KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Example-based Procedural Generation: Shape Inference and Grammar Induction from Voxel Structures

Gillis Hermans

Thesis submitted for the degree of Master of Science in Engineering: Computer Science, option Artificial Intelligence

Thesis supervisor:
Prof. dr. L. De Raedt

Assessor:
Dr. G. Marra
Prof. dr. A. Simeone

Mentor:
Thomas Winters

Academic year 2019 – 2020

Preface

I would like to thank both my promoter prof. dr. Luc De Raedt and supervisor Thomas Winters for allowing me the freedom to explore and experiment for the first few months of this thesis. I am immensely grateful for the weekly conversations with Thomas that always steered me in the right direction and kept me motivated. Finally, I would like to thank both my family at home and my friends in the Eikstraat for their endless support and bringing me the needed comfort and distractions.

Gillis Hermans

Contents

Abstract

In order to reduce development time, designers are increasingly relying on procedural methods for automatic generation of content in video games and films. Creating a procedural rule set that is capable of describing and generating a specific style of content is a considerable task. It requires extensive knowledge both of procedural methods and of the exact specifications of the desired content. A potential solution is to algorithmically induce the procedural rules that describe the style of a number of provided examples. We propose a method that finds the style elements of voxel-based example buildings in an iterative process with limited user input. The relationships between these style elements form a grammar of procedural rules, used to create and automatically generate new structures in the same style as the examples. Our method is able to successfully generate new buildings with a grammar induced from simple example buildings. It is, however, limited in a number of ways and is not entirely suited for automatic generation because the grammar only considers the local relationships between the style elements. Finally, we propose an extension of our method, which will allow the reliable generation of advanced structures that adhere to both the local and global style of the provided examples. This serves as a stepping stone to the relatively unexplored field of example-based procedural generation methods.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations

Abbreviations

CC	Computational Creativity
CGA	Computer Generated Architecture
IPM	Inverse Procedural Modeling
LSTM	Long Short-Term Memory
MDP	Markov Decision Process
ML	Machine Learning
PCG	Procedural Content Generation
PCGML	Procedural Content Generation via Machine Learning

Chapter 1

Introduction

1.1 Problem Statement

The design and creation of content for video games is traditionally done by hand. The different types of content artifacts are extensive, ranging over three-dimensional models, textures, sound effects, story elements and even the rules of the game itself [57, 26]. There are a few important drawbacks that handcrafted content in video games suffers from [55, 44]. For one, the technical complexity and scope of games is growing. There is a need for larger amounts of more complex pieces of content. Consequently, the task of manually creating these artifacts is becoming more difficult and time-consuming over time. Handcrafted content must often be modified and discarded when changes in the game or game engine make the content incompatible. Designers spend a great deal of time performing tedious tasks during the creation process that provide no creative value by themselves. Fortunately a great deal of content has a repetitive nature. Most artifacts share their basic structure with other similar pieces of content. Models of trees exist out of a highly similar structure of basic elements (trunks, branches and leaves), as do buildings, textures, quests and most other classes of content. This repetitive structure of artifacts allows us to define a set of rules that describes a specific piece of content, the style of that piece of content, or an entire content class. Procedural modeling [4, 53, 49] is the creation of, specifically, three-dimensional models and textures from a set of procedural rules. Instead of designing every model and texture from scratch the rule set can be followed to accelerate the design process. Additionally the rule sets are used to automatically generate new artifacts. This automated process lies in the field of procedural content generation (PCG) which is defined as *"the algorithmic creation of game content with limited or indirect user input"* [76] or, in other words, *"the (semi-)autonomous generation of game content by a computer"* [84]. PCG algorithms are not limited to three-dimensional models and textures but have been also used to generate, among others, quests [35, 5], world histories [3] and even entire games [11, 15, 77]. A significant advantage of procedural modeling and procedural content generation is the high degree of *database amplification* [61]. With only a few rules and parameters a large generative space can be created that allows for the generation

of many details. Instead of storing this entire space in a database, it can be recreated with only those few parameters and procedural rules. This does entail that small changes in the parameters can cause very large, and possibly unexpected, changes in the generated model.

Creation or induction of procedural rules Both procedural modeling and PCG suffer from the issue that the procedural rules and the PCG system must be created to generate a specific type or style of content. This is, in many cases, an incredibly complex process that requires extensive knowledge of the content specifications and of procedural modeling or procedural generation techniques [66, 40]. It is necessary to accurately specify the basic components and their possible compositions of the desired content. An important consideration is the strictness of the procedural rules and the space of allowed variations. It is possible to build an overly specific rule set or PCG system that is only capable of creating one artifact. On the other hand an over-generalized set of rules, or PCG system, would model and generate a set of artifacts that is too broad for the intended artifact style. Describing a specific type or style of content requires striking a balance between overly specific and general.

Instead of manually creating procedural rules that are capable of creating new artifacts in a certain style, it is possible to induce the rules from example artifacts in the desired style. PCG algorithms that utilize machine learning techniques (PCGML) are able to *"generate game content by models that have been trained on existing game content"* [72]. Of course PCGML has its own set of challenges. Games often lack the amount of data required for many state of the art machine learning techniques that are highly successful in other domains. A second issue is that many ML models are difficult to interpret [20], where interpretability for ML is defined as *"the ability to explain or to present in understandable terms to a human"*. In many domains an incomprehensible *black box* approach does not pose a huge problem. But when generating content for video games, it is necessary to have a large amount of control over the generative space. Many constraints must be met to follow the designers' wishes and, more importantly, to avoid causing bugs and breaking the game. Many procedural systems will be modified numerous times to change the output of the system. Trying to tweak a system that is not easily predicted, or even understood, will be an unnecessarily lengthy and strenuous process. Inverse procedural modeling (IPM) [40, 8, 66] considers the opposite problem of procedural modeling for three-dimensional models and textures. Instead of creating new models from a set of procedural rules, it seeks to find procedural rules and parameter values for existing example artifacts. Texture [81, 21] and model synthesis [42, 43, 44] algorithms propose creating new textures or models directly from one or multiple input texture or model examples. These methods are not limited by the issues of PCGML. However, while the solutions for textures and two-dimensional models are abundant [66, 40, 21, 74, 83, 28], the research on finding procedural rules and creating new artifacts from three-dimensional models is limited [44, 8].

Proposed method We propose a method for inferring the style of one or more three-dimensional voxel-based building examples with procedural rules in the form of a grammar. We describe a method for finding the basic style features present in the examples with an iterative process. These components can be used in various procedural rule specifications for creation and generation of new buildings. We discuss the *additive shape grammar*, a simple grammar that consists of a single rule type. This simple grammar provides limited capabilities for automatically generating new buildings in the same style. We discuss an extension of our method, inspired by more advanced grammar specifications, that is more suited for automatic generation, which is expected to produce better and more reliable resulting artifacts. We want to use a model that avoids data related issues, by not necessitating enormous amounts of data, and can be interpreted or explained. Our approach is capable of finding the style of a single, or multiple, examples. A rule that is not present in an example will not be able to be induced. Thus inferring a style from multiple examples may result in a better description of the style, if the examples consist of different style elements and relationships. There is, however, no need for enormous amounts of data, as a single example is all that is necessary. Our grammars are interpretable and modifiable because of the explicit rules and style features.

1.2 Research Questions

- Is it possible for an algorithm to infer the basic style features of one or more example voxel-based buildings?
- Can these features be used to induce an interpretable procedural rule set that captures the style of the limited examples and can be used to create and generate new buildings in the same style?

1.3 Relevance

1.3.1 Usage

The induction of procedural rules from example buildings that allow the generation of new artifacts is relevant in a number of fields. Outside of video games these techniques can be used to populate virtual environments in other computer graphics related fields such as architectural modeling and the film industry. Visual effects in film have extensively used procedural methods to populate certain scenes with generated artifacts [53, 30]. Synthesising enormous crowds of people from smaller crowd examples is another typical case of example-based generation used in films. Besides the entertainment industry, procedurally generating indefinitely large cities is a goal pursued for urban planning, traffic, driving and flying simulations [82, 49]. Cities exist out of different neighborhoods with many different types of buildings. The induction of a generative system from just a few examples could alleviate the strenuous task of manually creating rule sets for each style of building present in the cities.

1.3.2 Style Inference

Togelius et al. [75] address the challenge of building a PCG system that can create content in a style that has been somehow learned or inferred among seven other critical challenges in the field of PCG. A number of previous works have attempted to tackle this problem. Snodgrass and Ontañón [62], Dahlskog et al. [18] and Dahlskog and Togelius [16, 17] have built systems to imitate the style of 2D *Super Mario Bros.* [52] levels in newly generated levels. The techniques they used were respectively Markov chains, n-grams and evolutionary algorithms. Dahlskog et al. and Dahlskog and Togelius explicitly state style imitation as their goal, while Snodgrass and Ontañón learn patterns from existing levels in order to generate new levels with Markov chains. Either way, these studies were all relatively successful in imitating the style of the existing *Super Mario Bros.* levels. PCGML approaches [84] are, in essence, undertaking this challenge as well. By training a generator on existing content it will, deliberately or not, produce new content that imitates the style of the training data. Style imitation in video games has, as far as we know, not been addressed in more complex cases such as three-dimensional structures. The fields of inverse procedural modeling and model synthesis in computer graphics do contain limited research on imitating or inferring style from more complex three-dimensional structures [42, 43, 8]. We consider our method a step in the right direction of this challenge.

1.3.3 Computational Creativity

While humans have the inherent ability of keeping a piece of content structurally and creatively consistent while adding unique details, artifacts generated by procedural methods often lack both this consistency and detail [75, 57]. It does not matter if a PCG system can generate a near infinite supply of unique artifacts, when they are not perceived as unique by the players. Two nearly identical mazes are considered unique by the generator but to a player this variation is insignificant. We should strive to make PCG systems that create content that can be perceived as unique and can hold the interest of a player. The existence of an actual creative algorithm, that produces consistent creative content, could drastically change the discipline of PCG.

Computational creativity (CC) or *"the philosophy, science and engineering of computational systems which, by taking on particular responsibilities, exhibit behaviours that unbiased observers would deem creative"* [14] is a field that should be taken into account in PCG research. Furthermore, video games are an excellent venue for exploring CC, because of their multifaceted nature that combines many different disciplines of art [36]. Many people doubt computational creativity could even ever be achieved because they hold an inherent bias that machines can not be creative [47]. A system that can take an actual creative leap, that has not been programmed or encouraged by its designer, is considered highly unlikely by many. We leave further discussion on the feasibility and evaluation of computational creativity up to other works. Nevertheless, computation creativity is relevant for this thesis in the following way. An initial step in many human creative processes is the discovery of other work,

or elements thereof. A human will learn, and be inspired by techniques and creative concepts used by others. These ideas can then be explored and combined to produce new creative work [7]. An assumption is made that in order to generate creative artifacts certain aspects of human creativity must be simulated [13]. A computer system with the ability to learn new styles, concepts or techniques could be able to apply and combine these learned concepts on newly generated artifacts. Ordinary imitation is certainly not the goal of computational creativity research [79] and, to be clear, we do not claim any of the methods discussed in this thesis exhibit any form of creativity. Nonetheless, the ability of an algorithm to infer and imitate the style of example artifacts could be considered a step in the direction of computational creativity, just as data mining and machine learning methods could [79]. A system that can learn different concepts from many different, already existing buildings, and combine these into new structures seems feasible. If the resulting generated buildings could actually be considered creative is, of course, still up to discussion.

1.4 Overview

Chapter 2 reviews the relevant background information and related work. This chapter sketches the state of procedural content generation and procedural modeling and the methods that served as a direct or indirect inspiration for this thesis. Additionally related style inference and inverse procedural modeling research is discussed in more detail. Chapter 3 explores our method for finding the basic style components of an example building. In chapter 4 we define a simple grammar specification and discuss its results and limitations in automatic generation. We additionally propose an extension of our method that is more suited to automatic generation. Finally, chapter 5 concludes the thesis by summarizing our results and insights and discussing potential further work.

Chapter 2

Background and Related Work

This chapter explores procedural modeling and procedural content generation and touches on a number of relevant methods. We discuss the concepts that served as an inspiration for this thesis in depth. We further address methods that attempt to generate new artifacts from provided examples, such as texture and model synthesis and inverse procedural modeling, which we will compare to our own methods.

2.1 Procedural Modeling and Procedural Content Generation: Introduction

Procedural modeling is the creation of three-dimensional models or textures from a set of procedural rules or with a procedure or program [4, 53, 49]. This allows for accelerated creation and use of models and textures for video games, architectural models, films and other applications. A set of procedural rules and parameters are powerful in the sense that with only a few rules and parameters a huge number of different models can be created. This is an important advantage that leads to a property known as *database amplification* [61], where a database of a certain type of content need not be stored but can be generated from the small set of procedural rules instead.

Togelius et al. [57] define procedural content generation (PCG) as: *"the algorithmic creation of game content with limited or indirect user input"*. Game content refers to an extensive list of different types of content used in video games. This includes, but is not limited to, textures, character and object models, music and sound effects, game levels, quests or missions, story and the game mechanics or rules [31]. PCG is an autonomous or at least semi-autonomous process. Even though some PCG techniques allow a certain degree of user input, the highly human-directed algorithmic creation of content should rather be considered a tool that is used to aid the design process. Procedural generation algorithms have been used to generate most types of content in many different games. Even entire, albeit small and simple, games have been procedurally generated [11, 15, 77]. The *LUDI* system by Browne [11] has even generated a board game called *Yavalath* that has been physically produced and sold quite successfully. Hendrikx et al. [26] provide a comprehensive taxonomy of types

of game content and their respective PCG approaches. Procedural modeling, when used to automatically generate new textures or models, can be considered a subfield of procedural content generation. We will use the terms procedural generation and procedural methods to encompass both procedural modeling and PCG.

Because the scope of video games is constantly growing, the concept of algorithmically generating content is becoming an increasingly appealing solution to save time and money during development [57, 84]. Apart from saving time and money on a project PCG has a few other benefits. For one it can drastically decrease the file size of a video game. Only the algorithm must be stored on file while the content can be generated *on the fly*. This used to be a more important perk of PCG, as memory and disk space were significantly smaller. In recent years this is no longer a major concern for video game designers, but it remains a benefit nonetheless. Another advantage is the fact that PCG can substantially increase replayability. Many games use PCG as a tool to keep every single playthrough fresh and interesting. An ever changing, unpredictable game world has the ability of holding a player's interest for a longer time.

The various goals of procedural content generation techniques lead to different priorities for the generative systems [57]. A system that is designed to generate new content *on the fly*, during gameplay, should have much stricter time constraints than a generator that is used by a designer during development. A number of other priorities, aside from the speed, are important to consider. These are reliability, or the ability to guarantee a certain quality of content, controllability for user or algorithm input, expressivity and diversity of the content and finally the creativity and believability of the content. Depending on the goals of the generator, trade-offs will need to be made between these properties.

2.1.1 History of Procedural Content Generation

PCG was used in video games for the first time in the early '80s. In order to make every playthrough of *Rogue* [80] unique and exciting, the idea was conceived to algorithmically generate dungeons and the content inside them. *Rogue* birthed a whole genre of video games heavily reliant on PCG, called *roguelikes*. In *Elite* [10] PCG was used to work with the memory limitations of that time. With a meagre memory capacity of 22kB *Elite* planned on providing 2^{48} galaxies which each contained 256 different planets. The publishers however chose to release the game with only 8 of these generated galaxies. In recent years PCG is used in a broad range of commercial and experimental games for many different types of content. *Borderlands* [63] and *Diablo* [23] use PCG to generate a near infinite supply of weapons. The procedural systems in *Dwarf Fortress* [3] generate an entire planet and its history for every playthrough. These systems are intertwined where the history of the planet leaves its mark on the world with ruins of ancient cities or monuments honouring legendary figures. The story and conversation with the main characters of *Façade* [41] is procedurally generated in response to the players words and actions. PCG has been used successfully in many different applications, and the prospects for future games remain exciting.

2.2 Classic Procedural Methods

There are many different approaches to procedural modeling and procedural content generation. When used in games, procedural modeling can be considered a subfield of PCG for three-dimensional models and textures, whereas when used in other domains, it cannot. We discuss the combination of both fields in this section, under the overarching term *procedural methods*. *Procedural Content Generation in Games* by Togelius et al. [57], the fourth chapter of *Artificial Intelligence and Games* [84] by Yannakakis and Togelius and the work of Hendrikx et al. [26] contain a summary of basic techniques used in PCG. These include vastly different approaches for various types of content such as fractal and noise-based methods commonly used for textures and terrains, solver-based approaches that use constraint solvers to search for content that satisfies the given constraints, grammar and rewriting systems used for textual, natural and geometric content, and search-based methods that search until a sufficiently good artifact is found in the search space. We touch on the relevant basic methods in these summaries and more complex and specific approaches in this section.

2.2.1 Grammar-based Procedural Methods

Grammars are inherently suited for procedural modeling and procedural content generation. The iterative expansion of a rule set on strings, or other elements, allows the simple generation of artifacts that follow the rule set. Aside from formal grammars, similar formalisms such as L-systems and shape grammars are well suited for the generation of different types of content.

Formal Grammars

A formal grammar, as defined by Chomsky [12], is the 4-tuple $\langle N, \Sigma, P, S \rangle$ with:

- N a finite set of non-terminal symbols
- Σ a finite set of terminal symbols
- P a set of production rules of the form $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
- S a start symbol

Outside of using formal grammars for descriptive and analytical purposes they are capable of generating new instances by expanding the production rules. Beginning from the start symbol every iteration step, a string with at least one non-terminal symbol is rewritten to a new symbol or set of symbols by a production rule. A formal grammar is therefore also referred to as a string rewriting system.

Grammars have been used to procedurally generate language-based and other content. Hall et al. [25] use a context free grammar, a grammar that only considers recursive rules, to generate fables for simulated religions. Aside from the generation of natural language, other types of content can also be generated through formal

grammars. Sportelli et al. [65] use probabilistic context free grammars to generate sequential levels for an infinite running game. A derivation of the grammar requires some additional constraints to control the output. Because of the simplicity of the grammar, they succeeded in introducing these constraints in the probabilistic grammar without any additional mechanisms. For more complex structures and grammars, managing these constraints will become increasingly difficult. This is a significant issue for most grammar-based generative systems. It is not trivial to add additional constraints into the grammar itself, so that automatic derivation of the grammar produces a specific set of artifacts. Alternatively the constraints can be checked during or after the automatic derivation. Either way, the more complex the grammar and the additional constraints on the output, the more difficult it is to ensure good results.

