FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Eine Graphgrammatik zur Modellierung von 2D Objekten

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software und Information Engineering

eingereicht von

Viktor Pogrzebacz, BA
Matrikelnummer 01028937

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Mag. Martin Ilčík

Wien, 9. Juli 2019

Viktor Pogrzebacz	Martin Ilčík

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Graph Grammar for Modelling of 2D Shapes

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Viktor Pogrzebacz, BA
Registration Number 01028937

to the Faculty of Informatics

at the TU Wien

Advisor: Mag. Martin Ilčík

Vienna, 9th July, 2019

Viktor Pogrzebacz

Martin Ilčík

Erklärung zur Verfassung der Arbeit

Viktor Pogrzebacz, BA
Straußengasse 2-10/2/2, 1050 Wien

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

Wien, 9. Juli 2019

Viktor Pogrzebacz

Danksagung

Zuallererst will ich mich bei meinem Betreuer, Martin Ilčík, bedanken, der im Rahmen der Betreuung dieser Bachelorarbeit, welche ob meines Doppelstudiums mehr Zeit als geplant eingenommen hat, neben hilfreichen Anregungen auch große Geduld gezeigt hat. Insbesondere möchte ich Martins Einladung, den Inhalt dieser Arbeit als ein short-paper im Rahmen des CESCG[1] zu präsentieren, hervorheben. Meine Erfahrungen im Rahmen der CESCG haben mich davon überzeugt, dass ich in weiterer Folge eine akademische Karriere im Bereich der Informatik anstreben will. Des Weiteren will ich mich bei meinen Eltern bedanken, die mich während der Arbeit an diesem Text bedingungslos und mit größter Geduld unterstützt und zum Fertigstellen dieser Arbeit motiviert haben. Danke.

[1] Central European Seminar on Computer Graphics

Acknowledgements

I would first like to thank my advisor Martin Ilčík for his valuable advice and patience in overseeing this thesis, which has taken longer than expected to reach its completion, due to my parallel study. I wish to particularly express my gratitude for Martins invitation to present the content of this thesis in the form of a short-paper at the CESCG[2]. The experiences I have gathered throughout the process and during the conference have served to convince me that I am going to pursue an academic career in the field of computer science. Secondly I would like to give thanks to my parents, who, with great patience, unquestioningly supported me throughout the creation of this thesis, motivating me to actually finish my work. Thank you.

[2]Central European Seminar on Computer Graphics

Kurzfassung

Das Erstellen von Modellen für Computergraphik ist eine sehr arbeitsintensive Tätigkeit, was die Größe von Projekten in dem Bereich stark limitiert. Prozedurale Modellierung ist ein Forschungsfeld, welches versucht dieses Problem durch automatische Generierung von Modellen, in unterschiedlichen Variationen und mit unterschiedlichem Detailgrad, zu lösen. Innerhalb des Feldes der prozeduralen Modellierung gibt es unterschiedliche Techniken, welche sich entweder auf das Generieren von Pflanzen, wie z.B. L-Systeme, oder das Generieren von Gebäuden, wie z.B. Shape Grammatiken, spezialisieren. Diese Arbeit hat zum Ziel einen möglichen Weg zur Verbesserung dieser Situation aufzuzeigen, indem eine Grammatik beschrieben wird, welche zur prozeduralen Modellierung sowohl von organischen wie auch von künstlichen Objekten im 2D Raum geeignet ist. Es wird der gesamte Enstehungsprozess, von der Konzeption, über die Implementierung der Grammatik und unterstützender Software, bis hin zur Anwendung an ausgewählten Problemen, beschrieben. Eine Graph-Grammatik mit dem gleichen Ziel wurde bereits in Christiansen und Bærentzen [CB13] enigeführt, diese hatte aber eine andere Definition und gänzlich andere Charakteristika. Die vorgestellte Grammatik hat zum Ziel das Modellieren mit ihr möglichst einfach und intuitiv zu gestalten. Um die Vielseitigkeit der Grammatik darzustellen werden Beispielsproduktionen vorgestellt, welche eine Koch-Schneeflocke, zirkuläre und quadratische Muster, eine Gebäudefassade und einen Baum erstellen.

Abstract

The creation of models for computer graphics is a very work intensive task, which places severe limits on the size of projects. Procedural modelling is an ongoing field of research which aims to alleviate this pressure by automatically generating multiple differing variations of models at multiple levels of detail. Within the realm of procedural model generation, there are a number of techniques specializing in either modelling plants e.g. L-Systems or in modelling buildings e.g. shape grammars or other such specialization. The following paper aims to show a possibility of improving this situation, by describing the conception and implementation of a graph grammar and support software, suitable for procedural modelling of both artificial (e.g. buildings and furniture) and organic (e.g. trees and flowers) objects in 2D space. A graph grammar with such aims was previously introduced by Christiansen and Bærentzen [CB13], but with a different definition and different characteristics. This work aims specifically to make using the introduced graph grammar simple and improve intuitiveness. The proposed graph grammars versatility is displayed through example production definitions creating a Koch snowflake, circular and square patterns, a building façade schematic and a tree.

Contents

CHAPTER 1

Introduction

Modelling of realistic objects is a subject that is growing in importance, especially within the entertainment industry. Automated modelling techniques are a very promising approach to this problem, particularly because they offer a way of providing both detailed and varied models in a shortened time-frame [FYA10].

Most procedural modelling schemes specialize on creating a specific type of objects, e.g. plants or buildings. This work aims to devise a graph grammar which can be used for general purpose geometric modelling, while not being so complex as to reduce its potential user base to people with deep technical understanding of formal grammars.

First, an overview of different kinds of grammars used for automated modelling is given, and a short introduction to the formal aspects of graph grammars is presented. Following this, the research methodology applied in this work is presented in detail. Then, based upon the previous study of existing approaches, a formal definition of a new graph grammar for modelling is presented. This leads to a discussion of the implementation, technical details and finally a demonstration of the new graph grammars capabilities with a few selected example applications.

The implementation described in this work focuses on 2D space, so as to not exceed the scope of this paper. However, the graph grammar is defined in such a manner, that extending it to cover 3D space would be relatively easy.

Graph grammars boast applicability to a wide range of problem spaces. Regarding modelling of objects, they have the potential to allow one to reason directly about the 2D or 3D models rather than working on unrelated concepts which then need to be translated into meshes in a second step. This makes graph grammars well suited to applications in the field of generative modelling in the author's eyes.

The contributions this thesis makes are:

- The proposal of a single class of graph grammars, which is shown to be capable of representing a wide variety of modelling tasks.
- A proposal to improve the ease-of-use when defining productions of a graph grammar enriched with geometric information, by offering automatic calculation of new vertex locations, taking into account the relative positioning of elements between the left- and the right-hand-side of a production.

CHAPTER 2

Grammars for Procedural Modelling

The first part of this section gives an introduction to various grammars which have been successfully applied in procedural modelling. Based upon observations from this section, a new graph grammar for modelling will be defined and implemented in the later chapters. The last sub-section of this chapter starts with an introduction to the formalisms of graph grammars, before giving an overview of previous applications of graph grammars to procedural modelling. This serves to give the reader an overview of different approaches to defining graph grammars and their effects on the designs of productions.

The investigation of different approaches to procedural modelling based on grammars will focus on L-systems as described in Prusinkiewicz and Lindenmayer [PL96], CGA shape as defined in Müller et al. [Mül+06] and one recent work, which also aims to introduce a graph grammar for the purpose of modelling, called Generic Graph Grammar [CB13].

2.1 L-Systems

The term L-system designates a class of different string-rewriting grammars, which differentiate themselves from the formal grammars of the Chomsky hierarchy by using a parallel application of productions instead of a sequential application. Examples of different types of L-systems include deterministic, context-free L-systems (called D0L-systems), stochastic, context-free L-systems (called stochastic 0L-system), context-sensitive L-systems (called IL-systems), parametric L-systems and many more.

At the most basic level an L-system is a string rewriting system where one symbol is replaced with any number of symbols by a production. In Prusinkiewicz and Lindenmayer [PL96] it is shown, that while it is possible to model some plants with context-free

L-systems, a lot more expressiveness can be gained by using parametric L-systems and certain kinds of vegetation require context-sensitive, parametric L-systems. It is also noted, that stochastic L-systems are necessary if one wants to model multiple plants using the same L-system since the equality of the generated plants is otherwise very noticeable.

```
Axiom: X
Production rules:
X -> F[+X][-X]FX
F -> FF
```

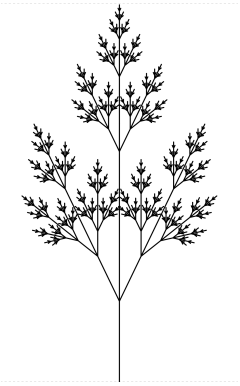

Figure 2.1: To the left are the rules of an L-system producing a simple 2D plant-like structure. To the right is the result of this L-system. From [PL96]

An example of an L-system for a simple 2D plant-like structure can be found in Figure 2.1. The left side shows the rules and the right side the result. To achieve said result, the set of rules is repeatedly run in parallel, usually for a relatively low number of steps (seven in this case) and then evaluated for display by a Logo-style turtle. The interpretations would be: `F` moves the turtle forward, `+` makes it turn left, `-` makes it turn right, `[` pushes the current position and direction onto the stack, and `]` pops the last saved position and direction from the stack. Each movement of the turtle draws a line over its path. Many L-systems for generation of flora use an extended set of functionality, such as a 3D space with operations to change pitch and yaw, changing the diameter, or changing the colour of line segment.

