to update the layout

of the whole tree. The last step then was to set the dimensions of the root block and trigger a repaint over the area covered by the tree.

There was actually another reason to use a custom layout manager. The blocks had to be able to visually interlock sockets and plugs. But Swing assumes components to be of rectangular shape and that those rectangles do not overlap. Drawing outside this rectangular bounding box would result in undefined behavior. But Swing allows components to announce that they are not fully opaque. Therefore my solution was to stay within the bounding box of the parent block but to use the transparency mechanism to draw the plug and the sockets. The custom layout manager then overlaps child components but stays within the bounding box of the parent. I then took control of the drawing in this overlapping area by overriding the responsible method. This way I was able to visually interlock sockets and plugs without compromising the assumptions of the Swing framework.

Since this made the visible area of blocks smaller than the bounding box, I also had to override the contains method to take the custom shape into account. This method is used by Swing to find the component that receives the events when the user presses the mouse.

4.5.2 Implementation of the Model Component

The model represents the state of the application. Although there are interpretations of the MVC pattern, where the model also is responsible for defining the behavior of the application, in this case that proved to add unnecessary complexity. Swing provides very useful functionalities for VPLs, but to use them I had to move some of the logic to the view. An example is that Swing is capable of detecting the deepest component at a given location. Another one is the capability of detecting if two components intersect with each other. To avoid reimplementation of these functionalities in the model, I handle logic related to which blocks interact with each other in the controller instead. This keeps the model quite simple. It holds information of which blocks exist and what information they need to store. The framework itself does not have a lot of block information. The application must extend the abstract models provided to add application specific blocks. It has to add a single instance of these blocks once to the block drawer. The framework then is able to create new Instances of them using the factory method pattern. This is why every block model is forced to provide a public method to create a new instance.

The block model holds references to parent and child blocks so that the application can work with this structure. It also determines which plug and sockets a block uses, to define the connectability with other blocks.

The canvas model holds references to blocks that are placed on the canvas directly. This list is sorted by the order in which they were changed. The most recently changed tree gets highlighted. This allows the application to only use the most recently changed tree. I introduced this behavior because the CGA shape grammar pipeline I use is only capable of rendering a single grammar at a time. This way the user can still work on multiple trees of blocks in the editor, while the application only renders the tree that is actively worked on.

4.5.3 Implementation of the Controller Component

The controller receives input events from the view and chooses how to respond to them. When an user input event is registered the controller uses Swing's capabilities to determine which views need to be part of the event. It then applies the necessary changes to these views and models. This means that the controller is in control of the life cycle of individual view and model instances.

Canvas and blocks require very different controller logic. Therefore I applied the separation of concern principle and introduced a separate controller for each of them.

The canvas controller handles user input events related to adding or moving blocks on the canvas. It checks if blocks should get added to the canvas or if they should get attached and then either handles them directly or delegates the task to the block controller if a block should get connected to another one.

The block controller on the other hand is only concerned with logic related to how blocks interact with each other and then performs the necessary actions to attach, detach, initialize or destruct block views and models accordingly.

4.5.4 Implementation of the User Interface

As stated before the user interface was realized using the Swing framework. I have already mentioned the architectural details of the block view component. Now I will go into the details of the applications user interface and its layout.