Formal grammars are limited in their use in procedural generation because of the sequential nature of the production rules and the fact that all basic components need to be reduced to a single symbol.

L-Systems

An L-system is a type of string rewriting system, similar to a formal grammar, originally designed by Lindenmayer [37] to model the organic growth of plants. An L-system is a 4-tuple: $\langle V, S, \omega, P \rangle$ where:

- V is a set of symbols containing elements that can be replaced, referred to as variables
- S is a set of symbols containing elements that remain fixed, referred to as constants
- ω the initial axiom or word existing out of symbols from V
- P is a set of production rules defining the way variables can be replaced with combinations of constants and other variables. A production consists of two strings - the predecessor and the successor.

In contrast to formal grammars the production rules P of L-systems are applied in parallel. A production rule is applied to all symbols in the word at every iteration. L-systems have proven a very successful graphical procedural modeling method for many different types of plants and geometric shapes such as fractals [54, 61, 33]. Parish and Müller [53] have extended L-systems to generate street networks of a city. Streets usually end when they connect with another street or loop back onto themselves while plant-like structures usually end in dead ends. Therefore, instead of a tree-like structure the topology of the street maps is more of a net structure. This modeling task requires a large amount of complex production rules. Whenever a new constraint is added many of these rules must be revised. Thus they relegate the setting and modification of the parameters to external functions so that all constraints can be taken into account at the end of production. Parish and Müller generate simple three-dimensional buildings with L-systems as well. The actual

three-dimensional geometry of the buildings is produced by transforming a bounding box of the buildings area with the rules output by the L-system while its texture is simply assigned to the building.

The parallel rule application of L-systems is suited for, but limited to, the generation of content with *growth*, such as plants, fractals and road networks.

Shape Grammars

The shape grammar formalism is a grammar of two- and three-dimensional spatial designs [67]. A shape grammar is a 4-tuple $\langle S, L, R, I \rangle$ where:

- S is a finite set of shapes
- L a finite set of symbols
- R a finite set of *shape rules* of the form $\alpha \rightarrow \beta$ where α and β are labelled shapes (S, L)
- I the initial labelled shape of the form (S, L)

A shape rule consists of a transformation τ from one labelled shape to another. A shape grammar is able to generate new geometric shapes through these shape rules. Shape grammars are similar to formal grammars and L-systems but labelled shapes contain additional data, in the form of symbols, such as their positions and orientations. A production is not completely predefined because the rules take these labelled values into account. Productions can be applied either sequentially or in parallel. Shape grammars have been used as a descriptive model for various architectural styles such as the Palladian grammar [69] and the Prairie House grammar [34]. A shape grammar can also be used for procedural generation where the shapes and shape rules, that describe a specific style, are used to generate new structures in that same style. A classic shape grammar is not suited for automatic grammar derivation. Many additional constraints cannot, or are very difficult to, be defined in the shape grammar itself. Consequently, the derivations are usually done by hand or with the assistance of a computer. Merrick et al. [45] apply shape grammars in a PCG application. They discuss how to guarantee that the output of the grammar follows the designers' wishes in the form of constraints. This is, as already discussed, a critical issue for shape grammars, and other grammar-based procedural generation methods. Aside from introducing the constraints inside the grammar it is possible to search through the output and only select derived artifacts that satisfy the constraints. They propose a hybrid approach where some knowledge is incorporated in the shape grammar, but the final output is selected based on some additional constraints.

Split grammars A significant problem with using shape grammars in procedural generation is the fact that automatic rule selection is not guaranteed to produce good new structures. Inspired by shape grammars Wonka et al. [82] define the split grammar formalism to be able to effectively use automatic rule selection. The

split grammar extends the set grammar formalism, which is a simplified shape grammar defined by Stiny [68] for computer implementations. A split grammar is a set grammar over a vocabulary of basic shapes (cuboids, cylinders, rectangles, etc.) with two types of rules. A split rule splits a basic shape into multiple other basic shapes. These new shapes must fit exactly into the volume of the previous basic shape. A conversion rule transforms a basic shape into another basic shape that must fit into the volume of the previous shape. These rules allow for a greater deal of control over the automatic rule selection process. The final set of basic shapes needs to be filled in with attributes, such as the textures or materials of the shape, that must be kept consistent over parts of the structure. A floor of a building, for example, will be often built out of the same materials or have the same window designs as the rest of that floor. In addition to the split grammar, that generates the basic shape structure, a control grammar distributes attributes throughout the structure in order to keep them consistent. The split grammar follows a strict hierarchy of rules. This simplifies the process of automatically deriving the grammar such that it can be controlled quite easily.

CGA grammars Müller et al. [49] introduce Computer Generated Architecture (CGA) grammars as an extension of and combination of split grammars [82] and the generation of urban environments with simple mass models [53]. A CGA grammar is a split grammar with a number of additional rules such as the combination of shapes. These additional rules, and the addition of new basic shapes such as roofs, invalidate the strict hierarchy guarantee of the split grammars. Thus, a priority is assigned to each shape rule according to the detail represented in the rule. This guarantees that the grammar is derived in a controlled hierarchical manner where low detail rules are derived before higher detail rules. The CGA grammar is highly successful at generating large city models with detailed buildings efficiently.

Open shape grammars Emilien et al. [22] define Open Shape Grammars as an extension of CGA grammars where the application of a shape rule can be undone if an external constraint is not met. In this particular case rules are undone when the placement of facade elements such as doors and windows collide with the terrain. A shape rule, such as placement of a door, is attempted a number of times until a placement is found that does not break any external constraints. If no placement is found, the rule execution is eventually abandoned.

Conclusion

Grammar-based procedural methods are capable of reliably generating various types of content, because of their inherent generative nature. Different grammar specifications are suited for different types of content: formal grammars for simple and sequential content, L-systems for content that shows *growth* and shape grammars for complex geometrical content.

2.3 Example-based Procedural Methods

A key challenge for procedural modeling and generation techniques is the definition of the procedural rules [66, 60]. Finding rules that accurately and correctly describe a generative space is a difficult task. It requires a lot of knowledge of both the procedural model and the content to be generated. The fact that with only a few rules and parameters a huge number of different models can be created is both the biggest strength and weakness of procedural methods. Just a single, seemingly insignificant, change in the rules or parameters can have an enormous effect on the resulting models or textures. Consequently these techniques are often difficult to predict and control [6, 73]. Designing procedural rules to create a certain type and style of content is thus often an iterative process of the following steps. After defining an initial set of rules, the designer executes the procedural system and examines the results. Next, the designer changes certain rules and parameters, that he expects to better describe the desired content. In order for the designer to know what rules to change, the procedural method must be interpretable to a certain degree. This is a far from ideal process that leads to one of the primary challenges in PCG [75]: the inference and imitation of style from example artifacts. A PCG system must be designed to generate exactly what the designer wants from the generator. This is an exceptionally difficult process that requires a lot of expertise. A PCG system that can somehow imitate the style of one or multiple example artifacts by inferring or learning that style could alleviate these issues.

The following example-based methods consider the challenge of generating new artifacts from example content. Machine learning PCG methods are trained on example content in order to generate new content in the same style. Texture and model synthesis methods automatically create large textures or models that resemble small input examples. Inverse procedural modeling methods induce a rule set from input model examples. This rule set can be used to generate new artifacts similar to the examples.

2.3.1 Machine Learning Methods

Summerville et al. [72] summarize procedural content generation methods via the machine learning domain (PCGML). Generators are trained on existing content in order to generate new content in the same style [84]. We discuss a number of methods here that are used to explicitly, or implicitly, infer the style of a type of content and generate new pieces of content in that style.

Markov Models and Markov Chains

Markov chains [39] are used to model a number of different states and the probabilistic transitions between them. A Markov chain is defined as:

- a set of states $S = \{s_1, s_2, ..., s_3\}$
- the conditional probability distribution $P(S_t|S_{t-1})$

This distribution represents the probability of a transition occurring from S_{t-1} to S_t when S_{t-1} is the present state. The Markov property states that this distribution of the future states is only dependent on the present state. This is also referred to as memorylessness. A Markov Model is a stochastic model where the Markov property is assumed. A Markov chain is in fact a simple type of Markov Model.

Snodgrass and Ontañón [62] use Markov chains to generate *Super Mario Bros.* [52] levels. They use higher order Markov chains that instead of taking into account just the present state, also consider a number of k previous states. The states are, in this case, the building blocks used in the map. The probability distribution is found by training the model on existing *Super Mario Bros.* levels and thus the generated levels will follow the style of these levels. The use of a Markov chain requires the assumption that the future state is only dependent on the present, or k previous states. The next building block in the map will depend only on a number of other blocks around it. The final resulting maps are playable after a cleaning step, which could be promising for the possibility of generation through Markov chains. Consistency over the whole level can not be guaranteed with Markov chains because of the Markov property. While the new examples will locally match the style of the example levels, the higher-order structure of the level does not necessarily.

N-Gram Model

The n-gram model is a Markov model that uses n-grams to predict the next item or word in a sequence. N-grams are sequences of n items or words. Every word is assumed to be independent from every word except the last n. N-grams are typically used in natural language processing. They have also been used to generate music [51] by predicting what note should come after the last n notes. The simplicity of n-grams comes with the same major drawback as other models that assume the Markov property. As decisions are made while only considering a small number of current items there will be no greater structure in the generated artifacts. Aside from notes correctly following other notes there is no notion of recurring themes, unless accidentally generated by the n-gram. One again guaranteeing stylistic coherence over the whole artifact is thus not possible with only n-grams.

Dahlskog et al. [18] use n-grams for generating two-dimensional *Super Mario Brothers* levels. They attempt to copy a style from a corpus of existing levels. Levels are split into vertical slices to create an alphabet of slices. The model is trained on the levels to predict the next vertical slice considering the last n slices. Because these slices are much larger components of the levels than the separate building blocks considered in [62], this model generates levels that are quite consistent and clearly in the same style as the training corpus. The choice of n defines how many states, or slices, are taken into account for the next state, and thus how locally or globally similar the generated artifacts will be to the examples.

Neural Networks

Various neural network architectures have proven useful for many ML issues, including generative systems. The neural approach has, for example, achieved great success in the generation of images through generative adversarial networks [24] and variational autoencoders [32]. A number of works have attempted to apply the neural approach to PCG.

Jain et al. [29] use autoencoders [32] to generate new *Super Mario Bros.* [52] levels. The autoencoders are trained on existing *Super Mario Bros.* levels. These networks are then used to discriminate generated levels from original levels. The network models can generate any number of maps, but there is no certainty that these will be playable, because of, for example, an impossible jump in the level. This approach could work for generating two-dimensional content that does not need to satisfy many (or preferably any) gameplay constraints.

To deal with the common problem that only the local style or local structural coherence is considered, for example because of the memorylessness of Markov models, Summerville and Mateas [71] generate *Super Mario Bros.* levels with *Long Short-term Memory* (LSTM) recurrent neural networks. LSTMs [27] are the state of the art in sequence learning approaches and are thus ideal for generating the sequential structure of 2D platformer levels. These networks have *memory mechanisms* that allow the network to remember or forget and can thus incorporate information about the entire generated artifact to generate the next states of the levels. Summerville and Mateas add simulated player path information to the input of the LSTM, which allows the generator to produce levels that adhere to good geometry and pathing.

2.3.2 Texture Synthesis

Texture synthesis [81] methods produce arbitrarily large textures that are visually similar to a small input example texture without any unnatural artifacts or repetition. This simplifies the texture creation process immensely. We discuss a basic texture synthesis method as an introduction to model synthesis. For further discussion see, for example, the survey of texture synthesis methods by Wei et al. [81].

Efros and Leung [21] tackle texture synthesis by preserving as much local structure as possible. Any pixel on the new texture must match a pixel within a square neighborhood of width w in the example texture. Textures are modeled as a Markov Random Field where each pixel is characterized by a set of neighboring pixels. New textures are generated by starting from a single pixel and growing the texture by finding neighborhoods in the sample texture that are similar and placing a new pixel from that neighborhood in the example.

2.3.3 Model Synthesis

Inspired by texture synthesis methods, Merrel [42] proposed model synthesis as a generalization of texture synthesis for higher dimension models. The method uses simple example shapes as input to automatically generate larger and more complex models that resemble the input in terms of its shape and local features.

Model synthesis works in both two and three dimensions. Example models exist out of a number of base building blocks connected to each other. The technique uses a three-dimensional lattice of vertices of building blocks connected by edges. For every vertex the possible building blocks are defined by its neighborhood of vertices and the rules between the building blocks. These rules define whether a certain building block is allowed above, below or beside another building block. A synthesised model consistent with these local rules can still contain conflicts, such as unwarranted overlapping components. A search for conflicts eliminates the simple ones, but is unable to remove all conflicts. This can cause issues in the consistency and quality of the generated models. The technique works reasonably well for two and three-dimensional models and can be extended to find symmetric, constrained and even higher-dimensional models. The building blocks are not learned from the example models, but must be predefined by the user.

Merrel [43] extended the model synthesis method with a continuous approach that takes any three-dimensional polyhedral structure as an example model. It is no longer necessary to provide the building blocks of the model. The method relies on an adjacency constraint, similar to the constraint defined in [21], that ensures that for every point x in the generated model a point x' exists in the example model whose neighborhood matches the neighborhood of x. This constraint guarantees a local similarity within neighborhoods that are defined by a radius ϵ. The proposed technique has its own limitations. Curved or highly tessellated models will perform inadequately because of increased time and memory requirements. It is also not suited for generating structures at different scales. Additionally, the adjacency constraint has the effect of an extended Markov property, limiting the possibilities in every step according to the current local state, ignoring the higher-order structural coherence. In a following paper [44] Merrel and Manocha extend the model synthesis algorithm once again to enforce satisfaction of numerous additional constraints besides the adjacency constraint defined in [43]. These additional constraints allow the users to control and increase the quality of the output of the model synthesis method.

2.3.4 Inverse Procedural Modeling

Inverse procedural modeling (IPM) considers finding a set of procedural rules and parameter values that describe existing graphical models [40]. This procedural rule set can take on different forms, such as L-systems and shape grammar which were discussed previously. These discovered rules can be used to procedurally generate new artifacts similar to the examples. IPM is the *inverse* of procedural modeling in the sense that instead of modeling or generating new pieces of content, given procedural rules, the goal is finding the rules, given the content. While some IPM techniques consider the rules known and attempt to discover the parameter values, others attempt to find the entire rule set. We discuss a number of IPM methods in both the two- and three-dimensional case.

Two-dimensional Inverse Procedural Modeling

Stava et al. [66] propose a technique to automatically generate L-systems from two-dimensional image examples. Repeated basic line segments are found and used as terminal symbols. Their composition is analyzed by calculating the transformation between all pairs of basic elements. These transformations are stored in clusters which are weighted and sorted according to user-defined criteria. These clusters are used to iteratively build a L-system using the most significant clusters, which represent the rules. This method finds a good description of the input image and can further be easily edited, by manipulating the L-system parameters and substituting the symbols for new symbols, to change the resulting generated images.

Teboul et al. [74] utilise shape grammars and reinforcement learning techniques to segment two-dimensional facade images into predefined style features such as windows and doors. They find a binary split grammar for a facade by optimizing a segmentation of the pixels for a merit function that associates style feature labels, such as *wall* or *window*, with pixels. These segmentations form rectangular shapes that form the basic components of the binary split grammar. This optimization is framed as a Markov decision process (MDP), an extension of previously discussed Markov chains, where in every time step the process is in a state s_t. An action a_t is taken on the basis of the state s_t. This action changes the state s_{t+1} and provides a reward r_{t+1}. Rewards have the Markov property and depend only on the current state and action. The goal is to maximise the final reward, and as such to maximise the MDP which be efficiently solved with reinforcement learning. The merit functions can be learned from either supervised or unsupervised data.

Martinovic and Van Gool [40] propose an approach that learns two-dimensional split grammars from labeled building facade image examples. They use *Bayesian Model Merging* [70], which adds a Minimum Description Length prior on the grammar to make the grammar induction problem tractable. The facade images are parsed into classes of style features. A grammar that connects these features is found by finding an optimal trade-off between the description length of the grammar and the likelihood of the input data. This is in other words a search for the simplest grammar description that fits the data. Their method is capable of generating good new buildings and even outperforms similar approaches with manually designed grammar rules.

Three-dimensional Inverse Procedural Modeling

The three-dimensional inverse procedural modeling case is quite a bit more complex than the two-dimensional case. By enlarging the solution search space with an additional dimension the time and memory requirements of the algorithms grows significantly. More importantly it becomes more difficult to guarantee satisfaction of the more complex constraints.

Aliaga et al. [4] propose a system for creating new buildings in the style of others. A set of images of a real-world building are mapped to a simple geometric building model that is built by the user. Following this step, the user subdivides

the building into its basic style features, such as doors, windows and floors. Next an algorithm finds a style grammar that captures the repetitive patterns in these segmented features. Production rules are sought for the base, ground floor, repetitive floors and the roof. Finally, a user can build a new simple geometrical model and the style grammar is used to automatically subdivide it. This fills the new model with style features found in the example images. The system requires users to manually segment the building images mapped onto a simple geometric structure. They claim this process takes a few hours for every example building. Automating this process could simplify it significantly.

Bokeloh et al. [8] tackle the inverse procedural modeling task through partial symmetry. Inspired by the texture synthesis solution of Efros and Leung [21] their goal is to keep as much of the local structure of the example structure in the generated structures. New structures are guaranteed to be *r-similar* to the example structure: any new point x must match a point x' on the example structure within a local neighborhood of radius r. This constraint is similar to the adjacency constraint used by Merrel [43, 44], relying on only the local state of a neighborhood. Their technique cuts the example structure along curves with symmetric areas so that shape operations can replace, remove or insert shapes into the structure while maintaining *r-similarity* with the example structure in every local neighborhood. These shape operations are combined into a shape grammar that can produce new structures in the same style or assist a user in designing new structures in the same style. The *r-similarity* guarantees that new structures generated from a closed example structure will always be closed. The technique works on any type of geometry. A major limitation of this technique is that it can not handle structures that have little to no symmetry or similarity. The technique is highly successful for extending or shrinking shapes because it relies on the cuts between symmetric areas.