The above focus on plants should, however, not lead to the impression that L-systems are wholly unsuitable for other tasks, indeed Parish and Müller [PM01] used an L-System to great effect for modelling an entire city, including both the street-map and the buildings.

2.2 Set, Shape and Split Grammars

Shape grammars where first introduced by Stiny et al. [Sti+71], as a system where labelled lines or points where replaced by different lines and points. It was initially difficult to efficiently apply such productions automatically, but later works found that shape grammars could be simplified to set grammars [Mül+06]. This specific use of set grammars is what most recent articles on procedural modelling mean when they speak of shape grammars.

The modern shape grammars for the modelling of buildings and façades where developed since the turn of the millennium, using what is called a split approach. It was introduced in Wonka et al. [Won+03], inspired by L-Systems and the work done in Parish and Müller [PM01], and has since then been improved and extended by multiple publications. Examples include the addition of scopes in Müller et al. [Mül+06], extending the use of non-terminal shapes in Krecklau, Pavic, and Kobbelt [KPK10], allowing for a shape's scope to be any convex polyhedra in Thaller et al. [Tha+13], allowing conditions on geometric information within productions in Schwarz and Müller [SM15] and allowing for layers and SVG[1] code inside 2D shapes in Jesus, Coelho, and Sousa [JCS16].

The term split approach has come about because, for the procedural modelling of architecture, it has proven quite effective to start with a simple 2D or 3D object and then continuously split it apart to add more details. For example, a rectangle of appropriate dimensions could be used as the starting point for a façade or a polygon representing a building plot, for modelling a complete house. This starting geometry is then refined by splitting it apart, extruding or intruding specific shapes, and often by replacing a shape with pre-defined geometric information from an external 3D modelling software. Because of the significance of the split operation in this procedure, shape grammars using this approach are at times referred to as split grammars.

A typical example of a production in a shape grammar, taken from CGA shape [Mül+06], would be:

$$1: \quad fac(\mathtt{h}) : \mathtt{h} > 9 \rightsquigarrow floor(\mathtt{h}/3)\ floor(\mathtt{h}/3)\ floor(\mathtt{h}/3)$$

Where `1` is the ID of a rule taking a shape with the label *fac* for façade and an attribute `h` for height and applying only if `h` is greater than nine. If it is applied, it splits apart the shape *fac* into three shapes, each with the label *floor* and a height of one third of the original shape. To produce an interesting façade additional functionality is needed to control the productions. In [Mül+06] these are scopes, similar to L-system scopes, a function to split along an axis, to scale along an axis, to repeat a shape as long as there is space, and finally a function to split a scope into its components of lesser dimension, e.g. split a 3D cube into 2D faces. Its productions are assigned a priority and applied sequentially, in a way so that at first all rules of the highest priority are executed, then those of the next lower one and so forth. Each production can be assigned a probability, which determines how likely it is that the rule will be selected, and a condition of application. Conditions of productions are logical expressions, which can reference attributes of shapes as well as special functions like occlusion testing, which have to evaluate to true for a production to be applied.

A different, but somewhat related approach to shape grammars is GML[2] [HF11]. It is a stack-based imperative programming language, mirroring the syntax of PostScript,

[1] Scalable Vector Graphics
[2] Generative Modeling Language

which is well suited to implement typical context-free shape grammars. In addition to its shape grammar like functionality, it also supports shape representations using pcB-Reps, Convex Polyhedra and Volumetric Bitmaps.

2.3 Graph Grammars

The basic thought behind graph grammars is to apply the same scheme of productions used for strings in grammars from the Chomsky Hierarchy on the more complicated, but also more expressive, data-structure of a graph. It is a natural evolution of formal string based grammars into more expressive types.

Graph grammars were invented in the late 60's to solve pattern recognition, compiler construction and data type specification problems [Roz97]. Since then its use has spread to fields such as database design, logic programming, compiler construction, visual languages and also to applications outside the computer sciences such as chemistry, biology, meteorology or geology. With respect to the applicability of graph grammars to procedural generation of 2D or 3D models, there have been a handful of works making attempts in this direction, but the body of research is not comparable to L-Systems or shape grammars and there is still a lot of room for further study.

Since this paper focuses primarily on the specification and practical application of a graph grammar, rather than the theoretical properties of graph grammars in general, the following theoretical sections are by necessity quite short. For more background information, detailed and formal treatment of graph grammars, and information about their theoretical properties consult the standard work in this field, the „Handbook of Graph Grammars and Computing by Graph Transformation“ in three volumes [Roz97], [Ehr+99a], [Ehr+99b]. Volume one deals with the theoretical underpinnings, the second volume with applications and the third volume with implementation details of concurrency, parallelism and distribution of graph transformations. A detailed, specialized treatment of double push out algebraic graph grammars can be found in Ehrig et al. [Ehr+06]. A comparison between the double and the single push out approaches to algebraic graph grammars can be found in [Par93].

2.3.1 Simple Example of a Graph Grammars Derivation

To introduce a reader unfamiliar with graph grammars to the topic, a very simple example without any formal definitions is presented. This should improve understanding of the following discussion of different kinds of graph grammars, their definitions and their categorisations, and provides context for the formal underpinnings of graph grammars provided later on.

One very basic graph grammar would for example take a node and replace it with two nodes and an edge between them, while keeping existing connections intact. A sample production of this kind and the two possible results of applying such a production three times in a row are shown in Figure 2.2.

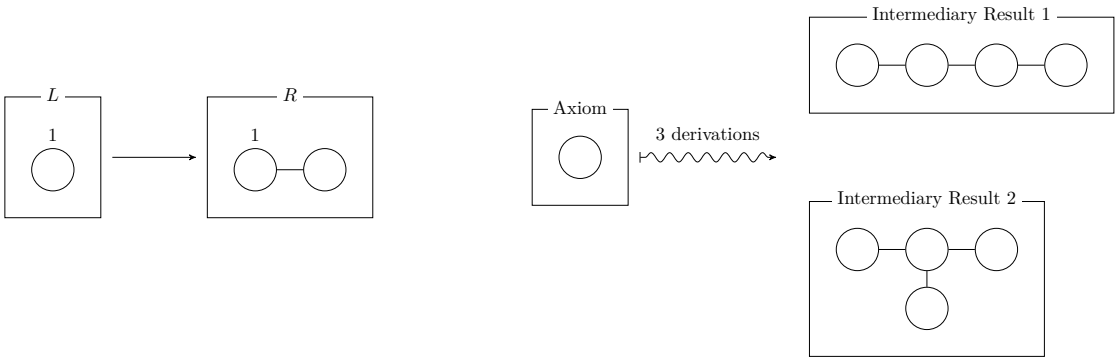

Figure 2.2: A simple example of a graph grammar. To the left is the production which simply adds a new node and edge to an existing node. To the right is the axiom graph and the two possible intemediary results after three derivations.

When applying a production to a graph, called the host graph H, the first step is finding a part of H which matches the left hand side L of the production. In the presented case this is very simple: Any node of H is a possible match for our rule. Secondly, a single match must be selected from all possible matches for L in H. When applying our single production to the axiom there is only one possibility; on the second step of our derivation sequence there are two possible matches, but they each lead to the same resulting graph; finally, on the third application, there are three possible matches with the two possible results shown on the right of Figure 2.2. The usual approach of choosing a match out of all possible matches is to choose one at random, but other solutions exist and have their place.

When applying a production to a match, we need to have some form of correspondence between elements of the left hand side L and elements on the right hand side R. If we had no means of creating such a correspondence, then no newly added elements could be connected to existing elements and all results would be disjoint. As this would put a severe limit on the expressiveness of a grammar, it is usual to have some means of connecting the newly added elements to the existing graph, for example by using *gluing* or *embedding*, which will be discussed in greater detail later on. For this simple example, I decided to uniquely map one node on the left hand side (marked by 1) to one node on the right hand side (also marked by 1), and to have all edges connected to the matching node transferred over to the corresponding node on the right hand side of the production. This approach is an example of *gluing*.

Throughout this example derivation we have come across a number of places where graph grammars can make divergent choices and differentiate themselves in their complexity, their expressiveness and their properties. Firstly, the definition of what kind of graphs are supported: directed graphs, undirected graphs, graphs with hyperedges, labeled graphs, typed graphs, attributed graphs, etc.; there are many possibilities which have been explored. Secondly, restrictions on the form of the left hand side of a production:

The usual options are allowing only nodes, only edges or allowing unrestricted graphs, but depending on the use-case other restrictions might be of use. Third, the process by which the new additions are added to an existing graph, which mainly separates into the *gluing* and the *embedding* approaches. Fourth, whether productions are applied in parallel or in sequence. And, last but not least, a host of smaller decisions, which often serve to make the grammar easier to work with in specific situations, such as controlling the order of application of productions, allowing for special conditional expressions on graph elements, allowing for production wide conditionals, etc.

In addition to the above, there are other characteristics of graph grammars, which are of interest during graph theoretical study of these grammars, and can be used to differentiate between them, such as *confluence*. As far as applications to procedural modelling are concerned, however, these qualities are of less import and therefore left out so as not to breach the scope of this work.