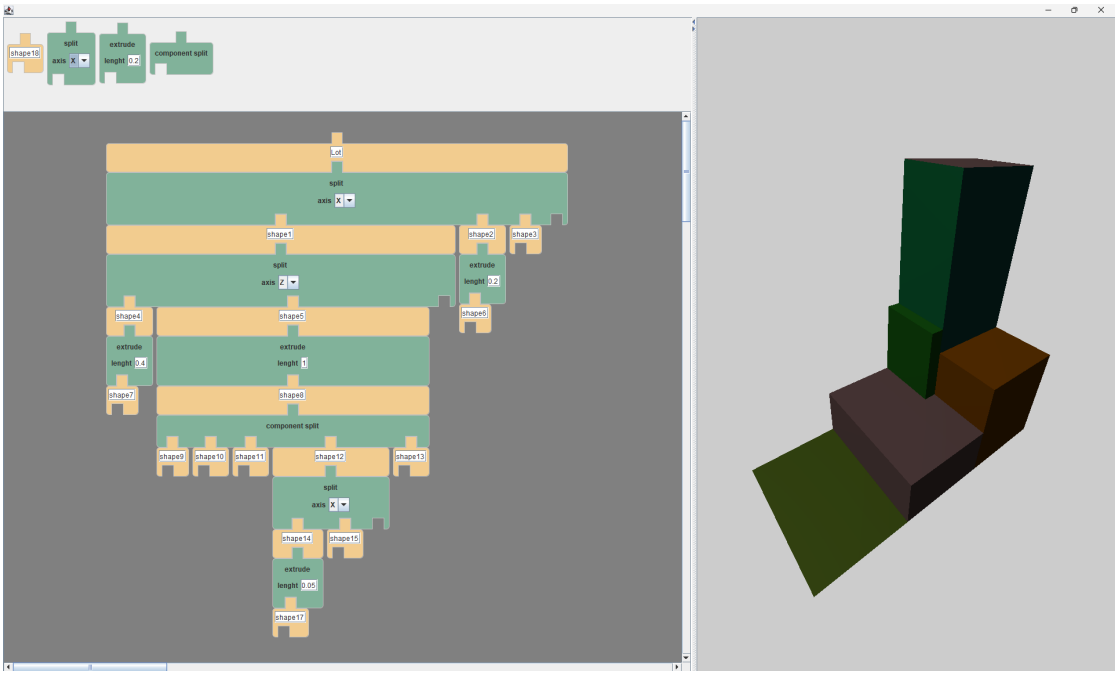

Figure 4.4: Screenshot of the final prototype showing an example program in the designed VPL.

Figure 4.4 shows that the editor is composed of three main components, canvas, block drawer and the 3D modeling preview. The canvas is the main workspace where the user arranges blocks. I utilized a scrolling mechanism for this component and gave it a size much larger than the available screen space on a 1080p display, allowing the user to create larger block trees than the available space permits. North of the canvas is the block drawer. This is the place where the user is able to draw new blocks from. Its size is dependent on the size of the blocks it houses, allowing for larger blocks in the future. East of both of these views is the modeling preview. To enable the user to balance the available space between the preview and the canvas, I made the border between them slidable. This helps mitigate the limitation of the available screen space in conjunction with the growing nature of the block widths. A very useful feature to further lessen this limitation would be to allow for zooming of the canvas. Unfortunately, Swing offers no built-in feature to resize all subcomponents of a view. It is possible to resize the drawing of an individual component, but this process was infeasible to get working with components like text fields and drop-down menus. The best compromise would properly be to base all the sizes of the block views on the text sizes of their contents. This way, the resizes could be done by incrementally resizing the text.

4.5.5 Connection to the CGA Shape Grammar

The grammar builder component is responsible for connecting the VPL to the existing CGA shape grammar framework. I hooked it into the block-based framework by utilizing the observer pattern the model already implements. The builder component observes the canvas model and starts the building process when the active tree has changed.

The grammar builder then initializes a new empty grammar to replace the existing one that currently gets rendered by the existing CGA shape pipeline. The CGA shape framework works in a similar way. It observes changes to the grammar and automatically triggers the pipeline to generate a new 3D model. The CGA shape grammar framework is capable of handling newly added grammar rules instantly, but I found no mechanism to remove rules. Instead of adding this functionality I decided that the builder handles VPL changes by replacing the previous grammar with a new one. This approach is easier to test and verify to work correctly.

The builder uses the following algorithm to obtain the grammar. It first initializes a fresh grammar instance and walks through the active block tree in a depth-first search, starting from the root block. It then only searches for action block models, since in the VPL action blocks directly relate to grammar rules. The builder calls a method of the action block model that returns the grammar rule. There is a chance that the model is not able to generate a rule. In that case the builder will mark this block model to be in an invalid state to signal the user that he must modify a parameter or add additional shape blocks. The builder skips invalid blocks and the entire subtree behind them, to maintain a valid grammar.