2.4 Procedural Methods: Challenges and Goals

2.4.1 General Challenges

Procedural methods have a number of significant issues. Creating a procedural method or rule set that exactly defines a space of possible artifacts is, as previously discussed, very difficult. This led to the challenge of inferring and imitating the style of example content, and the field of example-based procedural methods. Aside from this, procedural methods face a number of other challenges.

To ensure their quality and consistency, most types of content require various constraints to be satisfied. In video games specifically [72], there are strict structural constraints that need to be satisfied in order to ensure playability of the game. These constraints can be introduced into the set of rules although it will often be exceptionally difficult to do so [64]. Extensive and complex rule sets may need to be changed completely to be able to ensure satisfaction of just a single constraint. Tweaking the generation process requires editing the procedural rules and parameters. The embedded constraints will stand in the way of easy revision for two reasons. They will complicate and reduce the interpretability of the shape sets. Secondly, the

rules may no longer ensure satisfaction of the constraints after being modified. This will need to be reassessed after every revision. When multiple different constraints need to be satisfied these issues only get worse. A simple solution tests the results after generation and only selects the ones that satisfy the constraints. This is not ideal for a number of reasons. There is no guarantee that the process will find an artifact that is satisfactory in a reasonable time. For generators meant to produce content *on the fly* this is entirely unacceptable. Ideally every artifact produced should satisfy the required constraints, but this is not a necessity. Rewards can be given to artifacts that satisfy certain constraints fully or to a certain degree. The piece of content that is eventually picked will be the one with the highest reward. This is a problem that must be addressed in many PCG systems, but can be addressed in different ways depending on the specified content [78]. A *hybrid* approach for content with a few necessary primary constraints and a number of less important secondary constraints, can generate a number of artifacts that guarantee satisfaction of the primary constraints in a limited time. These will be tested on the secondary constraints where the best performing artifact will be chosen.

Many procedural generators produce generic and unoriginal content [75]. These artifacts are highly similar, with insignificant variation and detail. After witnessing a limited number of generated artifacts, a human can often already guess what all other generated artifacts will be like. A point is reached where any variation of the generated artifact will not elicit any surprise or interest. While not guaranteed, human designers are capable of the creation of much more surprising and interesting content. Generated content should strive to reach human-designed qualities, without any discernible difference between computer and human designed content. As previously mentioned, solving this issue is related to the far-off goal of reaching computational creativity [36].

2.4.2 Machine Learning Challenges

Aside from a summary of the PCGML field in [72] Summerville et al. give an overview of current and future goals and challenges specific to PCGML.

The most relevant challenge in using ML generators is the scarcity of data. For some types of content, this is not a significant issue, while for others, such as video game content, it is. Most games have exceptionally small amounts of data to train the machine learning model. Even large games provide very limited datasets compared to, for example, image datasets. The small amount of data could be artificially enlarged, data could be sought from alternative sources or techniques could be used and developed that do not require such large datasets.

Another important challenge is allowing input into the generation process. Letting designers edit parameters or constraints in the system gives them a chance to *expose and explore* the generative space. Controlling the generative space is a considerable issue for black-box machine learning models in particular. Trying to change the output of a model that is not interpretable, or difficult to predict, through parameters is hardly an easy task.

Yannakakis and Togelius [84] discuss the additional difficulty ML methods have with satisfying constraints. This is a difficulty in designing any PCG system, but is greatly amplified when working with ML generative models such as generative adversarial networks and variational autoencoders. They can generate images that clearly look like the content they have been trained on but contain subtle faults. These faults can invalidate important constraints and must be examined after generation to ensure the constraints are satisfied. For example, when generating a maze from example mazes there is no guarantee that an exit or entrance will be present in every generated artifact. For these reasons highly successful machine learning methods in other fields are less interesting than methods with more direct control over constraints for most procedural generation applications.

2.4.3 Other Example-based Challenges

Many inverse procedural modeling techniques predefine types of rules and features. This allows more control over the rule set that will be found but limits the possible generative space significantly. The methods by Teboul et al. [74] and Martinovic and Van Gool [40] parse a facade image into specific feature classes such as windows, walls and doors. If these feature classes are not inclusive to all possible types of features they could be misrepresented and as such misused in the generated artifacts.

Some methods require the manual definition of basic elements or the manual parsing of the examples [4]. Ideally all steps of an example-based method can be automatised, requiring little to no interaction with the user. A central motivation for these methods was simplifying the difficult process for the users.

The limited, and arguably, most successful example-based procedural methods for three-dimensional structures of Merrel [43, 44] and Bokeloh et al. [8] both rely on a highly similar constraint to ensure similarity between the generated and example models. This constraint ensures that the local neighborhood of any point x in the generated artifact matches a neighborhood of a point x'. This limits these methods to exact symmetry and similarity in the examples and require exactly matching parts to generate structures that differ sufficiently from the examples. Additionally, every step of the generation process relies entirely on the current state of a local neighborhood, not taking the higher-order structure of the buildings into account.

Chapter 3

Shape Inference

Inspired by the inverse procedural modeling techniques that induce a shape grammar from two-dimensional facades [74, 40], this thesis proposes a method for inducing a set of rules from one or multiple three-dimensional example building structures. The method relies on finding the basic features, or shapes, for the style of the example and finding the relationships, or rules, between them. These rules resemble a shape grammar [67] with a number of additional limitations on both the shape and rule set. The grammar can be used to construct new artifacts in a similar style as the examples, by manual or automatic derivation. We discuss the grammars and generation of new structures in chapter 4.

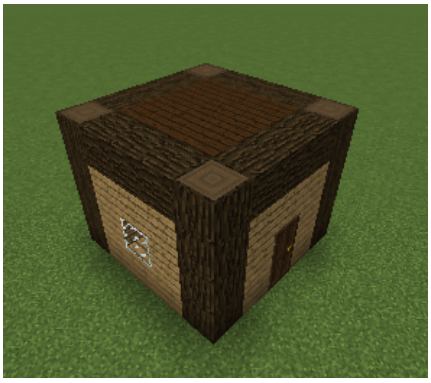
Overview This chapter explores finding the set of features, or shapes, from example buildings. We provide a specification for the input examples, the shapes and an algorithm for finding a suitable set of shapes. Instead of segmenting the examples into predefined feature classes [40, 74], we describe a cost function for the suitability of shapes and use a local search algorithm [1] to minimize the total cost of the shape set. Finally, we perform experiments on a number of algorithm and parameter variations and discuss their effects on the final set of shapes. In chapter 4 we discuss the grammar specifications, finding rules and the generation of new buildings from the grammar.

3.1 Problem Statement

Given a number of example building artifacts E, we want to find a set of basic style features, or shapes, S that describe the style of E. The specifications of shapes in S and examples in E are described in this section.

3.1.1 Example Building Structures

Input examples E are composed of a set of elementary components placed at certain positions in the example. In a two-dimensional case, where style features and procedural rules are found for textures or images, these elements could be considered



(a) E1: a basic example.

(b) E5: a highly complex example.

Figure 3.1: *Minecraft* building examples.

pixels or larger sets of pixels. In a three-dimensional case these components can, for example, be polygons, larger sets of polygons or voxels. Specifically, this thesis will attempt to find the style features for existing *Minecraft* [48] structures, such as in Figure 3.1. The reasons for this choice are as follows. First of all, *Minecraft* inherently exists out of simple voxel-based building blocks that can easily be manipulated by an algorithm. There is no need for a complex pre-processing of the input example or to translate the output of the final generation algorithm to a visual artifact in the game world. MCEdit [19] is a framework for *Minecraft* that provides a simple interface into the game world. Filters can be written in python code and directly applied to extract information or edit the world. There is no need to build our own interface into *Minecraft*. Finally, *Minecraft* provides a simple and intuitive way to construct example structure in a small amount of time. This facilitates a straightforward process for testing out different types and variations of example structures. The enormous amount of community created content can be used as well.

As *Minecraft* structures are composed of voxels, so are our input example artifacts. A voxel is a *1x1x1* cuboid block placed at a certain position p in the three-dimensional coordinate space. Every voxel, or block, has a type t that defines the texturing and behaviour of the block. The set of all *Minecraft* block types is T. An example artifact e in E is defined as a set of blocks, where a block b_i is a tuple (t_i, p_i) with a type $t_i \in T$ and a position p_i defined as (x_i, y_i, z_i) in the three-dimensional coordinate space and $x_i, y_i, z_i \in \mathbb{Z}$. The position p_i of a block b_i is unique:

$$\forall b_i \neg \exists b_j (p_i = p_j)$$

3.1.2 Shape Specifications

We are looking for a set of shapes S that describe the style of example E. Thus, a shape s in S ideally describes a part of the style of an example e in E, such as a

wall, window, awning or any other building style feature. The shapes are not limited to any predefined feature classes. Instead the algorithm discussed in section 3.2 is encouraged to find shapes that are likely style features. It will search for what the cost function considers a suitable set of shapes. There is, however, no guarantee that a shape will represent or be recognizable as a style feature. This section provides a formal specification of shapes and touches on a few related concepts and additional considerations.

Specification A style feature, or shape, s is a segment of an example building e in E. It is a subset of the blocks in example e: $s \subseteq e$. All blocks in s are connected such that they form coherent segments of the example artifact e. Consider a shape s a graph g where the vertices v_i are the blocks b_i in s and blocks whose positions are directly adjacent form an edge between their vertices. Two blocks b_i and b_j are directly adjacent if their Manhattan distance is equal to 1:

$$d(b_i, b_j) = \|b_i - b_j\|_1 = 1$$

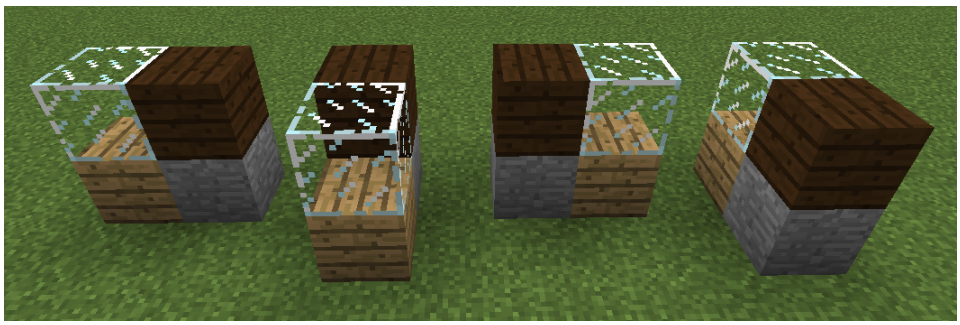
We enforce that all vertices in the graph g must be reachable from all other vertices. So for all pairs of vertices (v_0, v_k) there is a sequence of vertices $v_0, v_1, v_2, ..., v_k$ where the edge (v_{i-1}, v_i) is in the set of edges for $1 \leq i \leq k$. If there is a path from every block to every other block in the shape it is ensured to form cohesive segments without any excess unconnected blocks.

Restrictions This specification allows a shape to take on highly complex three-dimensional forms, such as the entire example structure, if all blocks in the example structure are connected. We can limit the complexity of shapes by restricting them in a number of ways. By enforcing that all blocks in the shape share the same position on a single axis, we limit shapes to a single plane. In this case all positions p_i of blocks b_i in a shape s are either (d, y_i, z_i), (x_i, d, z_i) or (x_i, y_i, d) where $d \in \mathbb{Z}$. This allows three types of shapes in the xy, yz and xz planes. Additionally, these planar shapes could be limited to rectangular shapes, where the blocks form a rectangle without any missing or protruding blocks. As rectangular shapes may suffice as a description for two-dimensional facades [74], these may as well suffice for three-dimensional buildings consisting of facades. We leave the possibility of diagonal, or other, shape specifications as future extensions. This leaves us with three distinct shape specifications:

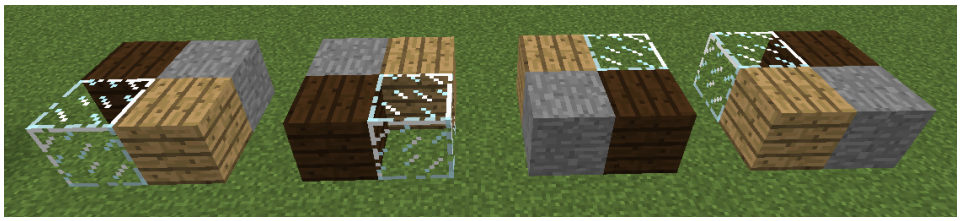
- the three-dimensional specification without additional limitations (as in Figure 3.8)
- the planar specification, where shapes are limited to a single plane (as in Figure 3.7)
- the rectangular specification, where planar shapes are limited to a rectangular form (as in Figure 3.6)

Not one specification is chosen over the others because they each have their possible merit for specific example structures. Simple structures will benefit from a simpler description, while more complex parts of structures are more difficult to model with a simpler shape specification. Describing slanted roofs with only rectangular shapes will give a large amount of small shapes, as no diagonal shapes are allowed. It might be more interesting to describe a slanted roof as one or more larger three-dimensional shapes.

Matching shapes Two shapes are considered to match when they contain the same blocks types in the same configuration.



(a) Matching vertical shapes in the xz and yz planes rotated along the z axis.



(b) Matching horizontal shapes in the xy plane rotated along the z axis.

Figure 3.2: Examples of possible matching shape rotations. Shapes that are rotated along the z axis are considered matching.

The position and orientation of the blocks in the examples E do not matter, only the positions of the blocks in a shape s relative to the other blocks in s. In other words two shapes s_i and s_j are matching shapes if a transformation τ_m exists that maps all blocks b_k from s_i onto their corresponding block b_k in s_j. If applying τ_m to all blocks in s_i results in s_j, these two shapes are deemed matching shapes. This transformation is of the form:

$$\tau_m(\begin{bmatrix} x_k \\ y_k \\ z_k \end{bmatrix}) = R \begin{bmatrix} x_k \\ y_k \\ z_k \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} \tag{3.1}$$

where $\Delta x, \Delta y, \Delta z \in \mathbb{Z}$ and R a rotation matrix along the z axis:

$$R = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

with:

$$\theta \in \{\pm\frac{\pi}{2}, \pm\pi, \pm\frac{3\pi}{2}\}$$

Figure 3.2 shows the possible rotations with R. This rotation is only possible along the z axis because other rotations would result in, what we consider, different style features. The two shapes in Figure 3.3 are not considered to match, because the vertical and horizontal versions of the window represent different style features, although their relative positions and block types do match.

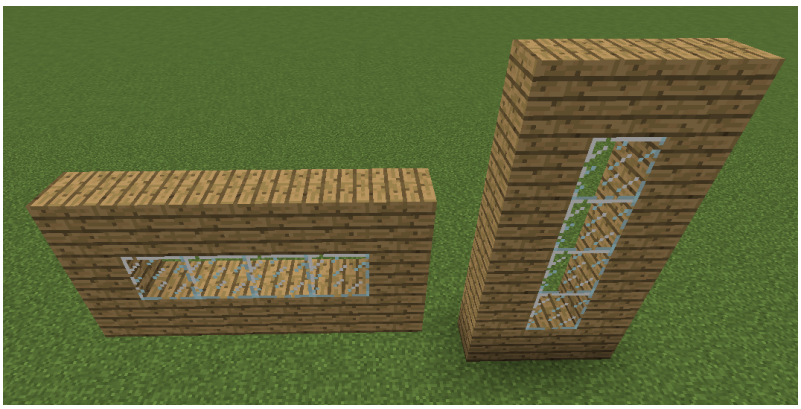


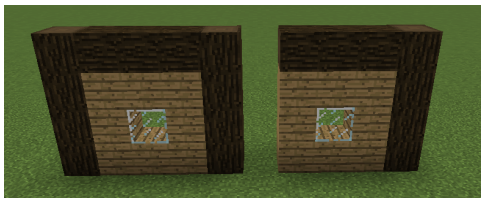
Figure 3.3: Example shapes that contain the same block types in the same configuration but are not deemed matching shapes.

Matching shapes can be reduced to a single shape with multiple possible positions and orientations. They represent style features that are present in multiple locations in the example structure. Matching shapes will play an important role in the first grammar specification discussed in section 4.2. Their existence allows the grammar to model a space of artifacts that is larger than just the example structures, while a grammar without any matching shapes will only be able to model copies of the example structures. Additionally, when multiple examples in E are used to model a style, matching shapes between two structures e_i and e_j will allow the combination of shapes and rules from both examples.

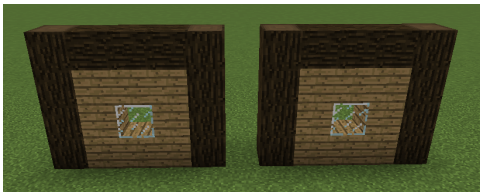
Overlap Finding a set of shapes S could be considered a pure segmentation of the example structures E where every block is present in one, and only one, shape. In this case it follows that:

$$\forall b(b \in E \rightarrow \exists s_i(b \in s_i)) \tag{3.3}$$

$$\forall b(\neg\exists s_i, s_j(s_i \neq s_j \wedge b \in s_j)) \tag{3.4}$$



(a) Two shapes from example 3.1a without overlap.



(b) Two shapes from example 3.1a with overlap.

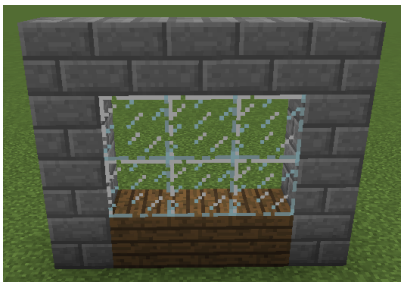
Figure 3.4: Both shapes in both figures represent two perpendicular walls with a shared corner in example 3.1a. This is an example of why we consider allowing overlapping shapes, or the inclusion of blocks in multiple shapes. In some cases it makes sense for blocks to be shared among multiple shapes. Additionally, these shapes match with overlap, allowing them to be reduced to a single shape.