2.3.2 Overview and Categorisation of Graph Grammars

The first work published on graph grammars was Pfaltz and Rosenfeld [PR69], which does not formally define the grammar, instead describing productions in informal sentences. The first formal definition of a graph grammar was Schneider [Sch70], defining the grammar to be a tuple of a node label alphabet, a terminal node label alphabet, a finite set of productions and an axiom graph. This basic structure has essentially remained the same to this day, with differences between grammars mostly focusing on the definition of a production.

From this starting point, a wide variety of different types of graph grammars where described and used. A detailed treatment of the early history of graph grammars can be found in Nagl [Nag79].

Many graph grammars work on labeled graphs, where every element of the graph is given a label, allowing productions to query those labels in their matching and connecting instructions. A typical example for how such a graph would be defined comes from Engelfriet and Rozenberg [ER97], where a graph over the alphabet of node labels Σ and the alphabet of edge labels Γ is defined as a tuple $H = (V, E, \lambda)$, with V being a finite set of nodes, E being a set of tuples $\{(v, \gamma, w) | v, w \in V, v \neq w, \gamma \in \Gamma\}$ describing the edges of the graph and λ being the node labeling function assigning to each $v \in V$ a $\sigma \in \Sigma$. Of course there exist other possible definitions of graphs, such as graphs only labeling nodes, but not edges, as in Pfaltz and Rosenfeld [PR69], or attributed graphs, where, in addition to being labelled, graph elements can also have an arbitrary number of attributes assigned to them as in Rudolf [Rud97].

The most common general categorization of graph grammars is along the limitations placed on the left hand side of a production. Node replacement grammars limit the left hand side to only contain a single node, which is replaced with a new graph on the right hand side of the production during application. Edge replacement grammars allow only for a single edge on the left hand side, and graph replacement grammars allow one to

replace arbitrary graphs on the left hand side with arbitrary graphs on the right hand side.

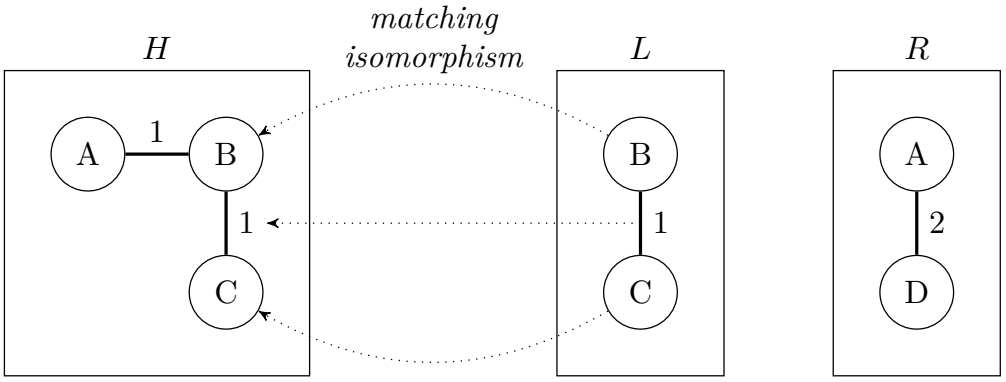

Figure 2.3: Schema of matching a production p to a hostgraph H.

A second common characterisation of graph grammars is based on the mechanism used to connect the right hand side of a production with the existing graph the production is applied to. Formally, when you apply a production p with the left hand side L and the right hand side R to a host graph H, the first step is to find a match H^L of L in H, a graph isomorphism from L to a subgraph of H as seen in Figure 2.3. Then those elements of H which are part of the isomorphism are deleted, leaving us with $H^- = H \setminus H^L$; in this case the node with the Label A and the dangling edge with the label 1.

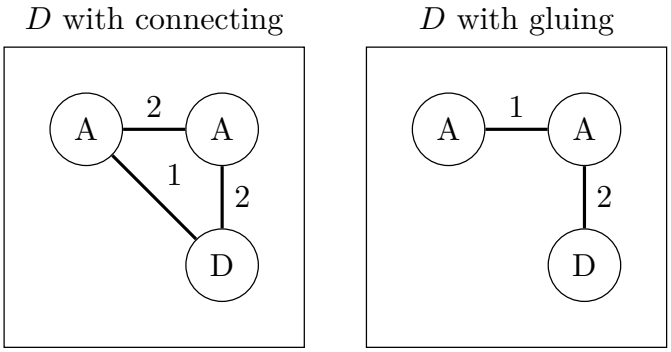

Figure 2.4: Result of applying p to H: on the left with the embedding instructions $\{(1, A) \mapsto \{(1, D), (2, A)\}\}$, on the right with gluing. Example adapted from Nagl [Nag79].

The next step is where the *connecting* approach, also called *embedding* approach, differs from the *gluing* approach. The *connecting* approach completely disallows dangling edges, so edges which were going from H^- to H^L, are also deleted in H^-, the result graph $D = H^- \cup R$ is formed and then new edges between H^- and R are formed based on instructions specified in what is called the embedding E. An example of a possible set

embedding instructions would be: $\{(1, A) \mapsto \{(1, D), (2, A)\}\}$. The first part of the rule decides if it is applied and to which elements of H^- new connections (if any) are made. So $(1, A)$ specifies, that new edges are connected on one side to nodes with the label A, which had an edge going to any element of H^L. The right side of the rule states to which nodes in R new edges are connected, and what label the new edges will have. In this case any nodes with the label D will be connected with an edge labelled 1 and any node labelled A will be connected with an edge labelled 2. The resulting graph of such an application can be seen on the left in Figure 2.4.

When applying the *gluing* approach some dangling edges are not deleted but effectively reconnected from H^L, the parts of H mapped to L, to elements of R according to a mapping between L and R specified within the production. Thus, a production takes on the form $p = (L, K, R)$ where L and R are as described above and K is the common interface between L and R, the intersection between the two graphs $K = L \cap R$. In this case $H^- = H \setminus (L \setminus K)$ and then $D = H^- \cup R$ so that the connections between elements in K and elements in R are retained. One can essentially imagine removing all of L from H adding all of R to it and then gluing all the dangling edges to the appropriate nodes of R, hence the name. As an example, imagine that the node with label B in L and the node with label A in R where the same (e.g. had the same unique node id) and therefore part of K. The resulting derivation can be seen on the right of Figure 2.4. It is also possible to define additional embedding relations for the gluing approach, which would be applied after the procedure described above. Of course in a practical implementation the details might be handled differently, such that the common interface might actually be a partial mapping $K : L \mapsto R$ or the dangling edges may be deleted first only to be re-created later on.

The *connecting* approach allows one to create new edges from and to nodes not matched within the left hand side of the production, whereas the *gluing* approach without extensions only allows for the creation of new edges between new elements or elements part of the left hand side. This explains why the connection or embedding approach is commonly used with node replacement grammars, where you cannot have multiple elements on the left hand side. Therefore, this approach significantly increases the expressiveness of the grammar. The gluing approach on the other hand is these days heavily associated with graph replacement grammars, since arbitrary graphs are allowed on the left hand side of a production and therefore the slightly more complicated definitions of embeddings would often be unnecessary. It is however instructive to note that there is no law demanding things be this way. After all the first formal graph grammar, specified in Schneider [Sch70], was a graph replacement grammar using embeddings and the simple example of a derivation given in the preceding section was a production of a node replacement grammar using gluing as the author considers its explanation to be simpler.

The *connecting* or *embedding* approach and the *gluing* approach also have a different set of terms used to describe them, namely the *algorithmic* or *set theoretic* approach and the *algebraic* approach respectively, though Engelfriet and Rozenberg [ER97] call this choice „unfortunate“. The term *algebraic*, in particular, is used quite commonly to refer

not just to a grammar using *gluing*, but to any graph grammar allowing arbitrary graphs on the left hand side of productions. This practice started with the first paper formally introducing arbitrary graph replacement grammars with gluing, „Graph-grammars: an algebraic approach“ Ehrig, Pfender, and Schneider [EPS73], and has continued since e.g. [Cor+97], [Ehr+06]. Following this tradition, in this work the terms graph replacement grammar and *algebraic* (approach) graph grammar are understood to be synonymous.

2.3.3 Algebraic or Graph Replacement Grammars

The following section takes a deeper look into the workings of algebraic graph grammars, because these grammars are of particular interest when it comes to the subject of procedural generation of geometric models. Allowing arbitrary graphs on the left hand side of productions not only gives the grammar a high degree of expressiveness, but also allows for a certain degree of geometric intuition to come into play when defining such productions. The information and examples relayed in this section stem primarily from Corradini et al. [Cor+97] and Ehrig et al. [Ehr+06].