On completion the algorithm provides a valid grammar. The builder then replaces the existing grammar of the cgaShapeParams instance. The CGA shape framework will then further process it automatically to render a new 3D view.

5 Evaluation of the VPL

In this chapter I will first introduce the heuristic evaluation method and its set of heuristics, carry out an evaluation of the prototype and then summarize the results of the evaluation.

Evaluation is crucial to enable assessment of the functionality of the approach. It also allows for the validation of the design decisions whether or not they were able to successfully address the problem they were intended to solve. I decided to use an heuristic approach, because a heuristic provides a standardized set of criteria, making it easier to objectively assess and compare against other approaches.

Kölling und McKay proposed a domain specific heuristic evaluation method for evaluating novice programming tools like for example VPLs. They saw the need of improving on previous sets of heuristics, which lacked to provide some useful information since they were not specific to this domain [5].

Making use of this set of heuristics will ensure a comprehensive evaluation, capable of identifying all major issues.

5.1 Methodology

To develop the set of heuristics, Kölling und McKay balanced the goal of maximizing fault detection with the need of being practical and manageable in size.

Specifically they defined two criteria that each individual heuristic had to fulfill. The first one was that each heuristic had to be able to uniquely identify a set of actual known issues in existing systems of the target domain. The second one was that each heuristic had to avoid ambiguity in classification of identified faults.

They also defined two criteria that the set as a whole should fulfill. It had to be small enough in size to remain manageable and it also had to identify all major known issues found in the domain.

With these criteria in mind, they came up with thirteen evaluation heuristics, which I will introduce in the next section.

5.2 Definition of the Evaluation Heuristics

This section introduces the set of heuristics Kölling und McKay proposed for evaluating novice programming tools [5].

1. **Engagement:** The system should engage and motivate the intended audience of learners. It should stimulate learners' interest or sense of fun.

2. **Non-threatening:** The system should not appear threatening in its appearance or behavior. Users should feel safe in the knowledge that they can experiment without breaking the system, or losing data.

3. **Minimal language redundancy:** The programming language should minimize redundancy in its language constructs and libraries.

4. **Learner-appropriate abstractions:** The system should use abstractions that are at the appropriate level for the learner and task. Abstractions should be driven by pedagogy, not by the underlying machine.

5. **Consistency:** The model, language and interface presentation should be consistent both internally and with each other. Concepts used in the programming model should be represented in the system interface consistently.

6. **Visibility:** The user should always be aware of system status and progress. It should be simple to navigate to parts of the system displaying other relevant data, such as other parts of a program under development.

7. **Secondary notations:** The system should automatically provide secondary notations where this is helpful, and users should be allowed to add their own secondary notations where practical.

8. **Clarity:** The presentation should maintain simplicity and clarity, avoiding interface elements of the environment.
9. **Human-centric syntax:** The program notation should use human-centric syntax. Syntactic elements should be easily readable, avoiding terminology obscure to the target audience.
10. **Edit-order freedom:** The interface should allow the user freedom in the order they choose to work. Users should be able to leave tasks partially finished, and come back to them later.
11. **Minimal viscosity:** The system should minimize viscosity in program entry and manipulation. Making common changes to program text should be as easy as possible.
12. **Error-avoidance:** Preference should be given to preventing errors over reporting them. If the system can prevent, or work around an error, it should.
13. **Feedback:** The system should provide timely and constructive feedback. The feedback should indicate the source of a problem and offer solutions.

5.3 Carrying out the Evaluation

Engagement

I targeted an audience of computer science students that might not be familiar with the CGA shape grammar yet, but might have heard of its capabilities of generating architecture and structures. The prototype tries to engage with this audience by providing it with the fundamental tools of that grammar to enable them to rapidly build the mass model of a structure, hoping that it might spark the creative interest of creating or recreating an architecture.