Intuitively we can see that in some cases, such as Figure 3.4 it makes sense for blocks to belong to multiple shapes in the example. Shapes are thus not necessarily disjoint, but can overlap [46]. Here equation 3.4 does not apply to every block b, but 3.3 still does. The corner between two perpendicular walls has as much reason to belong to either shape. From this perspective a pure segmentation of the structure will result in a worse shape set because one shape is chosen over the other to contain these blocks. Additionally, allowing overlap could promote a higher percentage of matching shapes in the shape set, as in Figure 3.4. Two matching perpendicular walls cannot match in a pure segmentation unless the shared blocks are part of a separate shape. If this is the case, allowing blocks to be part of multiple shapes is not necessary and may have a detrimental effect instead, needlessly complicating the shapes. Both options have merit in certain situations. We expect shape sets with overall larger shapes to benefit more from allowing overlapping blocks.

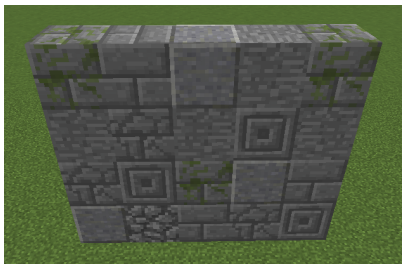
3.2 Finding a Shape Set

In order to infer the shapes present in the examples we discuss an algorithm for finding a set of shapes that should comply with the shape specifications defined in the previous section. Apart from making sure shapes meet these requirements we want to find *good* shapes that form a suitable description of and cover the entire example.

Ideally the shapes in the final grammar will describe the style features present in the example structure. We assume that most style features will exist out of only a few block types. The window in Figure 3.5a consists of glass, the windowsill of wood and the surrounding wall of stone blocks. These small style features consist of a single block type and the combined feature that covers the entire wall only consists of three block types. However, this does not apply to certain features such as intricately tiled walls and floors, as in Figure 3.5b. A shape that describes the entire building is likely too complex whereas a shape consisting of a single block will be too simple, but the exact boundary between too simple and too complex is hard



(a) Simple feature.



(b) Complex tiling.

Figure 3.5: A simple (small amount of block types) and more complex (many block types) feature. Most style features consist of a limited amount of block types. In some cases, such as a more complex tiling of a wall or floor in 3.5b, this does not hold.

to define and will vary per example structure. We strive to find shapes that are simple, yet complex enough to still have merit as a shape. Larger and more complex features, such as an entire wall, could be considered a single shape, or decomposed into smaller, less complex, shapes. Unlike the work of Martinovic and Van Gool [40] this is not a search for the simplest description that fits our examples, but a search for a balance between a simple components and a simple description consisting of these components.

3.2.1 Cost Function

In order to find a suitable set of shapes, we define a cost function that gives a lower cost for better shape sets while giving a higher cost for worse shape sets. What exactly constitutes a better shape set will specify the cost function. This cost function will be minimized by applying operations that modify the shape set and decrease the cost.

Basic Cost: Simple Shapes and Small Shape Sets

As mentioned earlier, a suitable set of shapes consists of shapes that are neither too simple nor complex. Shapes should be large and complex enough to represent an actual style element, but should be kept small and simple enough to avoid representing too many style elements in a single shape. Two concrete measures are introduced to find this balance.

We limit the complexity of shapes by increasing the cost of complex shapes. As a measure for this complexity, we use the entropy E_s [59], or the measure for the information content, of a shape s. The entropy rate will favor compact and homogeneous shapes, while favoring shapes overlapping with only a single style feature [38]. A shape that consists of only blocks of a single type does not contain much information, whereas more complex shapes that consist of multiple block types contain more information. Because style features are expected to have a small amount

of block types, shapes that represent style features are expected to have a lower entropy than more complex shapes. The entropy of a shape is defined as:

$$E_s = -\sum_{i=0}^{n} P(t_i) log_2 P(t_i) \tag{3.5}$$

where t_i is a block type present in the shape s and n the amount of block types in the shape. The entropy for a shape with a single block type is 0, and will increase for every additional type. Aside from the number of block types in a shape, the occurrence rates of the types will adjust the entropy values. The entropy value will increase for more even distributions of the block types in the shape.

If our cost function is simply the sum of all shape entropy values, a perfect cost of 0 is achieved when every shape consists of a single block type. Thus a large shape set consisting exclusively of minimal shapes, that contain a single block, will be considered a suitable set. To counterbalance the entropy, we introduce a cost for the amount of shapes in the set. The shape set with minimal shapes will no longer be optimal when the cost for the size of the shape set outweighs the minimal cost of the shapes. The *description length* or DL is defined as the number of shapes in the shape set:

$$DL = \#S$$

The cost function to be minimized is:

$$cost = (1 + DL)^{\alpha} \sum_{i=0}^{DL} E_{s_i} \tag{3.6}$$

where s_i is a shape present in the shape set S and α is a weighting parameter that provides a balance between the importance of simple shapes and the size of the shape set. The description length is increased by one to remove a strong bias for shape sets of size one. Otherwise shape sets of size 1 would not be affected by the α parameter and could skew the cost function in favor of just a single shape.

This final cost function in equation 3.6 can be minimized to find a small set of simple shapes. This entropy and description length based cost function is considered the basic cost function. The basic cost can be modified to take other attributes into account. We address a few extensions of the cost function here.

Discount for Matching Shapes

Matching shapes represent style features that are present in multiple locations in the example artifact. They give us much more information about how to describe a style. A shape set that consists of exclusively unique shapes provides insight into one possible configuration of the shapes, while shapes used multiple times in different locations are able to give more information about various other possible shape arrangements. These insights into the configuration of the style are important when finding a grammar for describing the style of the examples in chapter 4. The basic cost function gives no incentive to find matching shapes. We can introduce a

discount in the cost function for matching shapes, or the number of matching shapes, in order to promote the discovery of more shapes that can be reduced.

As any matching duplicate shapes are redundant in the shape set they can be removed from it. However, they must be kept in the shape set as long as the algorithm for finding a shape set has not finished and information about their original positions and orientations must be kept in order to find the set of rules at a later time. The cost of an arbitrary amount of matching shapes is equal to solely the cost of one of these shapes. The discount is applied by temporarily removing any matching shapes, while keeping one copy, from the shape set before calculating the cost. This new shape set is S_r. The cost function to be minimized is the basic cost function 3.6 where s_i is a non-duplicate shape in S_r and the description length is the number of remaining shapes in the shape set after removing duplicates:

$$DL = \#S_r$$

A potential issue is that small shapes, such as shapes that consist of one block, are more likely to match another shape. The cost function will turn strongly in favor of these small shapes, when this discount is applied. This issue can be dealt with by strongly increasing the α parameter to increase the importance of a small set of shapes.

Increase Focus on Less Complex Shapes

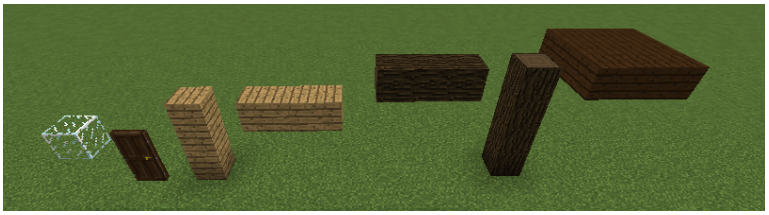
The basic cost function 3.6 increases for more complex shapes that contain more block types. This growth is, however, not very large and does not increase more when a new block type is added to an already complex shape then when it is added to a simple shape. The shapes in Figures 3.5a and 3.5b contain one and seven different block types respectively. The entropy cost of the former is 0 and 0.286 with an additional block type added, whereas the entropy cost of the latter is 2.733 and 2.871 when appended with an additional block type. The entropy cost difference when adding new types is limited and shapes that become more and more complex do not incur an increasingly higher cost because of it. In some cases it might be interesting to explicitly increase the cost of more complex shapes, so that the number of block types make a significant difference in the cost. This can be achieved in various ways, such as multiplying the entropy of a shape with the number of block types present in that shape. The cost function to be minimized becomes:

$$cost = (1 + DL)^{\alpha} \sum_{i=0}^{n} E_{s_i} T_{s_i}^{\beta} \tag{3.7}$$

where T_{s_i} is the number of block types present in shape s_i and β a weighting parameter for the amount with which the amount of block types increase the entropy cost.

It must be noted that in most cases this cost function will be redundant in the sense that the same resulting shape sets can be achieved with the basic cost function and a certain α value.

(a) $\alpha = 1.0$



(b) $\alpha = 0.5$

Figure 3.6: Resulting shape sets for example 3.1a for different α values and the basic cost function. The **rectangular** shape specification was used and overlap allowed. Duplicate shapes are removed.

Additional Cost Functions

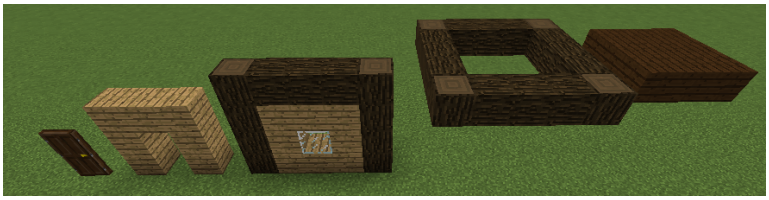
It is possible for the user to design an additional cost function with additional considerations that he deems important. We mention a few possibilities here.

In order to limit very large shapes being found, one could enforce a certain size limitation for shapes by adding an additional cost for large shapes starting from a certain size. If the user defines certain block types that do not match and are not expected to be present in the same style features, their inclusion in the same shape could incur a higher cost. One could, for example, define that stone blocks do not match with wood blocks because shapes that share both are expected to represent multiple style features. These shapes will be avoided during the optimization of the cost function because of this extra cost. What shapes do or do not match depends on the example structures, and the knowledge the designer has about the style of the examples.

3.2.2 Minimize the Cost

The cost is minimized by repeatedly executing operations on the shape set that decrease the cost until convergence. This optimization is a local search [1] in which every operation makes a local change that modifies the set of shapes into a neighboring set of shapes. This process resembles a region growing procedure used to segment images [2] and three-dimensional meshes [58]. We use a hill-climbing algorithm, as defined in algorithm 1, that guarantees convergence to a local optimum. The shape set will thus converge to a locally optimal set. Hill-climbing does not guarantee convergence to a global optimum so an entirely optimal resulting set of shapes is not assured. The resulting locally optimal shape sets will often be sufficient but it is

(a) $\alpha = 1.0$



(b) $\alpha = 0.75$

(c) $\alpha = 0.5$

Figure 3.7: Resulting shape sets for example 3.1a for different α values and the basic cost function. The **planar** shape specification was used and overlap allowed. Duplicate shapes are removed.

possible to find better local optima through more advanced optimization techniques [1, 9]. Even though hill-climbing will get stuck in bad local optima at times, it will lead us too far to tackle this problem. For now we prefer a simple and fast algorithm that finds sufficient local optima. Figure 3.6, 3.7 and 3.8 show the shape set results returned by the local search algorithm for different α values and shape specifications. The shape set in Figure 3.8b is an example of a bad convergence, where a shape got stuck in a sub-optimal situation.

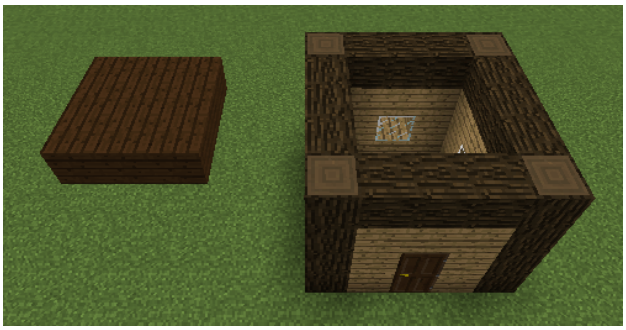
Operations

We define two distinct operations on shapes that can be applied by the algorithm.

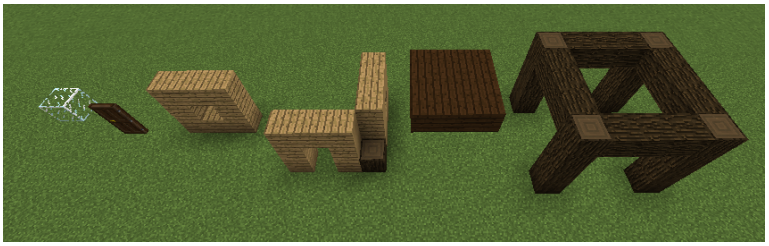
Merge The *merge operation* merges two shapes s_i and s_j into a single new shape s_n by combining both sets of blocks into one. This results in a new set of shapes S':

$$S' = (S \backslash \{s_i, s_j\}) \cup \{s_n\}$$
$$s_n = \{s_i \cup s_j\}$$

This new shape s must meet all requirements defined for shapes. All blocks in the shape must be reachable from all other blocks, so s_i and s_j require directly



(a) $\alpha = 0.25$



(b) $\alpha = 0.1$

Figure 3.8: Resulting shape sets for example 3.1a for different α values and the basic cost function. The **three-dimensional** shape specification was used and overlap allowed. Duplicate shapes are removed. Figure 3.8b shows a shape set that is clearly not optimal and has reached a local optima. The third shape from the right got stuck in a more complex shape than necessary.

neighboring blocks to be merged. Depending on the chosen specification, the shape must remain rectangular or fixed on a single axis. This limits the possible shapes that can be merged together.

Split The *split operation* splits a shape s into two new shapes, resulting in a new set of shapes S':

$$S' = (S \backslash \{s\}) \cup \{s_i \subset s, s_j \subset s\}$$

where $s_i \cup s_j = s$ and $s_i \cap s_j = \emptyset$. Once again these two new shapes must meet all requirements specified in the previous section. Rectangular shapes must be split into two new rectangular shapes, while planar and three-dimensional shapes have no additional limitations. Every possible way of splitting a planar shape will produce shapes fixed in the same axis, as will splitting three-dimensional shapes always produce three-dimensional shapes.

Use During the execution of the optimization algorithm, it is possible to use a combination of both operations or exclusively the merge or split operation. These schemes require different shape set initializations at the start of the optimization algorithm.

Initialization

Before the start of the optimization algorithm the shapes are initialized from the input examples E.

Merge When limited to merge operations the shape set is initialized to a minimal shape set: each shape is a size 1 shape with just a single block from the examples in E:

$$\forall b_i \in E : s_i = \{b_i\}$$

During execution of the hill-climbing algorithm these minimal shapes will be combined in an agglomerative fashion, akin to an agglomerative clustering procedure [56].

When using the rectangular or planar shape specification a shape is limited to a single axis. Once we merge two shapes one axis is no longer fixed. This may not be an ideal merge that will not lead to a good shape describing a style feature. This locks the block out of other potential good shapes. In order to improve the resulting shape sets we initialize each shape three times, belonging to every plane, and limit a merge operation to only merge shapes of the same plane, even if they would produce a shape that follows the specifications. At the end of the hill-climbing algorithm we ensure all blocks in E are part of the shape set S and remove the redundant shapes that are entirely covered by other larger shapes. Similarly, when overlap is allowed a block can be present in multiple shapes. With this initialization, once a block is part of a shape it can no longer be added to any other shapes. Thus, when overlap is allowed, once a shape is merged with another it is not immediately removed from the set of shapes. It can again be merged with another shape, as long as this does not form a duplicate shape already in the set. At the end of the hill-climbing algorithm we, once again, ensure that all blocks present in E are present in S and remove redundant shapes. This process makes it possible to easily allow overlapping blocks in multiple shapes.

Split When only executing split operations the shape set is initialized to a maximal shape set, where each shape contains as many blocks as possible, in accordance with the shape specification. For the three-dimensional specification this is simply a single shape consisting of all blocks in the example. The planar and rectangular specifications require a more complex initialization, where all initial shapes adhere to the conditions. This initialization is performed by executing the hill-climbing algorithm with only merge operations on a minimal shape set with the following cost function, that looks for the shape set with the least amount of shapes:

$$cost = (1 + DL)$$

The maximal shape set is split during execution in a divisive manner, akin to a divisive clustering procedure [56].

There is no need to duplicate the initial shapes in different planes, because the maximal shapes are already fixed to a certain plane. When overlap is allowed it is possible to split a shape into two shapes with a number of shared overlapping blocks.

Combination The combination of both operations could start with any initialization of the shape set. Starting from a minimal shape set will see the merge operation performed exclusively at first, whereas the maximal set will require splits. An initialization anywhere in between these two extremes will work as well. Our implementation and further experiments use the minimal initialization.

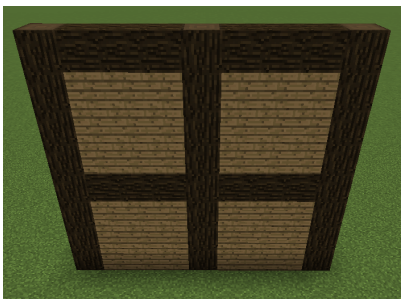
```
Data: E an example artifact, args a list of algorithm arguments (operations,
      shape specifications, cost function,...).
Result: S a set of shapes for which the cost has been minimized.
S ← initialize_shapes(E, args[operation]);
while operation applied do
    for op ← possible operations on S do
        new_shapes ← apply_operation(op, S, args[operation]);
        new_cost ← shapes_cost(new_shapes, args[cost]);
        if cost ≥ new_cost then
            S ← new_shapes;
            cost ← new_cost;
        end
    end
end
```

Algorithm 1: Hill-climbing pseudo-code. The method initialize_shapes(E) initializes the shapes from the examples E, apply_operation(S) applies an operation on the shape set S. The first operation that decreases the cost is applied. The best operation, that decreases the cost by the most, is not sought during every iteration. When no operation can be applied that does not increase the cost, the algorithm has converged and the resulting shape set is returned.