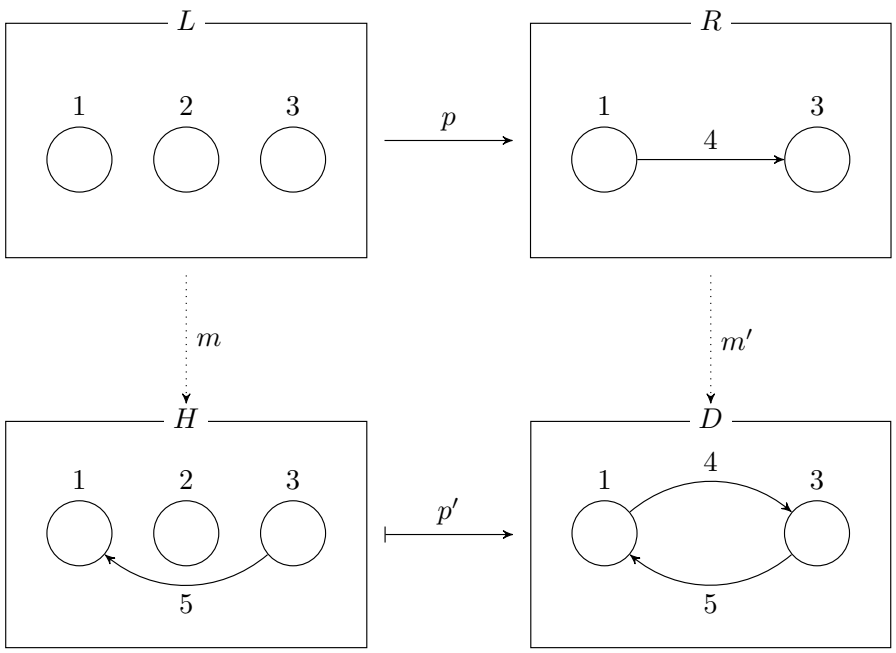

Figure 2.5: Example of a derivation step in an algebraic approach graph grammar. The node 2 is deleted and the edge 4 is added while transfroming H to the result D. m is the match from the left-hand-side to the host graph, a graph homomorphism, m' is the corresponding comatch, a homomorphism from the right-hand-side to the result. The production p defines the correspondence of elements between L and R, with p' doing the same between H and D. Adapted from [Cor+97].

In *algebraic* graph grammars, a production is defined as $p : L \to R$, with L and R being

called the left-hand side and right-hand side of the production. Given a host graph H, a match $m : L \mapsto H$ is, in the general case, a graph homomorphism mapping all edges and vertices from L to H. Applying a production p means deleting from H all elements which exist in L but not in R, adding all elements that exist in R but not in L and leaving all elements that exist in both L and R untouched. A visualization of this process can be found in Figure 2.5. The upper left graph represents the left-hand side L of the production p, the upper right the right-hand side R, the lower left graph is the original graph H and the right hand side is the resulting graph D.

When allowing matches to be homomorphisms rather than isomorphisms, there are a number of difficulties that can crop up if, for example, two nodes in L are matched to the same node in H. If exactly one of these two nodes from L is present in R, then it is unclear what should be done: Should the node in H be deleted or should it be left as part of H? Another such situation occurs when a production p deletes a node which is connected by an edge to another node in H. This would leave a dangling edge in H after applying p. The main difference between the *double push out* (DPO) and the *single push out* (SPO) approaches lies in how they deal with such special cases. Double push out disallows the application of the production in such problematic circumstances while single push out allows them and resolves the uncertainty by prioritizing deletion [Cor+97].

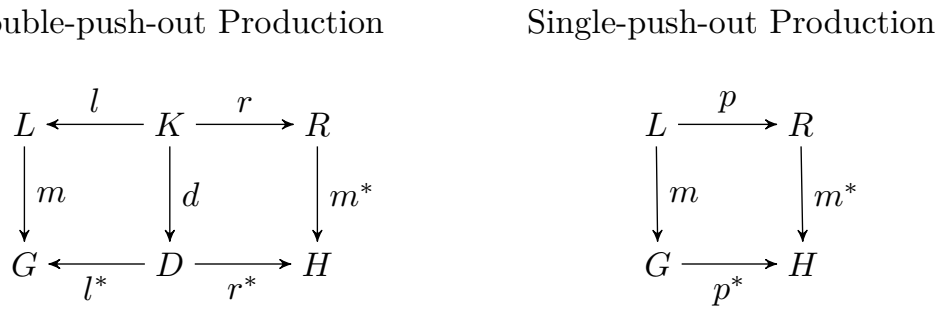

Figure 2.6: Schema of a production in a DPO graph grammar to the left and in a SPO graph grammar to the right. Adapted from [Cor+97].

In greater detail, the DPO approach defines a production as $p : L \xleftarrow{l} K \xrightarrow{r} R$, with L and R being total graph homomorphism from the common graph K called an *interface graph*. As seen on the right of Figure 2.6, L is matched to G and then a so called *context graph* D is created by deleting from G all elements that exist in L but not in K. This operation could be described as inverse gluing. In the next step, the result graph H is obtained from D by adding all elements that exist in R but not in K. In addition, the match m must satisfy two constraints, together called the *gluing conditions*. First, if a vertex v is deleted from G then the deletion of all edges connecting to v must also be ordered. This is called the *dangling condition*. Second, the *identification condition* states that any element that is to be deleted from G may only have one pre-image in L.

The SPO approach defines a production as $p : L \xrightarrow{r} R$, with r being a partial graph homomorphism as seen on the right side of Figure 2.6. There is no gluing condition; the match m can be any graph homomorphism. If a production specifies both the deletion and the preservation of an element then the deletion takes precedence. If the deletion of a vertex leaves behind a dangling edge the edge is deleted as well.

2.3.4 Generic Graph Grammar

Generic Graph Grammar (G^3), as defined in Christiansen and Bærentzen [CB13], uses a directed cyclic graph consisting of nodes, edges and faces, all without labels, called primitives to represent its objects. Productions are applied in parallel to all primitives of one kind (all edges, all nodes or all faces), making it a special combination of node replacement and edge replacement as well as algebraic approach graph grammars, where the left hand side of a production is limited to graphs that represent a single face. Every production is given a tuple of parameters defining the radius, direction and length of the produced primitive, it is also possible to specify variations for each of those values. Further it is possible to assign conditions, which are Boolean expressions querying values such as a random number, the age of the primitive, the number of neighbouring edges, the position of a primitive, etc., to each production. The last part of G^3 are the commands, which are used to define the outcome of a production. Available commands are *create edge*, *create face*, *grow face*, *split edge* and *split face*.

G^3's most notable point is that it does not use labels on its primitives which is a very unusual approach to graph grammars. As is noted in Fahmy and Blostein [FB92] graph grammars usually use labelled nodes and all grammars defined in Ehrig, Pfender, and Schneider [EPS73] and Rozenberg [Roz97] follow this method. Another notable characteristic of G^3 is its focus on atomar parts of a 3D mesh rather then using some sort of abstraction between the graph and the finished model. In comparison to other grammars analysed above, G^3 also severely restricts the number of attributes or parameters that can be added to a primitive.

The lack of labels make it difficult to select a particular part of a graph for modification by a production. Additionally, the limited options for left hand sides of productions make it impossible to match specific relations between elements of a graph. For these reasons the author considers the approach taken in G^3 somewhat unsatisfactory and sought a different solution.

What works well in this grammar is having productions work directly on 3D primitives, without requiring a layer of abstraction between them. This makes it relatively easy to visualize and predict what a production may change. Therefore, such a design direction was included in the grammar presented in this paper.

CHAPTER 3

Methodology

The aim of this work was to define and implement a graph grammar suitable for general purpose procedural modelling of 2D shapes, more specifically of both natural and artificial objects. Additionally, the definition of the grammar should be easily extensible to 3D models, as the limitation to 2D was only made so as not to exceed the scope of this work.

In order to gain an understanding of the problem space, a review of literature on successful procedural modelling grammars was undertaken. Based on this review, a list of features a grammar for procedural modelling would need was identified. The main requirements where a good access to randomness, labels to allow for control of productions and a system to allow the interpretation of productions in a geometric context.

In addition to these requirements, a set of example modelling tasks to test the suitability of the newly developed grammar was devised. As the goal of the new grammar was to have a single system suitable for a wide variety of tasks, specialities of both L-systems and of shape grammars where considered, with the limitation that they had to be possible in 2D space. The tasks chosen where modelling of a Koch Snowflake and foliage, domains where L-systems are very successful, a modelling of a building façade, a strength of shape grammars, and the creation of both circular and rectangular patterns for added diversity.

On this basis a first iteration of the formal definition of the new grammar was created and afterwards implemented in Python. The implementation made apparent certain weaknesses in the original definition of the grammar, which where then fixed in the formal definition. The same process of iterative refinement of the grammar was then applied during the implementation of the example modelling tasks.

While testing the first version of the implementation it became apparent that some form of visualisation for the productions was necessary. At that point a visualisation inspired by the typical diagrams for graph grammars in literature and the user interface of typical

3D modelling tools was developed. However, since the user interface was not the focus of this work, its development was stopped once a functionality adequate for the use case was achieved.

CHAPTER 4

Formal Definition of the Graph Grammar

grammar & axiom graph **terminal state** **graph**

The *grammar* is a tuple of the form (A, P), where A is the *axiom graph* or starting graph and P is a set of productions. A *terminal state* is implicitly encoded in a set of productions: If there is no production which can be applied to a graph then the graph is said to be in terminal form for this particular set P of productions. All *graphs* in the grammar are attributed and by default undirected.

production

At its most basic, the *productions* of the grammar are of the form (M, p, D) where M called the mother graph is the left-hand-side of the production; D, called the daughter graph, is the right-hand-side of the production; and p, the partial graph morphism, is a set of correspondences, a mapping from elements of M to elements of D describing those elements which remain unchanged during application of the production.

partial graph morphism

The *partial graph morphism* p is a morphism of some subgraph of M to D, i.e. $S \subset M$, $p : S \to D$. It is essentially a mapping of graph elements (e.g. nodes, edges, faces or volumes) between the mother graph M and the daughter graph D. In addition to the usual restrictions on a partial graph morphism found in literature, two constraints apply in the presented grammar:

- The preimage of an element of D may only contain zero or one element, which is to say that an element of D may be mapped to by at most one element of M.
- A graph element may only be mapped to another graph element of the same type, which is to say a node may only be mapped to a node, an edge to an edge and so on.