Non-threatening

The system reduces the complexity of the CGA shape grammar by hiding the more advanced concepts like branching rules and preconditions. The blocks provide easy access to the parameters that the user can change. The prototype further reduces complexity by

limiting the functionality to the core actions of the grammar. This reduced complexity avoids intimidating the user. The problem with the prototype is that the user has no capability of saving the progress to avoid loss of data. Therefore the user is always threatened by the fact that his creations will get deleted when the application exits. The CGA shape grammar would have the capability to be saved to a text-based format, but the prototype is missing this functionality.

Minimal language redundancy

The actions of the CGA shape grammar have quite distinct functions, therefore conveying them into blocks resulted in equally distinct visual actions. The most similar actions are the split and the repeated-split actions. But there is a very important distinction between them. The user has to provide a fixed amount of splits to the split action, while the repeated-split action only uses a length parameter to determine the amount of splits depending on the available scope. This makes this action context aware and enables users to assume that the resulting scopes will roughly have the given split length afterwards. This is an important feature to place details like windows and doors. Therefore I assess that the VPL has a minimal language redundancy.

Learner-appropriate abstractions

The language is targeted towards novice users of the CGA shape grammar. This was the reason to avoid more technical abstractions like rule blocks, which resulted in a more beginner friendly visual language. For the prototype I implemented the most elemental actions only, the split and the extrude actions. These give novice users enough tools to build rectangle based mass models and even allows for adding some details like windows. This will definitely result in the users outgrowing the prototype, which is currently lacking more advanced actions of the CGA shape grammar. Adding new actions and abstractions in the future always needs careful consideration which abstraction will add benefit to novice users and which might overwhelm them, or if the language should also target more advanced use cases.

Consistency

An example for an inconsistent behavior would be the differences of the Java language regarding object types and primitive types. Those types require different operators to do similar tasks. The minimal redundancy of the developed VPL also leads to a good consistency. Since there are no similar abstractions there are also no inconsistent but similar methods of achieving a similar task.

Visibility

Visibility was an important factor when developing the prototype. The system tries to keep the user informed about the status of the system by highlighting blocks that it is not able to process. A better but little more sophisticated method would be to not only highlight the parameter field instead, but to also give hints why the value is invalid. In this respect the prototype is lacking important functionality that would greatly improve the users awareness of the systems status.

Secondary notations

The system provides every block with an unique id as its name when being placed. But the user is able to change this id into something that is more meaningful at communicating his intent. For example he could name an action block 'window', to note that this subtree of actions creates a window-like structure. These notations are important to enable users to navigate the arising tree structures when connecting blocks.

Clarity

The biggest threat to maintaining simplicity and clarity are the block trees. The decision to align the blocks in a 2 dimensional tree-like structure, influences the size of root blocks exponentially in regard to the depth of their tree. This was the reason to constrain the blocks to elemental parameters, since every parameter that would increase the size of a block has a largely greater impact on the root block. To enable users to construct more complex structures a mechanism is needed that reduces visual complexity and size of the tree structure.

Human-centric syntax

All block types feature a keyword that expresses their meaning towards the user. This is important to allow readability of what a block actually does. The CGA shape grammar has a lot of different actions and shapes, therefore I considered the usage of keywords favorable over the more compact option of icons. The action keywords of the grammar were already very descriptive so using them for the blocks was fine. The grammar keywords for shapes are also descriptive but they are very technical. For the prototype I simplified the amount of shapes and used the keywords 'surface' and 'body' instead of 'polygon shape' and 'prism shape', to use a more novice friendly terminology.

Edit-order freedom

In order to fulfill the requirements of the CGA shape grammar to add an unlimited amount of shapes to an action, it was important that the amount of block-connectors would grow and shrink depending on the amount of added shapes. Although there are mechanisms possible to insert a block in between two existing ones, this functionality was outside of the reasonable scope for the prototype. This is a problem for the edit-order freedom. To reorganize shapes, users need to unlink all of them first and reconnect them in the favored order. The affected action will forget the set parameters for these shapes in the process. This is an issue in regards to the edit-order freedom heuristic.