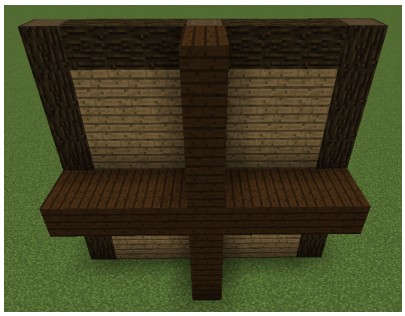
3.2.3 Limitations

This method of finding style features has a number of limitations. As previously discussed, the resulting shape sets are not guaranteed to be globally optimal and can get stuck in bad local optima. Furthermore, shapes are not guaranteed to be good representations of style features. The algorithm relies on style features being recognizable because they exist out of a small amount of block types, which differ from the other features. This is limiting for a certain subset of features as in Figure 3.5b. Our method is limited to a single shape specification. It is thus not possible to describe an example with a set of rectangular and planar shapes with, for example, a three-dimensional shape for the slanted roof. The combination of different specifications could provide much better descriptions of the example structures.

Split on intersecting shapes One flaw of the entropy-based cost function is the fact that walls or other features such as in Figure 3.9a will form one single large



(a) Results in a single shape.

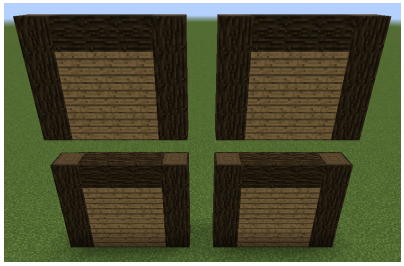


(b) Split on intersecting shapes.

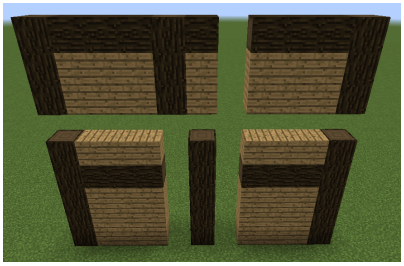
Figure 3.9: Entropy issue and possible solution with the use of domain knowledge. A wall as in Figure 3.9a clearly exists out of four different matching segments, but will result in a single large and simple shape, because this type of shape benefits from both the description length and the entropy components of the basic cost function.

shape, even though they contain a number of matching smaller features that could be reduced to a single smaller shape. This is exactly the type of shape the hill-climbing algorithm with the basic cost function is looking for: a large shape with a small amount of block types. This does not necessarily pose a problem if the wall is meant to function as a single large style feature, but otherwise it can. Applying the discount for matching shapes can form a solution, but this issue is not limited to compositions of matching features, as it also pose a problem for compositions of various features with similar block types. Simply applying this discount, or balancing the α value of the cost function will not deal with every case. Limiting the size of shapes, as discussed for an extension of the cost function in section 3.2.1, is another possible solution. However, there is no guarantee that the *ideal* shapes, as in 3.10a, will be found because another possible set of shapes, such as in Figure 3.10b, that makes even less sense than the initial shape is just as likely. We could look for matching shapes residing in the found shapes and split these up. There are many smaller matching shapes contained in most larger shapes: every two adjacent blocks of the same type form matching shapes. This method will necessitate us to somehow choose which matching shapes to split on. A possibility is to check for smaller shapes that are already present in the shape set, but there is no guarantee that the components of a large shape are present somewhere else in the example building.

In many cases two intersecting features, such as a wall and a floor in Figure 3.9b, will split the features into two distinct sections. We can use this domain knowledge to split large shapes after the execution of the hill-climbing algorithm. Depending on whether overlap of blocks in multiple shapes is allowed or not, the shapes are split accordingly. With overlap, this method will produce the *ideal* set of shapes as in Figure 3.10a. If there are no intersecting features this operation will, of course, change nothing. Shapes with many different intersecting shapes will likely be split up excessively, unnecessarily fragmenting good shapes. This operation is limited to the rectangular and planar shape specifications and should only be used with sets of large shapes.



(a) The *ideal* split.



(b) Maximum size constraint of 20 blocks per shape.

Figure 3.10: Possible splits for the feature in figure 3.9a.

3.3 Evaluation

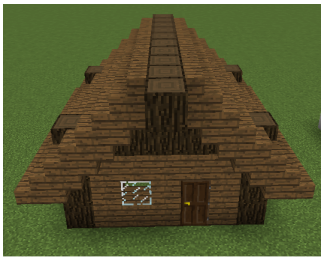
We perform an evaluation on the shape inference method discussed in the previous sections. The following methods discussed in chapter 4 rely on the inference of a shape set from the examples. These methods do not require an *ideal* shape set, but could in theory work with any shape inference results, as long as they exist out of a satisfactory amount of shapes of reasonable size. We consider a shape set as unsatisfactory when the example is covered by just a single shape or a shape for every block in the example. In spite of this, shape sets that better describe the style elements present in the examples will result in more interpretable generators, which lead to a more understandable generation process and simpler modification process. This evaluation is an exploration of the resulting shape sets rather than an evaluation of either the correctness or value of the different specifications and parameters of the algorithm described in the following section. We apply a set of procedures with different algorithm choices and parameters, discussed in sections 3.1 and 3.2, on a set of examples. We discuss the effects, side-effects and limitations of these procedures in regard to the results of the experiments.

3.3.1 Examples Used

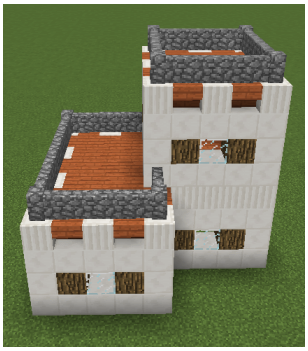
The example structures used in these experiments, as seen in Figure 3.1 and 3.11, were chosen to encompass a wide test set of various complexities and structural features. This set consists of example structures frequently tested during development and various other structures built by *Minecraft* community members [1]. The examples created by community members were considerably sized structures with many similar floors and a variety of additional details, such as furniture, spread throughout the buildings. In order to reduce the complexity of the resulting shape sets and the time spent on executing our procedures on highly complex examples we removed excess details and reduced the size of these structures by removing floors.

[1] retrieved from https://www.planetminecraft.com/projects/

(a) E2: multi-floor with railings.



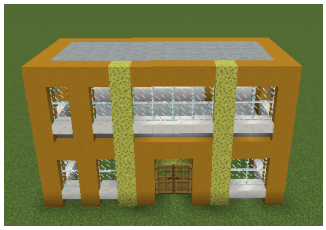
(b) E3: simple slanted roof.



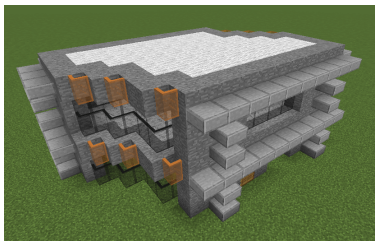
(c) E4: more complex multi floor with flat roof and railings.

(d) E6: more complex multi floor with slanted roof.

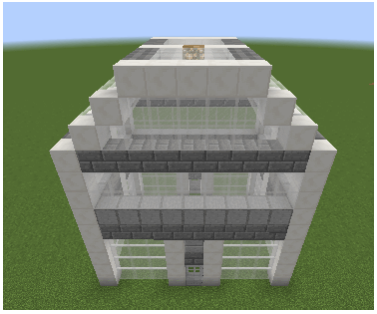
(e) E7: cylindrical tower.



(f) E8: office building with more complex facade.



(g) E9: diagonal walls.



(h) E10: simple, but large, empty structure with slanted roof.

Figure 3.11: Additional *Minecraft* building examples.

3.3.2 Experiments

The procedures that were explored during these experiments consider a number of different algorithm options and parameter choices that we expect to have a considerable effect on the resulting shape set. These are:

- shape specification: rectangular, planar and three-dimensional
- hill-climbing operation: merge, split and the combination
- cost function: basic, increased cost for multiple block types (surcharge) with β set to 1, discount for matching shapes
- allowing intersecting shapes to split each other: on or off
- α parameter: $\{0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2, 5, 10, 100\}$

Shape attribute measurements on these sets allow the estimation of the effects of these various procedures. Attributes such as the number, size and complexity of shapes describe the overall composition of a shape set and the number of matching shapes give an idea of the procedure's ability to find matching shapes. As a measure for complexity of a shape, in these experiments, we use the ratio of the number of block types present to the size of the shape. The number of shapes and matching shapes are taken over every shape set, and the shape size and complexity are averaged over all shapes in that set. In the following experiments, we execute the shape inference algorithm for all other algorithm choices and parameters and take note of the aggregate measurements of the results. We compare our expectations to these results and discuss the effects and limitations of certain parameter values and algorithm options. Our algorithm is deterministic, in the sense that performing shape inference on the same example with the same procedure will always lead to the same results, thus requiring a single execution for every example and procedure combination.

The effect of splitting on intersecting shapes The operation that splits shapes on intersections with other shapes was defined in section 3.2.3 to take advantage of domain knowledge that walls and floors cut other features into separate components. Example buildings without significant intersecting features, such as 3.1a and 3.11h, should barely be affected by this operation. We expect that in more complex buildings, such as example 3.1b and 3.11g with a lot of intersecting shapes, this optional operation will result in a disintegration of the shape set, decreasing the quality of the shape set instead. For relatively simple examples with a few possible intersecting shapes, such as example 3.11a and 3.11c, we expect the operator to perform well when the shape set consists of large shapes, decreasing the size of shapes and increasing the percentage of matching shapes in the set.

We compared the results of a number of relevant examples with and without the use of this operation in table 3.1. As we have not defined the application of this operation on three-dimensional shapes, we only consider the rectangular and

Example	Split	# Shapes	% Matching	Shape size	CR
E1(3.1a)	No	23.41	46.6%	12.504	0.365
	Yes	25.458	46.7%	12.331	0.383
E2(3.11a)	No	49.595	54.3%	20.498	0.355
	Yes	56.896	61.1%	14.6	0.383
E4(3.11c)	No	82.301	43.9%	19.102	0.382
	Yes	106.979	50%	12.037	0.44
E5(3.1b)	No	217.511	49.5%	15.645	0.424
	Yes	368.28	60.2%	8.298	0.541
E9(3.11g)	No	110.468	55.6%	21.117	0.338
	Yes	134.329	61.2%	14.638	0.38
E10(3.11h)	No	121.331	64.5%	28.682	0.327
	Yes	130.417	64.7%	21.531	0.354

Table 3.1: Results for a set of relevant examples with and without **splitting on intersecting shapes**. Measurements are averaged over the results of all examples and all procedure combinations, except for the three-dimensional shape specification for which this operation was not defined. CR refers to the complexity ratio, or the amount of block types over the size of a shape.

planar shape specifications. As expected, examples without any interior features, such as 3.1a and 3.11h, see little to no change. Because this operation relies on specific domain features, if these are absent it will have little effect. In spite of this, the number of shapes are slightly higher when shapes are split by other intersecting shapes. In cases where, for example the walls of these examples consist of many small shapes, it is possible that these intersect and split each other into even smaller shapes. We see that all other examples are affected to a much greater extent, decreasing the size of shapes and increasing the percentage of matching shapes.

Median size	Split	# Shapes	% Matching	Shape size	CR
$s < 3$	No	237.309	88.1%	2.2	0.76
	Yes	246.296	86.2%	2.08	0.743
$3 \leq s < 10$	No	53.974	46.6%	8.735	0.225
	Yes	96.678	46.5%	7.226	0.271
$10 \leq s$	No	9.109	22.7%	42.389	0.1
	Yes	15.225	30.5%	33.102	0.125

Table 3.2: Results with and without **splitting on intersecting shapes** for different median sizes of shapes in the shape sets. Measurements are averaged over the results of all examples and all procedure combinations, except for the three-dimensional shape specification for which this operation was not defined.

In table 3.2 we see that on average only relatively small shapes sets with a median shape size above 10 result in a higher matching percentage from this operation. These shape sets have reason to be split after the hill-climbing algorithm, while sets

with on average smaller shapes will be fragmented into even smaller shapes without increasing the percentage of matching shapes.

This operation should only be used on shape sets consisting of relatively large shapes to refine potential unnecessarily large shapes.

Cost functions We considered a number of different cost functions in section 3.2.1: the basic cost, the block type surcharge and the matching shapes discount. We compare the results of these three cost functions. The cost function that applies a surcharge for the number of types in a shape should result in less complex shapes. The cost function that introduces a discount for matching shapes is expected to result in more matching, but smaller shapes.

Operation	Cost	# Shapes	% Matching	Shape size	CR	Types
All	Basic	39.137	29.5%	95.294	0.152	3.641
	Surcharge	48.749	36.9%	60.062	0.171	2.64
	Discount	208.829	71.1%	36.534	0.698	2.061
Split	Basic	28.888	20.5%	106.542	0.12	4.133
	Surcharge	28.515	20.8%	90.92	0.123	3.641
	Discount	28.862	20.9%	106.38	0.122	4.145

Table 3.3: Results for all three **cost functions**. Measurements are averaged over the results of all examples and all procedure combinations.

The resulting average measurements in Table 3.3 show that the cost increase for the number of block types results in on average less block types per shape but a larger complexity ratio, because the size of shapes has decreased significantly. Whether or not this cost function can result in significantly different shape sets than by balancing the basic cost function remains up for debate. The percentage of matching shapes increases drastically when using the discount in all operations. When the shape set is initialized to the minimal set of shapes the identical cost function rapidly converges to an optimal cost for a large set of very small shapes. The shape set is not initialized minimally when using the split operation. When only considering this operation the percentage of matching shapes barely increases, occasionally finding a split that adds additional matching shapes.

These more advanced cost functions do not generally give very different resulting shape sets because of the optimization scheme that, in every step, executes the first operation that decreases the cost function. The potential of these more detailed cost functions is not revealed with our simple local search algorithm.

Hill-climbing operations The operation that minimizes the cost of the shape set during the hill-climbing algorithm together with the initialization of the shapes form various alternatives.

- Starting from a minimal initial shape set, with one block per shape, a merge operation is executed at every step.

- Starting from a maximal initial shape set, with as many blocks as possible in the shapes, a split operation is executed at every step.
- A choice is made between executing a merge or split operation at every step. The shape initialization can be minimal, maximal or somewhere in between. It is minimal in our implementation.

The expectation is that the combination of both operations will provide the most fine tuned shape sets. If a shape has been *overmerged* it is possible that it can be split at a later time. Because the merge and combined operations start from a minimal shape set, it is more likely for these to converge soon, resulting in large shape sets with small shapes. Conversely the split operation starts from maximal shapes and will be more likely to converge with on average larger shapes.

Cost	Operation	# Shapes	% Matching	Shape size	CR
All	Merge	137.609	58.5%	44.81	0.449
	Split	28.755	20.7%	101.281	0.122
	Both	131.741	58.7%	45.104	0.455
No Matching	Merge	54.25	39.6%	66.703	0.183
Discount	Split	28.702	20.7%	98.731	0.122
	Both	49.056	39.5%	67.271	0.181

Table 3.4: Results for all three **hill-climbing operation** possibilities. Measurements are averaged over the results of all examples and all procedure combinations.

The results in Table 3.4 confirm that on average the minimally initialized sets are more likely to converge to large sets of small shapes, while maximally initialized shape sets converge to smaller sets of larger shapes. We notice that the merge and combined operators produce very similar results. Performing hill-climbing with the combined operations, starting from a minimally initialized shape set, causes the first operations to follow the same path as when using only the merge operation. Occasionally a split operation will occur, resulting in slightly larger shape sets with smaller shapes.

The resulting shape set is highly reliant on the initialization of the set, because the local search algorithm will converge in the first local optima it finds.

Shape specifications We tested the shape inference procedure on the three different shape specifications defined in section 3.1.2: rectangular, planar and three-dimensional. The more constrained the specification is the smaller the search space becomes. Our expectation is that less restrained specifications have more options to find better shapes. This freedom will allow the shape inference algorithm to find larger shapes with limited block types. The ratio of the complexity of shapes to the size of shapes will be lower in general, meaning on average shapes are larger and contain less block types.

The average measurements for all three shape specifications in Table 3.5 follow our expectations. Less constrained specifications allow for larger shapes with less

Shape Specification	# Shapes	% Matching	Shape size	CR
Rectangular	127.982	57.2%	9.603	0.425
Planar	97.547	50.5%	23.458	0.335
3D	68.144	28.1%	168.207	0.253

Table 3.5: Results for all three **shape specifications**. Measurements are averaged over the results of all examples and all procedure combinations.

block types. In the case of three-dimensional shapes, the shapes found are often enormous, encompassing significant subsets of or the entire structure itself. Setting lower α values can alleviate this issue.

Alpha parameter The α parameter controls the weight of the description length, or the number of shapes, opposed to the entropy cost of the shapes. A higher value increases the weight of the description length cost, promoting a smaller set of shapes and thus larger shapes. An α of 0.0 will produce the set of minimal shapes because only the entropy is taken into account. Shapes with a single block type have a minimal entropy of zero and will not be counterbalanced by the cost of the description length. For every combination of a specific procedure and example a certain α value will produce the set of maximal shapes. At this point the cost of the description length will outweigh any effect of the entropy cost and a minimal shape set will be prioritized, which requires the largest possible shapes. The values in between these two extremities will result in a range from minimal to maximal shapes.

α	# Shapes	% Matching	Shape size	CR	% No matches
0.0	60.5	62.8%	5.296	0.211	7.7%
0.25	55	60%	5.342	0.211	15.2%
0.5	47	57.1%	6.214	0.201	20.9%
0.75	40	51.3%	7.1	0.197	21.7%
1	29	41.7%	10.875	0.161	23.7%
1.5	26	40%	13.067	0.16	26.7%
2	17	33.3%	18.25	0.152	30.9%
5	14	33.3%	21	0.148	31.6%
10	14	33.3%	21	0.148	31.6%
100	14	33.3%	21	0.148	31.6%

Table 3.6: Results for various **alpha parameter values**. Measurements are the medians over the results of all examples and all procedure combinations. The percentage of no matches refer to the amount of resulting shape sets that contain no matching shapes.

We test a number of α values on all examples in all other algorithm configurations in Table 3.6. The range of α values that produce reasonable shape sets depend on the example and the other algorithm options, such as the cost function or the shape

specification, used during inference. Larger α lead to larger shapes and a higher chance that there will not be a single matching shape in the resulting shape set. With an α value of around 5 or higher, the entropy costs of the shapes will on average be disregarded and the maximal shape sets will be returned. Generally an α value around 1.0 will provide reasonably sized shapes for the shape set.