Figure 4.1 provides an example for a production and the partial graph morphism p defined between M and D, indicated by the dotted arrows. The two nodes and one

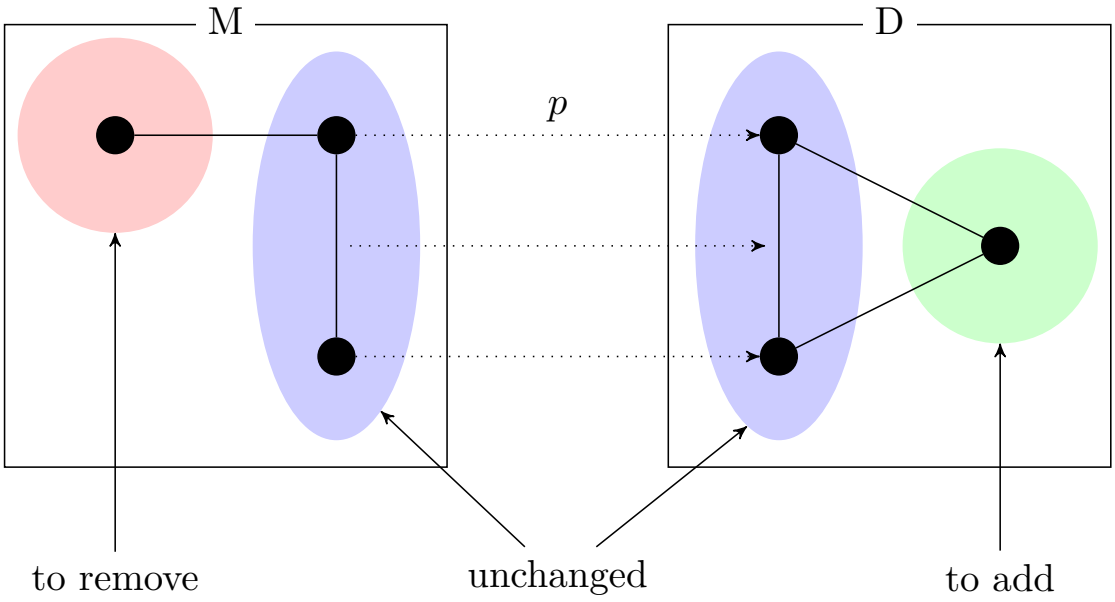

Figure 4.1: Example of a production definition. The dotted arrows going from M to D represent the partial graph morphism p. Elements which will be added are circled in red, elements to be added in green and elements kept unchanded in blue.

edge which are matched by p, highlighted in blue, will be kept unchanged during the application of this production. The node of M highlighted in red and the edge connecting to it are not part of p. This means they will be deleted when applying the production. The node of D highlighted in green and the two edges connected to it are also not part of p, but since they are part of D they will be the elements added during production application.

A *derivation step*, or an application of a production onto a hostgraph H, works by first finding a subgraph isomorphism $m : M \to H$. In the next step, for all elements of M not part of the domain of the partial graph morphism $dom(p)$, their counterparts in H are deleted: $R^* = H \setminus \{m(e) | e \in M \wedge e \notin dom(p)\}$. Afterwards, all new elements, those elements in D which are not part of the codomain $codom(p)$, are added to H and connected according to p, giving the result $R = R^* \cup \{n | n \in D \wedge n \notin codom(p)\}$. This process is outlined in Figure 4.2, which applies the production from Figure 4.1 to a graph containing a square. At first a match from M to H is sought. One such possible match m is indicated by the dashed arrows. Then one node and one edge is removed from H, and two new edges as well as one new node are added, as described above. The two newly added edges are connected to the existing elements of H by looking at the elements matched by the partial graph morphism p. For this purpose, elements $d \in D \wedge d \in codom(p)$ are equated to their equivalent in H: $m(p^{-1}(d))$. Newly added elements connecting to such an element d in D are reconnected to $m(p^{-1}(d))$. The partial graph morphism p essentially defines which elements of H are deleted, which elements

of D are merged together with elements of H, and which elements are added as new additions.

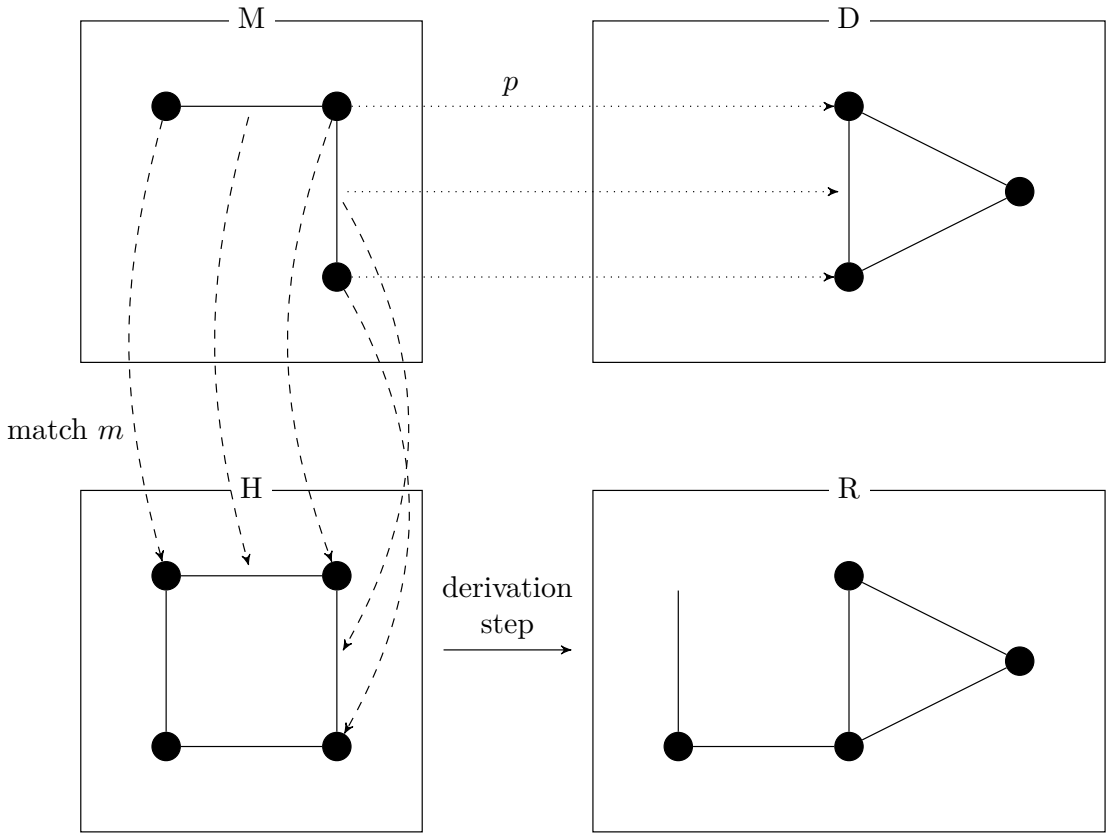

Figure 4.2: An example application of the rule from Figure 4.1. The dashed lines show the match found between M and H. R shows the result graph of applying the production.

A *derivation* is a sequence of derivation steps, the first one applied onto the axiom graph A, the following ones applied to the non-terminal graphs resulting from the previous derivation step. A derivation ends and produces a terminal graph when none of the productions in P can be applied, i.e. when there is no production $p \in P$ for whose right-hand side M exists a partial isomorphism from M to the host graph H. This definition allows for the creation of infinite derivations. That is not seen as an issue since, in practical applications, the number of derivation steps will always be bound by factors such as computer memory or time constraints. **derivation**

The above definition marks the introduced grammar as a graph replacement graph grammar using the gluing approach to embedding, also called the algebraic approach in literature. Within the algebraic graph grammars it is a single push-out approach graph grammar, using a mapping between the left and the right hand sides of the produc-

tion, rather than a common interface graph. It also allows for the deletion step in the derivation to leave dangling edges, which is a defining characteristic of single push-out grammars [Ehr+06].

The implicit definition of a terminal form was chosen out of consideration for practicality: A production in this graph grammar can be of such complicated form that any exhaustive listing of properties is unrealistic, in a practical context, and would add unnecessary tedium from a user interaction perspective.

4.1 Additional Extensions

To increase expressiveness and ease-of-use as a modelling tool, the grammar specified above is extended in a number of ways, detailed in this section.

Priority of productions. The order in which productions are applied can be controlled by setting a priority for each production. Those productions with the lowest priority value are executed first. If there are multiple productions with the same priority, then the order of application between them is random.