Minimal viscosity

Viscosity in the context of a block-based editor refers to the resistance of change when adding or modifying the existing structure. Apart from the issue mentioned regarding the edit-order freedom, the VPL improves this viscosity quite a bit over the text-based grammar. Changing the id of a shape block is a simple process that requires no further adjustments in other places, since the link to the id is provided by the connection hierarchy of the blocks instead. Adding another output shape to an action is as simple as adding another compatible shape block using an empty socket.

Error-avoidance

Error avoidance was one of the reasons to favor a block-based approach over a diagram-based one. The block-based VPL is able to avoid syntactic errors regarding the connection of blocks. Although the feature unfortunately did not make it into the prototype, the concept includes an approach to using the connector shapes to ensure this functionality. The shape of these connectors distinguishes between two dimensional and three dimensional shapes while also ensuring that shapes can only get connected to actions and vice versa.

Feedback

The feedback capabilities of the prototype are lacking in regards to the parameter input fields. They require careful consideration of whether values inside the input fields are of valid type and inside valid and meaningful bounds. For every input field there should be a feedback mechanism that not only highlights an invalid status, but also provides a meaningful hint to direct users towards the solution of the problem.

6 Conclusion

6.1 Summary of the achieved Goals

This thesis had the goal to develop a user-friendly VPL for the CGA shape grammar. From the results of the evaluation I conclude that this goal was achieved, although the developed prototype leaves room for future improvements which I will describe in the next section. The VPL is able to provide novice users with a tool to get to know the CGA shape grammar in an easy and fast manner. Utilizing a block-based approach meant that users could focus on their design ideas and learn about the capabilities of utilizing shape grammars for generalized visual modeling rather than modeling individual buildings. The VPL approach allows users to focus on the design rather than having to learn the syntax of the text-based grammar first. The design decision to use an action oriented mental model rather than the rule-based model made it easier for novice users to learn about the actions, but it limits the capabilities. My initial theory was that shape blocks could be augmented with the capabilities of the branching parameters like probability and condition. This theory proved wrong during the design processes of the VPL as it became clear that this would add a confusing characteristic to the shape blocks. In my opinion the best approach to add the full capabilities of the CGA shape grammar to a VPL would be the introduction of dedicated rule blocks. The current approach allows users to create a rule-based building that adapts to different lot sizes. A dedicated rule block would expand this idea further to also allow for variance even with a fixed lot size.

6.2 Lookout for possible further Development

The heuristic evaluation identified that the exponential growth of root blocks, related to the depth of the tree-like block structure is a huge concern. Solving this issue would be

of high priority for a possible further development of the VPL. A solution that reduces both visual clutter and the size could be the ability to collapse a subtree down to a single block. This could also be a first step towards letting users create their own reusable action blocks based on their needs.

Another issue identified by the evaluation was the edit-order freedom when connecting blocks. The missing feature negatively affects the user-experience. An important advancement for the prototype would be to add the feature to insert a block in between two existing ones.

Implementing the different connector types designed in the concept would dramatically improve the user experience. This would be an important addition to further develop the prototype. The prototype would also benefit from adding the identified missing feature of providing useful feedback to guide users towards entering valid values to input fields. This requires further development in order to get the prototype closer towards a useful product.

Another useful addition to the VPL would be to add dedicated rule blocks, instead of hiding the rule concept from the users. While this makes it more complicated for novice users, they will likely outgrow the capabilities of a visual language without these concepts. The addition of dedicated rule blocks would allow for mapping the probability and condition parameters. These would expand the VPL to a point where users would be able to learn the advanced concepts of the grammar.