3.3.3 Conclusion

Generally we recommend the following parameters that provide reasonable results in most cases. An α parameter value of around 1.0 yields a suitable balance between simple shapes and a simple set of shapes. The three-dimensional shape specification is unrestricted and tends to converge into enormously large shapes, while the rectangular or planar shape specifications find better, simpler shapes. We did not witness a significant difference in the cost functions, except for the matching shapes discount immediately converging when used in a minimal initialization. The hill-climbing operation could be chosen dependant on the preference for smaller or larger shapes, as the initialization mostly establishes the first local optima that will be found.

Chapter 4

Grammar Induction and Automatic Generation

After inferring a set of style features, or shapes, from example structures, rules can be found that describe the relationships between these style features. These rules form a grammar that can be used to construct and automatically generate new artifacts in the same style as the example structures. This chapter discusses the rule specifications, finding the rules that connect the shapes and the possibilities of generating new structures with these rules.

Overview We first discuss and implement a simple grammar, with a single rule type, that defines the style of the examples. This is an *additive shape grammar* that consists of exclusively additive rules that compose an already produced shape with another new shape. We describe its strengths and limitations, such as its unsuitability for generation through automatic derivation of the grammar and focus on local style. Finally, we discuss an extension for a grammar that is more suited for generation through automatic derivation inspired by split [82] and CGA grammars [49]. These were designed with automatic derivation in mind and handcrafted rule sets provide good results.

4.1 Problem Statement

Given an example building structure, or structures, E and the style features, or shapes, S for E we want to find a grammar G that describes the style of E with a set of production rules P and allows the creation and generation of new structures in the same style. The following two sections describe two distinct specifications of this grammar G.

4.2 Additive Shape Grammar

We first describe a simple grammar that makes use of the shapes obtained in chapter 3 and the relationships between them. When two shapes neighbor each other in the

example structure, they form two production rules that place one of the shapes in the same position and orientation relative to the other, already produced, shape. We refer to the grammar of these production rules as an *additive shape grammar* because all production rules in the grammar add a shape to an already produced shape. We explore the possibilities of automatic generation with the grammar and evaluate these results.

4.2.1 Grammar Specification

The additive shape grammar G is composed of the following:

- a finite set of shapes S, as specified in chapter 3
- an initial shape I that is selected from S
- a finite set P of production rules on the shape set S

Rule specification The set of production rules P consist of exclusively *addition* rules of the form:

$$S \rightarrow SS$$

where a shape from S is added on to another, already produced, shape from S. Two shapes make contact when a block $b_i \in s_i$ is directly adjacent to a block $b_j \in s_j$, as described in the shape specification in section 3.1.2. When shapes s_i and s_j do make contact in an example structure in E they form two rules:

$$s_i \rightarrow s_i s_j \tag{4.1}$$

$$s_j \rightarrow s_j s_i \tag{4.2}$$

When the shape on the leftmost side of the rule is present in the production, the second shape can be added to the production. The position and orientation of the second shape is transformed in order to comply with the same configuration of s_i and s_j in the original example structure. If s_j is placed on top of s_i in the example structure, and the position of s_i has changed in the production, rule 4.1 will add s_j to the production and change its position such that it is placed on top of s_i. Every shape has information about its original positions and orientations in the example structure. During production these can be changed by applying a production transformation τ_p to every block in the shape, which takes on the same form as the matching transformation τ_m in Equation 3.1. If a transformation τ_p was applied to s_i in the production and rule 4.1 was applied to it, the same transformation τ_p will be applied to s_j before being added to the production. Figure 4.1 provides an example of a number of production rules.

This specification of the *additive shape grammar* is a description of the example structure it was induced from, in the sense that it is possible to recreate the example, or subsets of the example, by expanding the production rules in the grammar. Instead of modeling the style of the example, this grammar will define just the example structure. In order to introduce variation in the grammar, by allowing it to describe

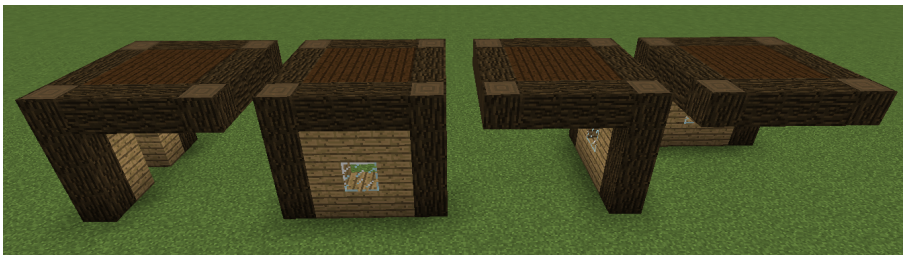


Figure 4.1: Rules for the rightmost *ceiling* shape in the shape set 3.6a from example 3.1a. There is a rule for every other shape in the shape set that makes contact with this shape. The rule produces the new shape in the same position and orientation relative to their positions in the example structure.

and generate more than just the example structure, we make use of matching shapes in the shape set.

Matching shapes A style feature, such as a window, that occurs multiple times in different positions on an example structure may be adjacent to different shapes in each of these positions. In the previous specification these matching shapes in different locations retain their own rules in accordance to the shapes that neighbor them. In the following specification we assume that matching shapes share rules amongst themselves. We provide the following scenario as an intuitive explanation. A window and balcony feature that are adjacent on the second floor of a building form rules in which they can be placed next to each other. A matching window feature is present somewhere else in the building, such as the third floor, without being adjacent to a matching balcony feature. In a new production, if the matching window features share rules, a balcony will be able to be placed adjacent to the window feature on the third floor. This was not possible in the original grammar specification. Sharing rules between all matching shapes will not always make structural or functional sense. This is the case in a similar scenario where a window is adjacent to a door on the first floor, and a matching window feature exists on the second floor. Here the door will be able to be produced adjacent to the window on the second floor. A door in the middle of a building facade will make no functional sense. It is very difficult to define which rules should be shared among which matching shapes, without predefining feature classes [40, 74]. Not allowing shapes on different floors to share rules could pose a solution to the problem in this scenario, but the question remains if it will take care of all problems of this nature and how many rules that do not form a problem will be removed because of it. This presents a choice between a broader or more general grammar for the example that will allow more variation in the generated artifacts, or a more specific grammar with less differentiation but more structurally coherent buildings. We choose the more general grammar that allows more variation but will lead to more rule derivations that make less functional sense.

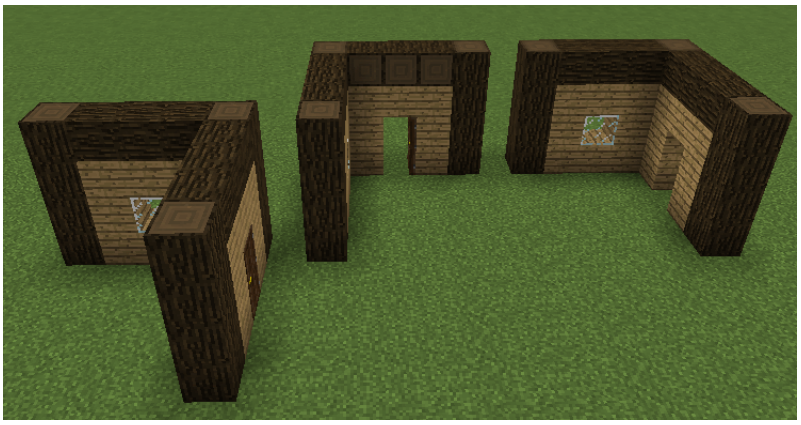


Figure 4.2: Matching shapes share rules. The rightmost rule production represents the original rule from the shape set 3.6a from example 3.1a, where the wall with a door can be placed next to the wall with a window. Because there are two identical shapes with a window in the example, this rule is shared with them. In the production of these rules (the left and center productions) the shape with a door is placed in the same configuration as the original rule, but transformed according to the position of the duplicate shape.

In this new specification the production rules 4.1 and 4.2 are extended to:

$$d_{s_i} \rightarrow d_{s_i} s_j \tag{4.3}$$
$$d_{s_j} \rightarrow d_{s_j} s_i \tag{4.4}$$

Where d_s is the set of shapes that match s, including s. Starting from a duplicate shape s_d in d_{s_i} it is possible to expand rule 4.3 that adds a new shape s_j to the production. Its new position and orientation will be relative to the original shape in the rule and the duplicate shape as seen in Figure 4.2. Both the transformation τ_m from s_i to the matching shape s_d, and the transformation τ_p performed on s_i during production are applied to s_j. The sharing of rules between matching shapes provides the possibility of rule expansions outside the original example structures space. Thus more matching shapes will provide more variation in the grammar, and in the resulting generative space. Additionally if two example structures e_i and e_j in E have matching shapes, these two examples will be linked in the grammar, because the rules their matching shapes share provide a bridge between both production spaces.

4.2.2 Finding Production Rules

Production rules are found by analyzing the relationships between the final shape sets S inferred from the example structures in E. The directly adjacent shapes are found for all shapes, not excluding duplicate matching shapes. These adjacency relationships each form two rules 4.1 and 4.2, which are extended the rules 4.3 and 4.4. The sets of matching shapes are reduced to a shape and a number of transformations

τ_m that transform the original shape in the rule s into the shapes that match s. These transformations are used when expanding a matching shape rule.

4.2.3 Creating and Generating New Structures

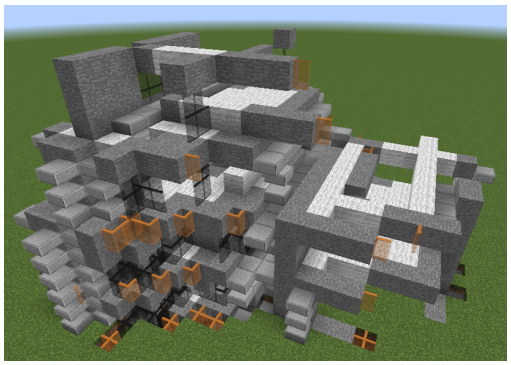
After the grammar has been induced from the example structures, it can be used to derive new structures that follow the grammar production rules. The set of produced shapes F contains all produced shapes, and is initialized to the initial shape $\{I\}$. This shape is chosen from S, and its position and orientation can be modified by a transformation τ_p. Starting from the initial shape I, rules are applied by choosing a shape s and a rule r whose leftmost shape s_l is s or a matching shape of s. The rightmost shape s_r is added to the production F after applying the same τ_p applied to the initial shape and after applying τ_m, in case s_l is a duplicate of the original shape in r. A new shape is randomly chosen from F, and the process can be repeated indefinitely. It is possible to manually derive the grammar and construct new artifacts in a similar style as the example structures. A design tool could assist a designer, by providing a user interface that allows the user to choose which rule to apply from a set of permitted rules in every iteration [45]. It is also possible to automatically derive rules in the grammar in order to automatically generate structures. We discuss this process in the following section.

Automatic Generation

When manually deriving the grammar to construct an artifact it is possible for a designer to ensure certain structural constraints by choosing the exact rules to expand. Rules that are not sensible in a particular context, such as adding a door to a facade on the second floor, can be ignored and the user can ensure creative consistency within the entire artifact. After automatic generation with the grammar, where rules are expanded in a stochastic fashion without any additional constraints, the generated artifacts are accumulations of shapes that are neither structurally nor creatively consistent. Outside of the features and the relationships between them, there is another higher-order layer of structure to buildings [46]. The additive shape grammar does not take this additional layer into account, only considering the local relationships between shapes. This will often lead to largely incomprehensible structures such as in Figure 4.3. In some cases the automatic derivations can provide an interesting unfinished concept for a building, that could be completed manually to form a coherent piece of content, as in Figure 4.4. The additive shape grammar allows indefinite application of new rules, which allows the generation of indefinitely sized new structures. On the other hand this implies that there is never a stopping point during derivation of the grammar. One must choose a stopping point for the derivation, such as after a certain number of rule applications or after one or multiple constraints have been met.

Constraints Various constraints can be used during or after the derivation of the rules to limit the possible productions, and improve the results. As discussed in

(a) From example 3.11b.



(b) From example 3.11g.

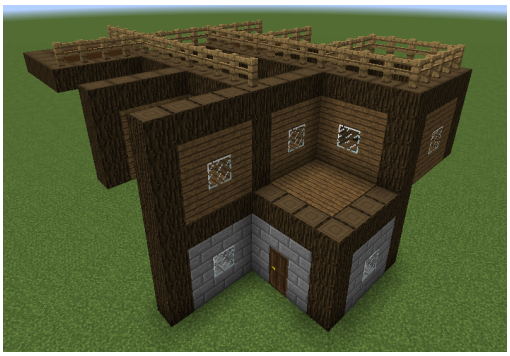
Figure 4.3: Examples of **automatic derivation** where too many small shapes and rules are randomly produced, while overlapping each other, and will produce largely inconsistent and incomprehensible results.

chapter 2.4 this is a significant challenge in many procedural methods. Introducing a constraint into a grammar or rule set is in most cases enormously difficult [64]. A search-and-test method that checks satisfaction of the constraints after generation and accepts or rejects an artifact based on the constraints is not ideal for many applications of automatic generation. There is no guarantee that a sufficient artifact will be generated in a certain time. This method can not be relied upon for *on the fly* generation, for instance during gameplay of a video game. The potential constraints that can be enforced on three-dimensional structures are extensive. We discuss limiting the size of a generated artifact and enforcing an enclosure constraint, that will allow the generation of somewhat structurally consistent buildings through automatic derivations of the additive shape grammar.

Size restriction We first define a constraint that sets a boundary for space in which shapes are allowed to be derived. In city generators and other applications, buildings are generated to fit into predefined plots of a certain size [49, 53]. Our generator must at the very least be able to generate structures in a limited space. The set of produced shapes is restricted to a range (x_r, y_r, z_r) where $x_r, y_r, z_r \in \mathbb{Z}$. We consider the size of the production x_p, y_p, z_p where $x_p, y_p, z_p \in \mathbb{Z}$, the distance between the smallest and largest block position for every axis. When adding a newly derived shape to the production that increases the size of the production to a point that the size is larger than the permitted range, this derivation is discarded and instead a new rule is derived. If this process fails a number of times and no shape can be found that does not pass over the set boundary the generation is halted. This allows us to restrict the generated artifacts to a chosen size in all dimensions.

Enclosure

The automatic derivations of Figure 4.4 represent structures that, if finished, could result in interesting new buildings in a similar style as the example structures. Notice



(a) From example 3.11a.

(b) From example 3.11f.

Figure 4.4: Examples of **automatic derivation** that provide a comprehensible concept for a building, which could be manually finished.

that finishing these buildings means filling in missing shapes, or removing redundant shapes, to form a coherent enclosed building with a clear distinction between the in- and outside. Enforcing enclosure will constrain the generative space and significantly improve the results of the generation process.

Enclosure specification We define enclosure as follows. When the shape specification is either planar or rectangular one of three axes of every shape s is fixed. Take for instance the rightmost shape in Figure 3.6a that is locked on the z axis. All blocks b_i in this shape s have a position p_i of the form (x_i, y_i, d), where d is the same value for every block b_i. This shape has an under- and upper side at a z position of $d-1$ and $d+1$. A shape s has two distinct sides $side_1$ and $side_2$, which exist out of the positions p_i from every block b_i in the shape s where the fixed axis is changed to either $d-1$ or $d+1$ where d is the value of the fixed axis for that shape. The shape s is enclosed when either $side_1$ or $side_2$ can not be reached through a path of empty *air blocks* starting from a position that is on the exterior of the generated structure. The sides that are unreachable are considered *inside* the structure, whereas sides that are reachable are considered *outside* the structure. Unreachable sides of a shape are enclosed by other shapes that block the path from the exterior of the structure. When both sides of a shape s are reachable s is not enclosed as both sides are *outside*. When both sides are reachable the entire shape s is confined *inside* the structure. For three-dimensional shapes there are no obvious distinct sides to the shapes. This makes it much more difficult to ensure enclosure of the structure. We do not extend the enclosure constraint for three-dimensional shapes.

Finding reachable sides A simple pathfinding algorithm that explores the space of produced shapes can find the reachable sides of the shapes in the production. The algorithm can move through empty *air blocks* in the production space. Starting from a position outside the production all reachable positions are explored. If at any step the algorithm reaches a single position that is considered a side of a shape that

entire side is considered reachable. Once the pathfinding algorithm has explored all reachable positions, we consider any shape for which both sides were reachable redundant. Removing all redundant produced shapes provides a production with consistent enclosure, as in Figure 4.5. We can remove all interior shapes, leaving only the *frame* of the building by removing the shapes whose both sides are enclosed.

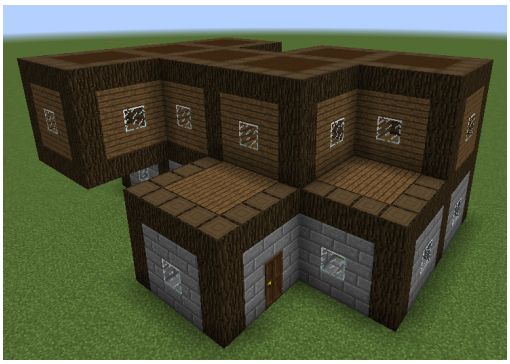
Limited enclosure While most buildings have an interior, not every style feature or larger part of a building is necessarily enclosed. Whereas the walls, floors and ceilings of a structure are usually enclosed by other features there exist a number of structural components where this does not hold. Fences and gates, support columns, and balcony railings are a number of features that are not usually enclosed. Enforcing enclosure on all shapes in the production will thus have the additional effect of removing shapes that do not require enclosure, removing details that might provide significant value. This can be seen in Figure 4.5b where no railings are placed on the roof of the building as in the example 3.11a it was based on. A potential solution marks shapes that are not enclosed in the example structures. These marked shapes are not removed when not enclosed in the produced shapes. In the following section we further discuss and evaluate the results and limitations of generation through the additive shape grammar.