Control of end of generation. In addition to defining a grammar in such a way that it will eventually enter a terminal state (a state where no production can be applied), it is possible to define a maximum number of derivation steps to execute. This can be done globally, meaning the entire derivation will be stopped after x steps, or on a per-priority level. These two modes can be combined. For example, consider a grammar with two priorities of productions: 0 and 1. For priority 0 a limit of 100 production applications is defined. In addition, a global limit of 500 production steps is set. Then first, productions of priority 0 will be applied at most 100 times, following which productions of priority one will be applied up to a limit of 500 total applications. But if the productions of priority 1 make changes which lead to a terminal graph with regards to priority 1 productions in, say, 200 steps, then the derivation will stop, even if productions of priority 0 could still be applied. Examples of grammars making use of this feature to control their derivations are the Koch Snowflake from Section 7.1 and the tree models of Section 7.3.

Multiple D graphs for one M. To allow for control of the randomness when choosing between multiple derivations with the same left-hand side M, the productions take the form $(M, \{(p, D, w)\})$. M, p and D are as defined above and w stands for the weight this particular daughter graph has, when selecting which one will actually be applied amongst all possibilities. Thus, a mother graph has a set of possible daughter graphs, each with their own partial graph morphism p. If n is the number of possible daughter graphs of a given production, then the likelihood of a particular daughter graph D_i $0 \leq i \leq n$ being chosen is $w_i / \sum_{j=0}^{n} w_j$.

Multiple matches of the same production. By default, if a mother graph M has multiple matches in a hostgraph H, one of the matches is chosen at random. It is possible to change this on a per-production level, by setting an option to choose the match with the oldest average element age. The age of an element referring to the number of derivation steps the element has been part of the graph for.

Attributes of graph elements. Any graph element can contain an arbitrary amount of attributes, which are tuples of $(name, value)$ where $name$ is a string identifying the attribute and $value$ can be any arbitrary value.

Conditional Productions. When a production is matched against a hostgraph, in addition to finding a matching isomorphism m from M to H, it is also possible to define matching conditions for any element $e \in M$ on a per-element basis. These matching conditions take the form of a function with access to all the attributes of $m(e)$, the elements potential match in the hostgraph H. If the all matching conditions in the form of functions return true, then the match is accepted. If any of the functions returns false, the potential match is discarded and another isomorphism is tested.

Calculation of new attribute values. In order to support the calculation of new values for the attributes of graph elements, the value fields of attributes in the daughter graph D can contain calculation instructions rather fixed values. These calculation instructions can have access to any attribute values of any element matched in H, i.e. to any element of the set $\{m(e)|e \in M\}$.

Wildcard nodes in M. Edges in the mother graph M of a production may be connected to wildcard nodes, which can be matched to any node in the host graph and are left completely unchanged by a production application. This is merely syntactic sugar simplifying the definition of productions; it has no effect on the expressiveness of the grammar.

Saving vertex coordinates. To support a geometric interpretation of productions, each vertex saves a x- and a y-coordinate in its attributes. These attributes function like normal attributes: They can be queried in the mother graph and calculated in the daughter graph, but, if no calculation function is supplied in the daughter graph, the new values are calculated automatically. When using the grammar to model structural relations these attributes can be ignored completely, but, when interpreting them as replacements on geometric shapes, having access to the x/y-coordinates is very helpful.

Matching spatial relationships. To further support a geometric interpretation of productions, it is possible to set an option on a per-production basis which requires that any potential matches for the mother graph M in the host graph H respect the total ordering which is defined by the elements position on the x and y axes. This option has proven itself to be very helpful in providing intuitive results for productions.

Optional directed Edges. Edges can optionally be interpreted as being directed. This allows for additional expressiveness for some productions when used in a geometric context. An example of such a grammar is the Koch Snowflake of Section 7.1.

This set-up allows for defining a graph grammar which not only has a high level of expressiveness but also affords intuitive interaction with the system.

4.2 Sources of Randomness in the Grammar

Since randomness is an important source of expressiveness for the purpose of procedural generation of 2D models, this sub-section will give a short summary on the different means by which variation between results can be achieved in the proposed grammar.

There are two types of variation which can be differentiated. Structural variations, which change the content and/or structure of the graph and parametric variations, which are changes in the attributes of graph elements. The grammar offers a single way of adding variations in the attributes of elements, namely by evaluating an arbitrary python expression with access to the random library. In practice this offers a sufficient degree of freedom for attributes, so that no additional means of adding randomness were necessary.

As for structural variations, there are multiple ways by which it can be achieved in the proposed grammar. If there are multiple productions with the same priority, one of the productions is chosen at random. If all these productions make changes to the graph, which inhibit other productions of the same priority from being applied to the graph, then this can be used to produce structural variation. In addition to this method, it is possible to define a single mother graph with multiple daughters, each of which has a weight attached. When such a production is chosen for application and the mother graph successfully matched against the host graph, one of the daughter graphs is chosen at random, with the weighting providing more control to the user. Lastly, it should be noted that parametric variations can be used to lead to structural variations, in consecutive derivation steps, by using the value of a randomly calculated attribute as an application condition in multiple productions of lower priority.

CHAPTER 5

Implementation

Regarding the practical implementation of this graph grammar, there are two distinct parts, loosely coupled with each other. One side is the graph management and calculating functions in the form of a library, and the other side is the GUI which is used to work with the grammar in day to day operations.

5.1 Matching Algorithm

Finding subgraph isomorphisms of one graph within another is an NP-complete problem [GJ79]. Therefore any algorithm, trying to speed this problem up, will try to prune the search space early for common cases or employ heuristics, if finding all possible matches is not a necessity.

In this particular application a simple and direct implementation of the matching algorithm, testing an element in the mother graph of a production against every element of the host graph and then looping on that was implemented; see Algorithm 5.1 for details. In practical application this turned out to have acceptable runtimes for smaller mother graphs of about five elements matching against moderate host graphs of about 1000 elements.

Further improvement, such as using a specialised graph matching algorithm like GraphQL [HS08] or VF3 [Car+18], is certainly possible and even necessary for working with the grammar on a greater scale, but outside the scope of the current work.

5.2 Replacement Algorithm

The replacement algorithm, displayed in Algorithm 5.2, uses, as stated before, a singe push-out approach. To manage all the relations between different graphs and graph

Algorithm 5.1: Pseudo-code of the matching algorithm.

Input: mother graph M and host graph H and whether geometric ordering is to be kept

Output: R is a list containing possible matchings.

All lower case m and h are graph elements (i.e. either edges or vertices) with $m_x \in M, \forall x$ and $h_x \in H, \forall x$;
$\{x$: $y\}$ defines a dictionary mapping x to y. T is a task list containing tuples of (mapping, unmappend elements);
get starting element m_0 from M;
For h **in** *H:*
    if *h.matches(m_0):* **then**
        add ($\{m_0$: h$\}$, {unmapped neighbours of m_0: m_0}) to T;
    else
    end
While *exists* t **in** T*:*
    pop (mapping, unmapped elements) from list T;
    if *mapping is complete* **then**
        add the complete mapping to R;
        continue;
    else
    end
    pop (m_x, $m_{x\text{-parent}}$) from unmapped elements;
    map $m_{x\text{-parent}}$ to $h_{x\text{-parent}}$ according to mapping;
    Create a list *newmappings* of possible new mappings;
    For *neighbours* h_x *of* $h_{x\text{-parent}}$*:*
        if h_x *is already mapped* **then**
            continue;
        else
        end
        if m_x *is a directed Edge:* **then**
            check if h_x matches the direction of m_x, if not **continue**;
        else
        end
        if h_x *does not match the matching functions defined by* m_x*:* **then**
            continue;
        else
        end
        if *The already matched neighbours of* h_x *and* m_x *are incompatible:* **then**
            continue;
        else
        end

```
35
36  |
37  |   | if Geometric ordering in M should be kept by the match then
38  |   |   | Test if ordering of h_x is compatible with m_x, if not continue;
39  |   | else
40  |   | end
41  |   | We found a matching element h_x for m_x;
42  |   | A new mapping including {m_x: h_x} to newmappings;
43  | end
44  | if The list newmappings is empty then
45  |   | This branch has no possible mappings, discard it. continue;
46  | else
47  | end
48  | Calculate the new unmapped elements dictionary;
49  | new mapping in new mappings
50  |   | add (new mapping, new unmapped elements) to task list T;
51  | end
52 end
53 return R;
```

elements, the implementations ends up using five graphs with mappings between them to apply a production.

The usual explanation of a derivation process of applying a production p to a host graph H encountered in literature takes the following actions:

1. Delete elements of H which need to be removed according to p.
2. Add the new elements that p defines need to be added.
3. Connect the existing elements of H to the new elements according to the instructions in p.

In the actual implementation some additional steps were necessary and two of the above steps where collapsed into one, but it follows the same structure:

1. Calculate which elements to add, remove or change and for which elements new attribute values need to be calculated. (Done when creating/loading the production)
2. Delete any element marked for removal. (Lines 3 to 9 of Algorithm 5.2)

3. Add the new elements, which will automatically connect them as necessary. (Lines 10 to 14 of Algorithm 5.2)

4. Calculate the new value of attributes, for those elements where this is necessary. (Lines 15 to 18 of Algorithm 5.2)

Algorithm 5.2: Pseudo-code of the production application algorithm.