Bibliography

[1] Chang, Shi-Kuo: Icon semantics—a formal approach to icon system design. In: *International Journal of Pattern Recognition and Artificial Intelligence* 1 (1987), Nr. 01, S. 103–120

[2] *Google Blocky*. https://developers.google.com/blockly. – Accessed: 2024-04-16

[3] Haakonsen, Sverre M. ; Rønnquist, Anders ; Labonnote, Nathalie: Fifty years of shape grammars: A systematic mapping of its application in engineering and architecture. In: *International Journal of Architectural Computing* 21 (2023), Nr. 1, S. 5–22

[4] Jiang, Tao ; Li, Ming ; Ravikumar, Bala ; Regan, Kenneth W.: Formal grammars and languages. In: *Algorithms and Theory of Computation Handbook, Volume 1*. Chapman and Hall/CRC, 2009, S. 549–574

[5] Kölling, Michael ; McKay, Fraser: Heuristic evaluation for novice programming systems. In: *ACM Transactions on Computing Education (TOCE)* 16 (2016), Nr. 3, S. 1–30

[6] Krispel, Ulrich ; Schinko, Christoph ; Ullrich, Torsten: The Rules Behind–Tutorial on Generative Modeling. In: *Proceedings of Symposium on Geometry Processing/Graduate School* Bd. 12, 2014, S. 1–2

[7] Kuhail, Mohammad A. ; Farooq, Shahbano ; Hammad, Rawad ; Bahja, Mohammed: Characterizing visual programming approaches for end-user developers: A systematic review. In: *IEEE Access* 9 (2021), S. 14181–14202

[8] Lieberman, Henry ; Paternò, Fabio ; Klann, Markus ; Wulf, Volker: *End-User Development: An Emerging Paradigm*. S. 1–8. In: Lieberman, Henry (Hrsg.) ; Paternò, Fabio (Hrsg.) ; Wulf, Volker (Hrsg.): *End User Development*. Dordrecht :

Springer Netherlands, 2006. – URL https://doi.org/10.1007/1-4020-5386-X_1. – ISBN 978-1-4020-5386-3

[9] Lindenmayer, Aristid: Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. In: *Journal of theoretical biology* 18 (1968), Nr. 3, S. 280–299

[10] Lipp, Markus ; Wonka, Peter ; Wimmer, Michael: Interactive visual editing of grammars for procedural architecture. In: *ACM SIGGRAPH 2008 papers*. 2008, S. 1–10

[11] Lowdermilk, Travis: *User-centered design: a developer's guide to building user-friendly applications*. " O'Reilly Media, Inc.", 2013

[12] Maloney, John ; Resnick, Mitchel ; Rusk, Natalie ; Silverman, Brian ; Eastmond, Evelyn: The scratch programming language and environment. In: *ACM Transactions on Computing Education (TOCE)* 10 (2010), Nr. 4, S. 1–15

[13] Müller, Pascal ; Wonka, Peter ; Haegler, Simon ; Ulmer, Andreas ; Van Gool, Luc: Procedural modeling of buildings. In: *ACM SIGGRAPH 2006 Papers*. 2006, S. 614–623

[14] Parish, Yoav I. ; Müller, Pascal: Procedural modeling of cities. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001, S. 301–308

[15] Patow, Gustavo: User-Friendly Graph Editing for Procedural Modeling of Buildings. In: *IEEE Computer Graphics and Applications* 32 (2012), Nr. 2, S. 66–75

[16] *Popularity of Rhino Grasshopper*. https://aecmag.com/news/rhino-grasshopper/. – Accessed: 2024-05-13

[17] *Procedural Content Generation in Unreal Engine 5*. https://www.youtube.com/watch?v=aoCGLW53fZg. – Accessed: 2024-02-13

[18] Prusinkiewicz, Przemyslaw ; Hammel, Mark ; Hanan, Jim ; Mech, Radomir: L-systems: from the theory to visual models of plants. In: *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences* Bd. 3 Citeseer (Veranst.), 1996, S. 1–32

[19] Prusinkiewicz, Przemyslaw ; Lindenmayer, Aristid: *The algorithmic beauty of plants*. 1990

[20] Stiny, George ; Gips, James: Shape grammars and the generative specification of painting and sculpture. In: *IFIP congress (2)* Bd. 2, 1971, S. 125–135

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

A Appendix

A.1 Source Code

The source code to the thesis is available on CD and can be requested from the primary supervisor.

Der Anhang zur Arbeit befindet sich auf CD und kann beim Erstgutachter eingesehen werden.

Erklärung zur selbständigen Bearbeitung

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

Ort	Datum	Unterschrift im Original