4.2.4 Evaluation

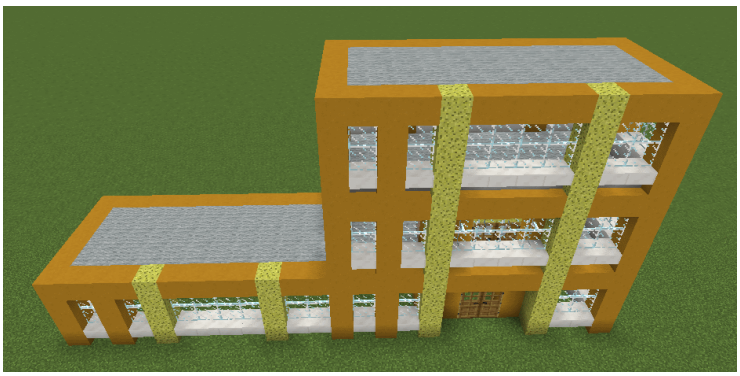
Results

While the results of an unconstrained automatic derivation of the additive shape grammar are unusable artifacts by themselves, solely enforcing the enclosure constraint can produce reasonable new buildings, as in Figure 4.5. Our method allows the inference of a grammar from multiple example structures as evidenced in Figure 4.6. This is a grammar for the shared style of both examples. Resulting generated structures can consist of both unique and shared features. Larger amounts of examples of a particular style will result in a more accurate grammar description of the style, if they contain unique features and relationships. Judging by these results our method works as intended. We infer shapes and a grammar that describe the style of a number of example structures. Finally, this grammar can be used to generate new structures in a similar style. The additive shape grammar is interpretable, as it is possible to follow along with the derived rules and to understand the process of generating new structures. The simplicity of this grammar and its rules helps to improve the understandability. It is not necessarily trivial to interpret, especially for grammars inferred from complex examples, but it can be modified by the designer to remove redundant rules, add new rules and add new shapes to the grammar. Even grammars with enormous amounts of rules are interpretable, because in every step a single shape is chosen to apply a rule to. These shapes have a limited number of rules defined by the limited shapes that neighbor or match the shape. The inferred grammar can be adjusted by the designer to generate a different space of structures. In spite of this, this method has a great deal of limitations.

(a) From example 3.1a with shape set 3.6a.



(b) From example 3.11a. All railings are removed because of the enclosure constraint.

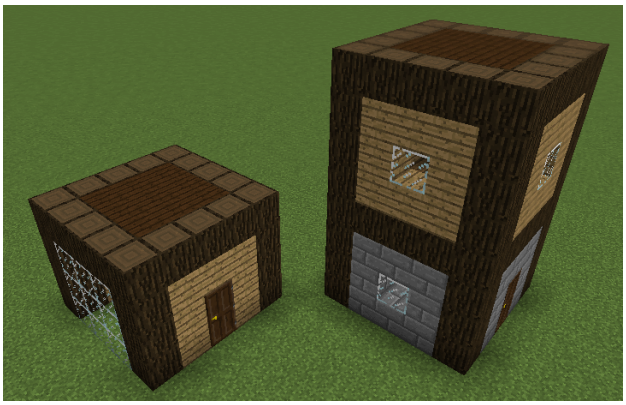


(c) From example 3.11f.

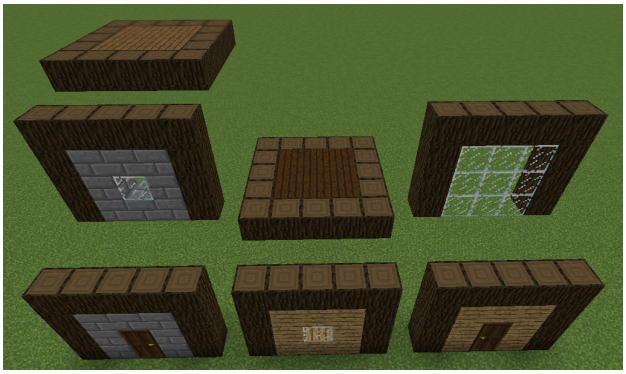
Figure 4.5: Examples of automatic derivation with the **enclosure constraint**. Note that for more complex examples, with more shapes and rules, the enclosure constraint will often remove most, if not all, produced shapes. This leaves us with an empty generated artifact.

Limitations

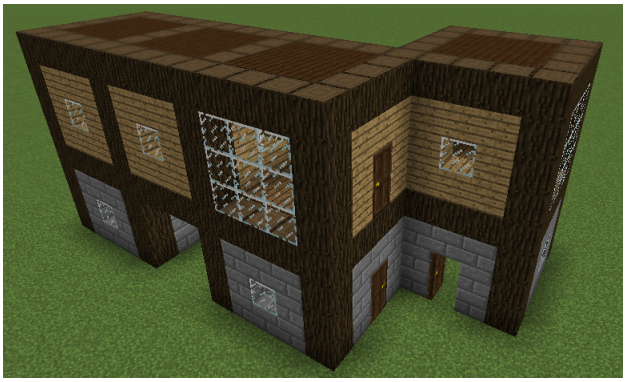
Complexity of examples, shapes and grammars We only achieve relatively reliable and valuable results for the simplest of examples. The more complex the examples the more shapes are needed to describe the examples and the more complex the resulting grammar will be. Randomly deriving these grammars results in largely incomprehensible results such as the artifacts in Figure 4.3. The additive shape grammar with the enclosure constraint more reliably produces sufficient results when the shape set exists out of large shapes with a limited number of rules between them, such as in Figure 3.6a. Shape sets with smaller shapes, such as in Figure 3.7c, rely on many more rules to be derived to form an enclosed structure. Even simpler shape sets with many matching shapes, such as in Figure 3.6b, have more rules between all shapes, which increases the chance of the derivation of rules that do not make sense in that particular context. Just as in model synthesis [42], these conflicts



(a) Two examples in a similar style, with matching and distinctly different style features present in the examples.



(b) The shape set produced for both examples with rectangular shapes, an α value of 1.0 and splitting on intersecting shapes after finding a shape set. The central shapes are the shared shapes between both examples, the left are the shapes unique to the larger example and the rightmost shapes are unique to the smaller example.



(c) A building generated from a grammar induced from both examples in 4.6a. The enclosure constraint was used. Both the shared and the unique features of the examples are present in this generated structure.

Figure 4.6: Inferring shapes and inducing an additive shape grammar for **multiple examples** in a similar style and generating new buildings from this style.

reduce the quality and consistency of the produced structures. It is possible for a newly produced shape to overlap other shapes entirely or partially. When blocks are allowed in multiple shapes a partial overlap during production is necessary, but it can occur that produced shapes become entirely covered by a new shape, or two shapes that are not related in the example structures intersect. This will only obfuscate the results. The simpler the shape set, the more complex the grammar and the more difficult automatic generation becomes. Shape sets with large complex shapes have simpler grammars that are more suited for automatic generation. Larger shapes, such as the ones used to successfully generate new structures in Figure 3.6a and 4.6b, may, however, capture more than one style feature in a single shape.

Matching shapes The additive shape grammar relies entirely on matching shapes in the example structures in order to generate buildings with any variance. Without any matching style features this method is only able to generate subsets or copies of the examples. Even if there are duplicate style features present in the examples they must be found during the shape inference procedure. This is an issue that similar previous example-based three-dimensional generators faced [8, 44] and one we believe to be an inherent to this problem. When inferring the style of an artifact with the goal of generating artifacts in a similar style, it will always be necessary to find similarities in the examples that can be exploited. All procedural methods rely on similarities, repetition and symmetry.

We may need to infer many shape sets with different algorithm parameters to find a suitable shape set with matching features. Even then, there is no guarantee that any matching shapes will be found in any configuration of the inference algorithm for a certain example. Improving the cost function and optimization scheme of the shape inference procedure to more reliably find matching shapes in the examples can alleviate the problems of this dependency to a certain degree. Another option is to weaken the matching shapes by finding shapes that match for the most part, with a limited amount blocks that do not coincide. As discussed previously, even if matching shapes have been found it does not always make sense to share certain rules between two matching shapes. These rules will be part of the grammar and can reduce the quality of the generated artifacts.

Enclosure In any newly generated artifact there is no guarantee that any part of the structure will be enclosed. After applying the enclosure constraint to a production without enclosed shapes, the resulting production will be empty. Even for the relatively reliable shape sets of large shapes, this is a possibility. This generation process is thus not suited for *on the fly* generation. The structural consistency of a building hinges on more than just the enclosure. Without any other constraints the quality of the generated artifacts remains low. There is no higher-order structure imposed on the generated artifacts, aside from the enclosure constraint. From this perspective, the additive shape grammar represents only the local style of the examples. In every production step the newly produced shape only depends on the current shape taken into account. This is the same problem encountered when

generating with Markov models and n-grams [62, 18], where only the local style is transferred because of the Markov property, and the *r-similarity* constraint [8] and the adjacency constraint [44]. These constraints enforce similarity between local neighborhoods between the generated and example structures, with the additional effect of ensuring enclosure of a generated structure if the examples are enclosed. Instead of only relying on adjacency rules, an extension with a constraint that enforces similar local neighborhoods, could ensure enclosure during generation and limit the derivation of inconsistent rules that reduce the quality of the generated artifact.

Similar style Finally, we question if the generated buildings actually depict an artifact in the style of the original examples. A grammar induced from just a single example structure is limited in the style it describes. Only shapes and rules present in the example can make their way into the grammar. A different example built by a designer in the same style will most likely lead to a different grammar. Inducing a grammar from more examples with different configurations of style features will result in a more well-rounded description of the intended style. The fact that matching shapes share rules may overly generalize the style of an example. Note the example 3.11a, with generated structure 4.5b. This example may depict an artifact in a style of tower-like buildings, where other examples will have more rooms stacked on top of each other. Instead, the grammar depicts a very different style in the generated artifacts. This can not simply be fixed by inducing the grammar from more examples in the style, because the matching shapes share rules in a way that, in this case, will always allow the generated structure to become much more immense than a tower of cuboid rooms. A subset of the possible generated artifacts of this grammar will be the intended style, while the grammar will depict a more general style.

4.3 Grammar for Automatic Rule Derivation

We propose an extension of our methods to induce a more advanced grammar from the example structures. The grammar discussed in this section is designed with automatic derivation in mind, because the additive shape grammar defined in the previous section struggles with the generation of new structures. It takes a higher-order structure of the buildings into account and derives rules hierarchically from low to high detail. We explore the requirements, advantages and limitations of this new method and illustrate it with a simple prototype.

4.3.1 Introduction

Both the split grammar defined by Wonka et al. [82], as the CGA grammar defined by Müller et al. [49] are shape grammars designed with automatic derivation in mind. The split grammar approach removes design ideas out of the grammar and uses a separate control grammar to distribute the design ideas over the whole structure. A split grammar is sufficiently restricted to suit automatic derivation. Starting from

a large basic shape, split rules iteratively split shapes into smaller shapes until all shapes in the production are terminal. After every split a control grammar distributes design attributes in the basic shapes in order to keep the style over the entire artifact consistent, such as setting attributes exclusive to the first floor. The CGA grammar is an extension of the split grammar approach, allowing more complex rules and the generation of complex *mass models*. The rules are derived in a hierarchical order from low to high detail, ensuring a structural coherence of the generated artifacts not present in the additive shape grammar. Both grammars are able to reliably generate satisfactory artifacts from manually designed grammars. These grammars have been induced from examples to successfully generate new two-dimensional structures [40, 74, 50]. As far as we know, no example-based split or CGA grammar approaches have been discussed for three-dimensional structures.

Inspired by these shape grammar approaches for automatic generation of new structures, we discuss a method that derives rules in a hierarchical top-down manner, and constructs a new building starting from basic primitive shapes that are filled in with the actual shapes discussed in chapter 3. This extension will limit the grammar's reliance on matching shapes and improve the coherence, reliability and quality of the generated structures. We do not specify the entire method, but sketch the main ideas and the possible advantages and implement a basic prototype as an illustration of the method.

4.3.2 Method

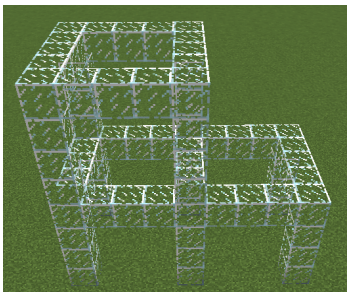
The additive shape grammar has difficulty generating structurally coherent artifacts because it is missing a notion of the higher-order structure in the example buildings. This higher-order structure [46] can be described in the form of a *mass model* [49], a building framework, or the combination of a set of primitive volumetric shapes, such as cuboids, cylinders and pyramids. Our basic example 3.1a exists out of a single cuboid, the example 3.11b consists of a cuboid and a pyramid for the slanted roof. The lighthouse in Figure 3.11e consists of a cylindrical shape and a number of smaller shapes for the stairs and the top of the tower. More complex example buildings will require more, smaller primitive shapes to form a *mass model* description for the examples. The primitive shapes in a building are composed in various ways, such as the two cuboids in Figure 3.11a placed on top of each other. If we can infer the primitive shapes and their relationships from the examples, as we did for style features, we can create new similar compositions of these primitive shapes. We infer the style features, or shapes, and their relationships inside these primitive shapes separately, following the method discussed in chapter 3, and finally use these to fill in the primitive shapes of the newly generated buildings. Instead of just finding style features and composing these into new structures we find the general structure of the buildings, the style features inside these higher-order features and compose these accordingly. The entire specification and implementation of this method will require a number of clearly defined procedures and decisions.

Primitive shapes First of all, we need an extension that is able to find the primitive shapes that make up an example structure. An algorithm that fits primitive shapes to the example can be optimized to find a small set of primitive shapes that cover the example in a similar fashion to the method discussed in chapter 3. The rules between these primitive shapes can be found from the examples. We must decide whether exclusively compositions found in the example structures are allowed, such as placing two cuboids on top of each other in example 3.11a, or additional compositions not present in the examples are also used during generation. If only relationships between the primitive shapes seen in the examples are allowed, larger examples or larger sets of examples should be provided, that give a clearer indication of the possibilities. Most examples in Figures 3.1 and 3.11 will be limited in the possible compositions of their primitive shapes. Alternatively, primitive shapes are connected through a smaller shape or style feature present in the example. If this feature is present in a primitive shape it acts as a *connection point* to another primitive shape. The ceiling of the lower cuboid in 3.11a could be the element that allows connection of the higher cuboid. In case the ceiling of the second cuboid matches the initial ceiling, another cuboid could be placed on top of it.

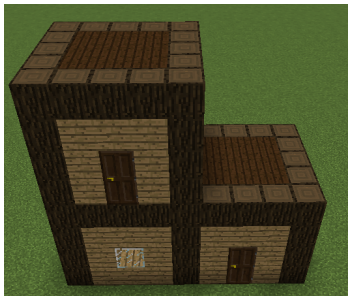
Split rules For every primitive shape that covers a part of the example, we perform the shape feature inference procedure, effectively finding the split rules from the primitive shape to smaller style features. The style features inside these new shapes can again be found, to create new split rules, where we consider the final features terminal. Iteratively finding the shapes present in a larger subset of shapes will in effect construct a split grammar [82].

If we consider just the split rules actually present in each primitive shape, every derivation of that primitive shape will result in the same component. In this case we could just as well just define primitive shapes as large three-dimensional shapes, as discussed in section 3.1.2, limited to certain basic shapes. We want to introduce an additional variance in the split rule derivations. This is a difficult problem that could rely on matching shapes and similar split rules between different primitive shapes.

Control grammar The control grammar that is used in unison with a split grammar [82] provides additional control over the coherence of the generated buildings. It can distribute certain attributes along just a single floor, for example allowing only exterior doors on the first floor. Although we believe the hierarchical derivation of rules from low to high detail will already provide much better generated artifacts than the additive shape grammar, the addition of a component similar to the control grammar could increase the quality even more. It would infer design ideas from the examples and distribute these over the newly generated artifacts. The more information that can be inferred about the style of the examples, the more accurate and reliable the generation process will become.



(a) A generated composition of cuboids.



(b) The primitive shapes filled in.

Figure 4.7: Illustrative prototype of the extended method for example 3.1a.

4.3.3 Limitations

This method will struggle with more complex example buildings, such as 3.1b, because they will be much more difficult to describe through primitive shapes. Not just one primitive shape will encapsulate, for example, the entire roof. The amount of primitive shapes will increase because of the many protruding blocks and details that need to be covered by their own basic shapes. As was an issue with the shapes in the additive shape grammar, many small shapes result in many different, potentially unnecessary, rules and thus lesser results. The more complex the description of the example through primitive shapes, the less these represent a higher-order structure of the example. In general it will be much more difficult to find a higher-order description of very complex buildings. A potential solution adds another layer of abstraction to the structure of the buildings, that covers larger components of the structures and is split into smaller primitive shapes.

This method remains interpretable, but the multiple layers of the split grammar intertwined with the distribution of attributes by the control grammar will complicate the method and the modification process.

4.3.4 Prototype Evaluation

We implemented a simple prototype to illustrate this extended method. Instead of inferring the primitive shapes and their compositions from the examples, we created a grammar specifically for example 3.1a. The example exists out of a single cuboid, which can be split into four walls and a ceiling. The terminal shapes were found with the shape inference method described in chapter 3 and correspond to the features in Figure 3.6a. These could still be split further to refine the lower detail style features.

Because the example is captured by a single cuboid shape without any relationships with other primitive shapes we allow duplicates of this cuboid to be placed next to and above each other. In the actual method we will induce the relationships between primitives and use these rules exclusively, or with a limited number of predefined rules. Starting from an arbitrary cuboid, we expand the rules in the grammar for primitive shapes. In the case of example 3.1a this places cuboids next to

Figure 4.8: Additional generated results for the prototype based on example 3.1a.

or on top of other cuboids restricted to a predefined parcel, as illustrated by Figure 4.7a. In Figure 4.7b these primitive shapes are filled in with terminal shapes. A single ceiling feature is allowed in the ceiling *slot*, so it is filled in with the same shape in every cuboid. The four walls present in the example are allowed in every wall *slot* in the primitive shape, if their neighboring shapes match the shapes in the example. This will ensure the local similarity between the examples and the generated artifacts is considered during generation.