Input: host graph H, mother graph M, daughter graph D.
Output: The graph resulting from applying the production R.
Data: There are the following graphs and abbreviations:
- R : Result graph
- H : Host graph
- M : Mother graph
- D : Daughter graph
- C : Copy of daughter graph

With the starting relationships:
R <-> H : 1-to-1 copy
H <-> M : partial isomorphism
M <-> D : manual mapping
D <-> C : 1-to-1 copy

```
1  Copy H to R
2  Copy D to C
3  for edges e which get reconnected by the production do
4  |  remove the connection to the vertex which will be disconncted from e
5  |  remove e from the neightbourhood list of the vertex
6  end
7  for e in elements to remove (e part of R): do
8  |  remove e from R
9  end
10 for e in elements to add (e part of C): do
11 |  map connections of e from C to R
12 |  if the position of e needs to be calculated automatically do so
13 |  add e to R (This automatically connects elements as necessary)
14 end
15 Calculate the vectors which will help in calculating attribute values
16 for e in elements whose attributes need to be calculated (e part of R): do
17 |  change attributes of e according to attribute deffinitions in D
18 end
19 return R
```

For a sketch of the logic behind the application of a production see Algorithm 5.2. The most difficult to understand part of the whole algorithm is the hierarchy of the five different graphs and how they are mapped to each other. A graphical representation of

the relationships can be found in Figure 5.1. M and D are the mother or left-hand-side and daughter or right-hand-side graphs of the production, respectively. The mapping between them is supplied by the user and decides whether an element is kept, deleted or added to the result. The mother graph M is matched to the host graph H with a subgraph isomorphism as described in Section 5.1. R and C are deep copies of H and D respectively. Working with a copy of H as the basis of the result graph R allows one to just delete old elements, add new elements and change some of the remaining ones, while leaving the majority of elements within R untouched. Without creating this copy first, one would have to create a copy of each individual graph element as needed and then add them to a result graph, which would result in more complicated code. C, the copy of the daughter graph, is also just a function of convenience. This allows one to add the new elements to R by just changing the references to neighbours in elements of C, without creating new elements in each individual case.

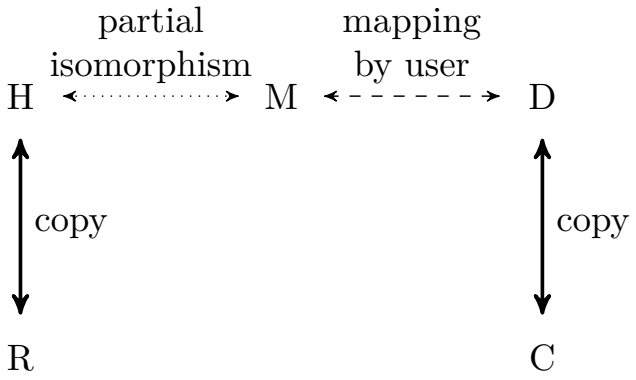

Figure 5.1: Overview of the hierarchy of graphs in a production application.

As stated before, which elements are kept, added or removed is decided by the mapping between M and D supplied by the user. The rules, which are visualized in Figure 5.2, are:

1. Elements of the host graph H not part of the partial isomorphism with M are kept without changes, except for maybe losing connections to deleted elements.
2. Elements with a mapping from M to D in the production (blue in the figure) are kept, but their attributes are recalculated according to instructions in the corresponding element in D.
3. Elements of M without a mapping from M to D (red in the figure) are deleted.
4. Elements of D without a mapping from M to D (green in the figure) are added to the result. References to elements from point 2 are translated to references to elements in R, the copy of H.

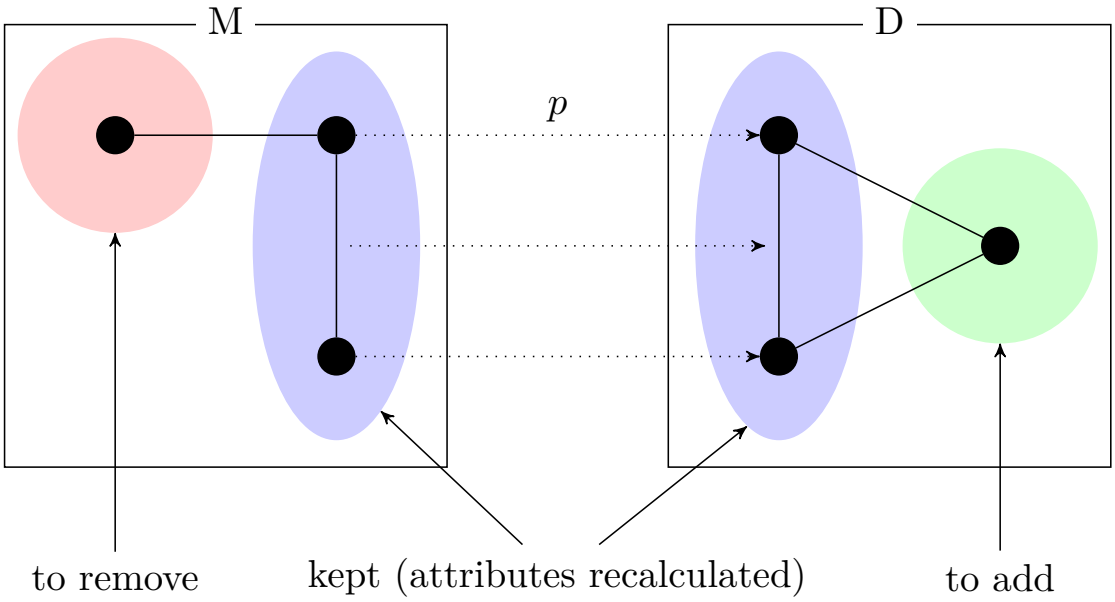

Figure 5.2: Visualisation of the effect of the partial graph morphism p. The dotted arrows going from M to D represent the partial graph morphism p. Elements which will be added are circled in red, elements to be added in green and elements kept unchanded in blue.

5.3 Calculation of Attributes and Position of Elements

The attribute calculation instructions can be arbitrary python instructions, allowing, in particular, for the use of the python library „random“ to add randomness to attribute values. To ease the calculation of attributes which are of geometric nature, such as new positions or lengths of vectors, a production can define vectors which will be available for use in the argument calculation formulas. For examples see Section 7, and in particular the Koch Snowflake production.

The automatic calculation of new positions currently calculates the barycentre of the daughter graph D and the subgraph isomorphism of the mother graph in the host graph H^M. The delta between an element of the daughter graph and the barycentre of the daughter graph is used to calculate the new position relative to the barycentre in the host graph. To account for potential rotation of the match in the host graph, a „direction“ is calculated for both the mother graph and its matching subgraph in the host graph, using total least squares. The difference in direction between the two directions is used to rotate the newly calculated position. This new position is also scaled by the ratio of the maximum extent of H^M and D divided by the ratio of the maximum extent of M to D. This scaling allows productions to extend or shrink objects, depending on what subgraph of H they are matched to.

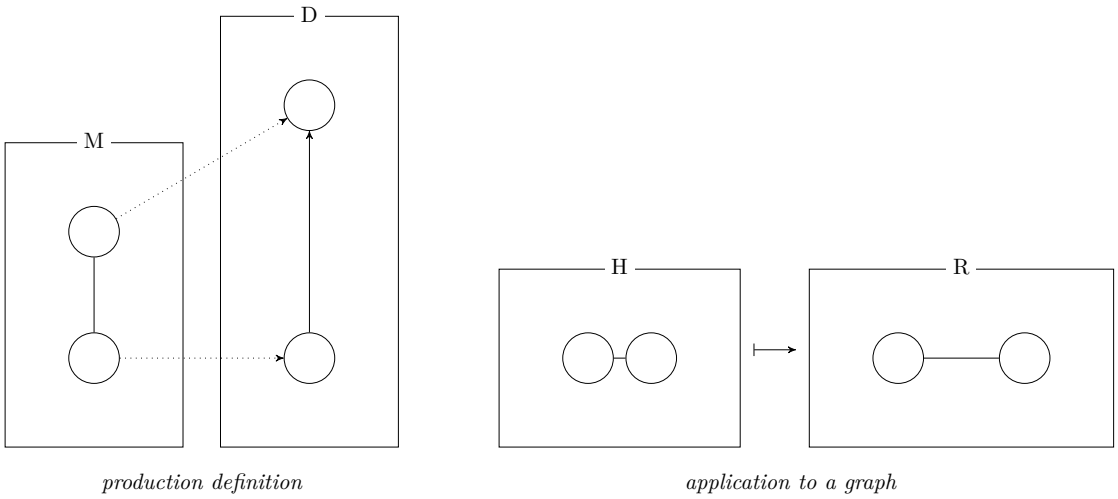

Figure 5.3: Example of the automatic calculation of positions. The production as defined on the left of this diagram will double the distance of any two connected nodes it is appied to. On the right is an example application of this production which shows how the direction of the growth is taken into account.

5.4 Export

The derivation result of the grammar can be saved to a YAML[1] file containing all information about the produced graph, or as an SVG file for purposes of visualization. SVG was chosen as the visual export format because it has a simple structure and good library support in python. Every element of the graph is exported into precisely one SVG tag. By default a vertex is exported as a circle and an edge as a line, but this can be changed and configured by setting special attributes on graph elements starting with `.svg_`.

For example, a node can also define an attribute `.svg_tag` with the value `path` and another attribute with the name `.svg_d` containing the SVG path information. This would then be exported as an SVG `<path d="x">` tag instead of a circle. The strength of this system is that it allows productions to change the export settings, through calculations based on the values of other attributes. This can be seen in full effect in the creation of circular patterns in Section 7.