A final extension would be to find a component similar to a control grammar that distributes the final terminal shapes over the entire generated structure. It could, for example, enforce that doors are only allowed on the first floor.

This illustrative prototype has ignored many crucial parts of the method and implementation issues that should be addressed in detail in potential further work. Nonetheless, the artifacts generated by this extended method guarantee a general higher-order structure in the buildings, removing the need to check for satisfaction of enclosure. This scheme could be used to generate new structures *on the fly*. Because the separate relationships between the final shapes are used to derive the final terminal shapes we also ensure a local lower-order similarity with the provided examples.

Chapter 5

Conclusion

This thesis explored inferring style features and shape grammars from three-dimensional voxel-based structures. This final chapter discusses the results and limitations of our work and proposes possible extensions as future work.

5.1 Conclusion

Procedural content generation and procedural modeling suffer from the issue that for every desired type of content a rule set, or algorithm, must be devised to generate the desired content. Potential solutions for this problem are example-based methods, such as PCGML [72], texture and model synthesis [21, 44] and IPM [4, 8, 66, 50]. These generate content or rules sets for a style of content from examples. Existing research on example-based methods for generating three-dimensional artifacts is limited and relies on highly similar systems and constraints [8, 44].

We proposed a method for inferring a set of style features, or shapes, from provided example structures. These shapes are then used to form a shape grammar that can be used to model and automatically generate new structures in the same style. Our method for finding a set of shapes relies on optimizing a function that determines the cost of a set of shapes. The cost function seeks a balance between simple shapes and a simple set. A limited local search [1] scheme optimizes the cost function and converges to a local optimum. The resulting shape sets are mostly satisfactory, but there is no guarantee that the obtained local optimum is good. We explored variations of various components of this algorithm that take other considerations into account and performed experiments to validate or refute their intended effects.

We discussed two grammar specifications and fully implemented the first. The additive shape grammar is a simple grammar that consists exclusively of additive rules. This shape grammar is able to recreate subsets and copies of the provided examples, and more varied new structures in a similar style if there are matching shapes present in the shape set. The variety in generated buildings relies completely on finding matching shapes in the shape inference step, and should be optimized to find matching shapes when using this grammar. The additive shape grammar is not

well suited for automatic generation, because the indefinite random derivation of rules is able to create enormously large structures without any concern for higher-order structure. Nonetheless, we define the enclosure constraint which allows for the generation of acceptable artifacts from simple examples. Besides the many limitations we have successfully inferred style features and grammars from limited examples capable of generating new structures in the same or similar style. This method can infer a style from multiple examples, combining the features and rules of all examples if they share a matching shape. It is interpretable and can be modified to change the generative space of the grammar by removing or adding rules and shapes.

The two primary shortcomings of the additive shape grammar for automatic generation are the lack of higher-order structure throughout the productions and the boundless derivation of rules. We discuss a potential extension to solve these issues inspired by split and CGA grammars [82, 49]. These are shape grammars designed with automatic derivation in mind, working hierarchically from low to high detail and reaching a terminating point in the derivation. Our extension requires a method for finding primitive shapes that cover large parts of the example structures and represent low detail higher-order structure of the examples. These primitive shapes can be composed and filled in with actual style features found through our original method. We illustrated the main ideas of this approach with a prototype built for a simple example, concluding that this method could ensure a higher-order structure, while still providing the local similarity with the examples. Crucial methods and details will need to be addressed in further work.

5.2 Real World Applications

Although the use of this method in real world application will require increased reliability and quality of the generated artifacts, the potential for example-based procedural methods is enormous. The entertainment industry, forefronted by the video game industry, relies increasingly on procedural methods to efficiently create content. Our example-based methods simplify the process of creating procedural rules and systems for multiple different building styles. These could be used to populate virtual models of cities existing out of many different styles of buildings [53, 30]. Aside from the entertainment industry, these could be used in the architectural field and in various simulations [82, 49].

5.3 Future Work

Despite the realization of our initial goals, our method remains limited. We propose a number of extensions and improvements to alleviate these limitations and incorporate additional functionality.

The style feature, or shape, inference method discussed in chapter 3 can be improved to more reliably return a better set of shapes. The simple hill-climbing algorithm converges to the first local optimum that it finds. Other more advanced local search methods could improve these results [1, 9] through backtracking executed

operations, restarts and iterative local search methods. Additionally, the cost function to be optimized could be improved or modified to encourage more specific shape set requirements.

The generation of artifacts through the additive shape grammar could be refined and improved by applying various constraints during and after generation. A refinement of the induction of rules could remove redundant and undesirable rules from the grammar, improving the results of random derivations. However, as discussed, the additive shape grammar is intrinsically not suited for automatic generation. We could apply a constraint that ensures the local neighborhoods in generated artifacts always match local neighborhoods in the examples, which has been addressed extensively in the work of Bokeloh et al. [8] and Merrel [43, 44]. Improvements to this method are more akin to patchwork than to fixing the underlying issues. We should instead focus on methods designed with these issues in mind.

We discussed an approach designed for automatic derivation, inspired by split [82] and CGA grammars [49]. We believe this method will strongly increase the reliability and quality of the artifacts generated in a similar style as the provided examples. It will ensure a higher-order structure in all generated artifacts, which will allow this method to be used *on the fly*, for example during gameplay. The concretization, implementation and evaluation of this method is the next step.

Instead of exclusively using style features present in the examples we could consider the usage of similar but slightly different shapes. Inferred shapes could be extended or reduced to form similar shapes of different dimensions. A grammar that uses these will not exactly describe the style of the examples, but will have an additional layer of variance in the generated artifacts. These possibilities highlight the important trade-off between the variety and expressivity of the generated artifacts and their resemblance to the example structures.

Our methods are currently limited to rigid voxel structures, while in most applications the three-dimensional structures are modeled with polygons. A final extension will allow these, or similar, techniques to be applied to polygonal examples.

5.4 Epilogue

Style inference and example-based procedural methods are an important development in the field of procedural generation. This thesis discussed the relatively unexplored problem of inducing a set of rules that are capable of generating similar structures from three-dimensional buildings, and serves as a stepping stone for further research in this field. Aside from inferring and imitating a style, algorithms that are able to learn new concepts and styles could form an important foundation for eventual computationally creative systems.

Appendices

Appendix A

Program Setup

This chapter describes how to set up and test the implementations of the methods discussed in this thesis.

A.1 MCEdit

MCEdit can be deployed by following the instructions at https://github.com/Podshot/MCEdit-Unified.

A.2 Filters

Our implementation is in the form of MCEdit filters that can be cloned from https://github.com/gillishermans/thesis_filters. These must be placed in the stock-filters folder within the root directory of the MCEdit repository. The filters can be applied by following these steps:

- start the MCEdit application as described at https://github.com/Podshot/MCEdit-Unified
- open a minecraft map (CTRL+O), such as the *example_world* map provided in the filter repository, by selecting the *level.dat* file in the map folder
- select the entire example structure or structures to be examined with the select tool
- apply the **infer_and_generate** filter with the filter tool to infer a set of shapes and a grammar and to immediately generate a new structure

A.3 Filter Parameters

The **infer_and_generate** filter has a number of parameters that are explained here:

- the hill climbing operation: 0 (only merge), 1 (only split), 2 (both)
- the shape specification: 0 (rectangular), 1 (planar), 2 (three-dimensional)
- the cost function: 0 (basic), 1 (multiple types cost increase), 2 (matching shapes discount)
- the α parameter: a value $\in \mathbb{R}$
- is overlap of blocks in multiple shapes allowed: true or false
- is the post processing split operation allowed: true or false
- the amount of rule derivations allowed during automatic generation: 0 (not generation), any other positive value $\in \mathbb{Z}$
- is the enclosure constraint enforced: true or false
- experimental split grammar option: true or false (no results guaranteed)
- visualization of overlapping generated shapes

Bibliography

[1] E. Aarts, E. H. Aarts, and J. K. Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.

[2] R. Adams and L. Bischof. Seeded region growing. *IEEE Transactions on pattern analysis and machine intelligence*, 16(6):641–647, 1994.

[3] T. Adams and Z. Adams. *Dwarf Fortress*. Bay 12 Games, 2006.

[4] D. G. Aliaga, P. A. Rosen, and D. R. Bekins. Style grammars for interactive visualization of architecture. *IEEE transactions on visualization and computer graphics*, 13(4):786–797, 2007.

[5] P. Ammanabrolu, W. Broniec, A. Mueller, J. Paul, and M. O. Riedl. Toward automated quest generation in text-adventure games. *arXiv preprint arXiv:1909.06283*, 2019.

[6] B. Beneš, O. Šťava, R. Měch, and G. Miller. Guided procedural modeling. In *Computer graphics forum*, volume 30, pages 325–334. Wiley Online Library, 2011.

[7] M. A. Boden. *The creative mind: Myths and mechanisms (second edition)*. Routledge, 2004.

[8] M. Bokeloh, M. Wand, and H.-P. Seidel. A connection between partial symmetry and inverse procedural modeling. In *ACM SIGGRAPH 2010 papers*, pages 1–10. 2010.

[9] I. BoussaïD, J. Lepagnot, and P. Siarry. A survey on optimization metaheuristics. *Information sciences*, 237:82–117, 2013.

[10] D. Braben and I. Bell. *Elite*. Acornsoft, Firebird and Imagineer, 1984.

[11] C. B. Browne. *Automatic generation and evaluation of recombination games*. PhD thesis, Queensland University of Technology, 2008.

[12] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, Sep. 1956.

[13] S. Colton. Creativity versus the perception of creativity in computational systems. In *AAAI spring symposium: creative intelligent systems*, volume 8, 2008.

[14] S. Colton, G. A. Wiggins, et al. Computational creativity: The final frontier? In *Ecai*, volume 12, pages 21–26. Montpelier, 2012.

[15] M. Cook and S. Colton. Multi-faceted evolution of simple arcade games. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 289–296. IEEE, 2011.

[16] S. Dahlskog and J. Togelius. Patterns and procedural content generation: revisiting mario in world 1 level 1. In *Proceedings of the First Workshop on Design Patterns in Games*, page 1. ACM, 2012.

[17] S. Dahlskog and J. Togelius. Procedural content generation using patterns as objectives. In *European Conference on the Applications of Evolutionary Computation*, pages 325–336. Springer, 2014.

[18] S. Dahlskog, J. Togelius, and M. J. Nelson. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*, pages 200–206. ACM, 2014.

[19] K. David Rio Vierra and all other contributors. *MCEdit*. 2010-2015.

[20] F. Doshi-Velez and B. Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[21] A. A. Efros and T. K. Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1033–1038. IEEE, 1999.

[22] A. Emilien, A. Bernhardt, A. Peytavie, M.-P. Cani, and E. Galin. Procedural generation of villages on arbitrary terrains. *The Visual Computer*, 28(6-8):809–818, 2012.

[23] B. Entertainment. *Diablo*. Blizzard Entertainment, 1996-2017.

[24] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[25] J. A. Hall, B. Williams, and C. J. Headleand. Artificial folklore for simulated religions. In *2017 International Conference on Cyberworlds (CW)*, pages 229–232, Sep. 2017.

[26] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1, 2013.

[27] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[28] Y. Hu, J. Dorsey, and H. Rushmeier. A novel framework for inverse procedural texture modeling. *ACM Transactions on Graphics (TOG)*, 38(6):1–14, 2019.

[29] R. Jain, A. Isaksen, C. Holmgård, and J. Togelius. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC Workshop on Computational Creativity and Games*, 2016.

[30] G. Kelly and H. McCabe. Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8–16, 2007.

[31] A. Khalifa, M. C. Green, D. Perez-Liebana, and J. Togelius. General video game rule generation. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 170–177. IEEE, 2017.

[32] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[33] J. Knutzen. *Generating climbing plants using l-systems*. Chalmers University of Technology, 2009.

[34] H. Koning and J. Eizenberg. The language of the prairie: Frank lloyd wright's prairie houses. *Environment and planning B: planning and design*, 8(3):295–323, 1981.

[35] B. Kybartas and C. Verbrugge. Analysis of regen as a graph-rewriting system for quest generation. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2):228–242, 2013.

[36] A. Liapis, G. N. Yannakakis, and J. Togelius. Computational game creativity. In *ICCC*, pages 46–53. Citeseer, 2014.

[37] A. Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

[38] M.-Y. Liu, O. Tuzel, S. Ramalingam, and R. Chellappa. Entropy rate superpixel segmentation. In *CVPR 2011*, pages 2097–2104. IEEE, 2011.

[39] A. A. Markov. Extension of the limit theorems of probability theory to a sum of variables connected in a chain. *Dynamic probabilistic systems*, 1:552–577, 1971.

[40] A. Martinovic and L. Van Gool. Bayesian grammar learning for inverse procedural modeling. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 201–208, 2013.

[41] M. Mateas and A. Stern. *Facade*. Procedural Arts, 2005.

[42] P. Merrell. Example-based model synthesis. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 105–112, 2007.

[43] P. Merrell and D. Manocha. Continuous model synthesis. In *ACM SIGGRAPH Asia 2008 papers*, pages 1–7. 2008.

[44] P. Merrell and D. Manocha. Model synthesis: A general procedural modeling algorithm. *IEEE transactions on visualization and computer graphics*, 17(6):715–728, 2010.

[45] K. E. Merrick, A. Isaacs, M. Barlow, and N. Gu. A shape grammar approach to computational creativity and procedural content generation in massively multiplayer online role playing games. *Entertainment Computing*, 4(2):115–130, 2013.

[46] N. J. Mitra, M. Wand, H. Zhang, D. Cohen-Or, V. Kim, and Q.-X. Huang. Structure-aware shape processing. In *ACM SIGGRAPH 2014 Courses*, pages 1–21. 2014.

[47] D. C. Moffat and M. Kelly. An investigation into people's bias against computational creativity in music composition. *Assessment*, 13(11), 2006.

[48] Mojang. *Minecraft.* Mojang, 2011.

[49] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, pages 614–623. 2006.

[50] P. Müller, G. Zeng, P. Wonka, and L. Van Gool. Image-based procedural modeling of facades. In *ACM Transactions on Graphics (TOG)*, volume 26, page 85. ACM, 2007.

[51] G. Nierhaus. *Algorithmic composition: paradigms of automated music generation.* Springer Science & Business Media, 2009.

[52] Nintendo. *Super Mario Bros.* Nintendo, 1985.

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

[54] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants.* Springer Science & Business Media, 1990.

[55] T. Roden and I. Parberry. From artistry to automation: A structured methodology for procedural content creation. In *International Conference on Entertainment Computing*, pages 151–156. Springer, 2004.

[56] L. Rokach and O. Maimon. Clustering methods. In *Data mining and knowledge discovery handbook*, pages 321–352. Springer, 2005.

[57] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural content generation in games*. Springer, 2016.

[58] A. Shamir. A survey on mesh segmentation techniques. In *Computer graphics forum*, volume 27, pages 1539–1556. Wiley Online Library, 2008.

[59] C. E. Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64, 1951.

[60] R. M. Smelik, T. Tutenel, R. Bidarra, and B. Benes. A survey on procedural modelling for virtual worlds. In *Computer Graphics Forum*, volume 33, pages 31–50. Wiley Online Library, 2014.

[61] A. R. Smith. Plants, fractals, and formal languages. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 1–10, New York, NY, USA, 1984. ACM.

[62] S. Snodgrass and S. Ontanón. Generating maps using markov chains. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[63] G. Software. *Borderlands*. 2K Games, 2009-2019.

[64] N. Sorenson and P. Pasquier. Towards a generic framework for automated video game level creation. In *European conference on the applications of evolutionary computation*, pages 131–140. Springer, 2010.

[65] F. Sportelli, G. Toto, and G. Vessio. A probabilistic grammar for procedural content generation. 07 2014.

[66] O. Šťava, B. Beneš, R. Měch, D. G. Aliaga, and P. Krištof. Inverse procedural modeling by automatic generation of l-systems. In *Computer Graphics Forum*, volume 29, pages 665–674. Wiley Online Library, 2010.

[67] G. Stiny. Introduction to shape and shape grammars. *Environment and planning B: planning and design*, 7(3):343–351, 1980.

[68] G. Stiny. Spatial relations and grammars. *Environment and Planning B: Planning and Design*, 9(1):113–114, 1982.

[69] G. Stiny and W. J. Mitchell. The palladian grammar. *Environment and planning B: Planning and design*, 5(1):5–18, 1978.

[70] A. Stolcke and S. Omohundro. Inducing probabilistic grammars by bayesian model merging. In *International Colloquium on Grammatical Inference*, pages 106–118. Springer, 1994.

[71] A. Summerville and M. Mateas. Super mario as a string: Platformer level generation via lstms. *arXiv preprint arXiv:1603.00930*, 2016.

[72] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[73] J. O. Talton, Y. Lou, S. Lesser, J. Duke, R. Měch, and V. Koltun. Metropolis procedural modeling. *ACM Transactions on Graphics (TOG)*, 30(2):1–14, 2011.

[74] O. Teboul, I. Kokkinos, L. Simon, P. Koutsourakis, and N. Paragios. Parsing facades with shape grammars and reinforcement learning. *IEEE transactions on pattern analysis and machine intelligence*, 35(7):1744–1756, 2012.

[75] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley. Procedural content generation: Goals, challenges and actionable steps. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[76] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2nd international workshop on procedural content generation in games*, page 3. ACM, 2011.

[77] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 111–118. IEEE, 2008.

[78] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[79] H. Toivonen and O. Gross. Data mining and machine learning in computational creativity. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 5(6):265–275, 2015.

[80] M. Toy, G. Wichman, K. Arnold, and J. Lane. *Rogue*. Epyx, 1980.

[81] L.-Y. Wei, S. Lefebvre, V. Kwatra, and G. Turk. State of the art in example-based texture synthesis. 2009.

[82] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. *Instant architecture*, volume 22. ACM, 2003.

[83] F. Wu, D.-M. Yan, W. Dong, X. Zhang, and P. Wonka. Inverse procedural modeling of facade layouts. *arXiv preprint arXiv:1308.0419*, 2013.

[84] G. N. Yannakakis and J. Togelius. *Artificial intelligence and games*, volume 2. Springer, 2018.