[1] YAML Ain't Markup Language

CHAPTER 6

User Interface

Early on in the project, when implementing the first example productions, it became apparent that, although a graphical interface was not technically necessary to work with graph grammars, a purely text-based interface was tedious and confusing to use. Therefore a simple user interface, which would offer the basic functionality necessary to view and edit productions and derivations, was implemented. The vehicle of a bachelors thesis did not offer enough room to increase the scope to include a detailed study of user interfaces for procedural generation; hence this section appears for the sake of completeness, not because of major contributions to the field of user interface design.

It should be noted that graphical representations of graphs is not without its pitfalls. Graphs are, after all, abstract data structures without any geometric relation between the elements. The proposed graph grammar has the capability of supporting both such pure graphs and graphs with added geometric information. A graphical representation will, however, always suggest to the user a geometric relation, whether or not one exists. This will, at times, suggest intuitive relations which do not hold and therefore produce results surprising for the user. Nonetheless, even being aware of this caveat, it is the author's believe that graphical representations of graphs and of graph grammars are the best and most intuitive means of interacting with them.

The graphical user interface is split apart into three tabs, each of which serve a different purpose in the process of defining a grammar. First comes the host graph view, where the user can create the starting graph for the grammar. Second, the production view, where most of the users time is spent to create the rules of the grammar. And lastly, the result view, where the user can inspect all graphs produced during a derivation and export selected graphs to an SVG or YAML file.

The host graph view seen in Figure 6.1 allows one to see a list of available axiom graphs for a grammar. The selected graph is visualized on the right side of the panel and can be modified as necessary. Two buttons below the list allow one to add new and remove

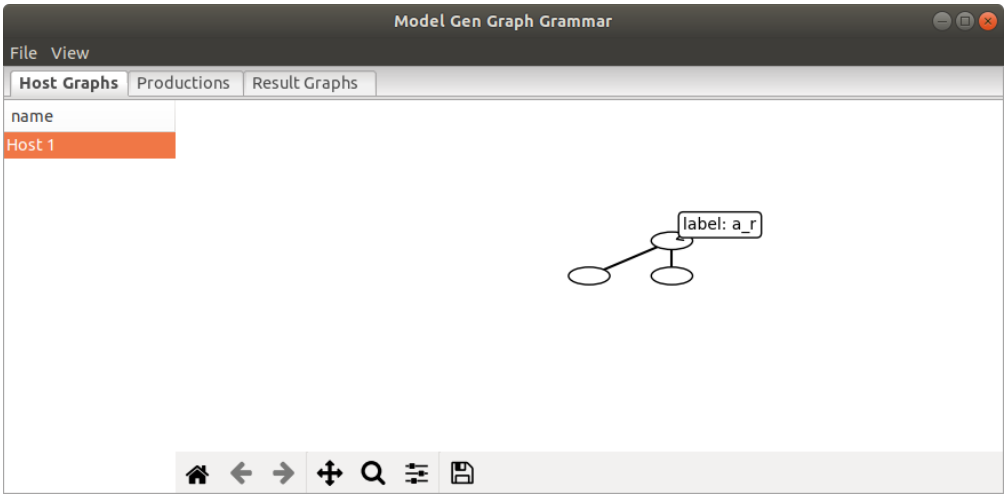

Figure 6.1: Host graph view of the GUI

existing graphs from the list. Once an appropriate start graph has been prepared, it can be selected as the active start-graph and will hence be used for any applications of productions.

Having a separate view for start graphs was not strictly necessary, the grammar could have been defined to always start with a graph containing only a single starting node without losing any expressiveness. However, when considering this question from the point of practicability, having the capability to apply a specific production to a more complicated, hand crafted start-graph to see the results and to be able to switch between them is a useful function. Without such a feature, actual development of a complex set of productions could to become tedious, especially during bug-fixing.

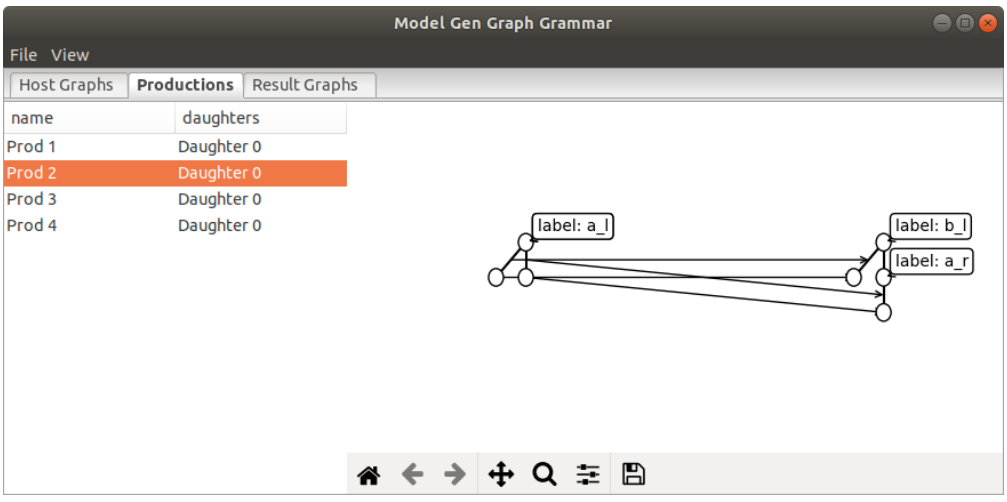

Figure 6.2: Productions view of the GUI

The left hand side of the interface remains quite similar for the production view depicted in Figure 6.2, the only difference being that it has two columns: One naming the mother graph, the other part differentiating between different daughter graphs of the same mother graph. The graph display portion has some different features compared to the start-graph and result-graph displays. The visualisation is split in half, the left side representing the left-hand-side or mother graph of the production while the right side represents the right-hand-side or daughter graph of the production. The partial graph morphism p is depicted by arrows pointing from the elements in the mother graph to the corresponding elements in the daughter graph.

This particular view also contains the contribution this paper makes to graph grammar definition UIs: The automatic calculation of new positions depending on the relative positions on elements in the mother and daughter graph. If two mapped points are closer together in the daughter graph than in the mother graph, applying this production will move them together, relative to the distance of the two points in the host graph. The same process is available for increasing the distance of two points and for rotations. A detailed description of how the calculations are done can be found in Section 5.3. The author is not aware of any previous work proposing such a solution to ease the definition of productions in a graph grammar.

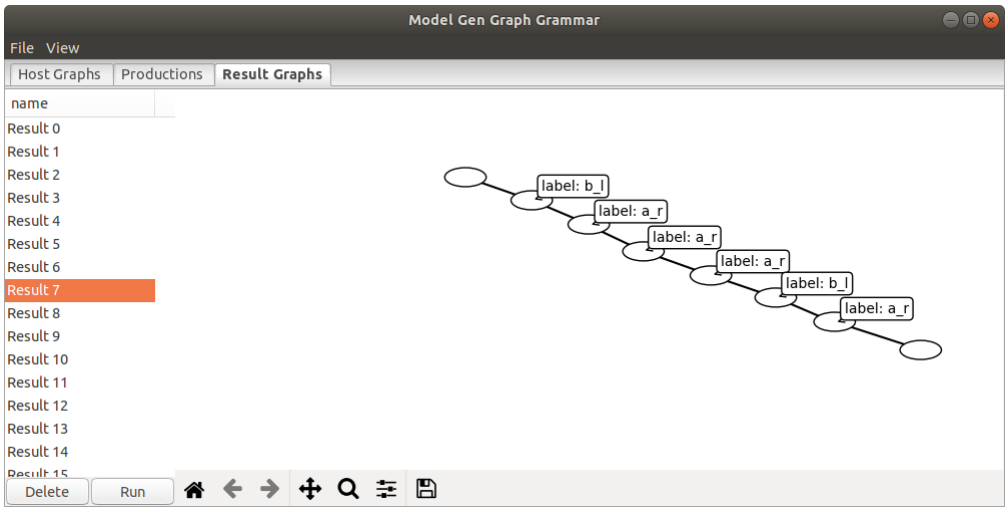

Figure 6.3: Results view of the GUI

As for the result view shown in Figure 6.3, the basic UI elements are the same as with the start-graph view, the only difference being that no changes to these graphs can be made. To allow the user to retrace how a result came about, the result view shows all intermediate derivations steps, not just the final result.

CHAPTER 7

Results

This section reports the results of applying the new grammar proposed in this work to a number of modelling tasks. The tasks where chosen to represent a variety of modelling problems, in which either L-systems or shape grammars find the most successful application.

The descriptions show the results of the SVG export, a diagram of the axiom graph and details of the involved productions. For each production, there is a diagram showing the mother graph, the daughter graph and the partial graph morphism between them, as well as a separate table detailing the attributes of the elements of both graphs. For elements of the mother graph those attributes are application conditions, while for elements of the daughter graph these attributes are either constants or calculation instructions, which define the way in which the new value will be derived. Furthermore, if the production uses vectors to manually calculate new positions of vertices, then the definition of these vectors are also listed in the aforementioned table.

7.1 Modelling of a Koch Snowflake

The Koch Snowflake or Koch curve was defined by the Swedish mathematician Helge von Koch as splitting a line into three equally long parts, extending the middle part into an equilateral triangle pointing outwards, and then removing the original centre part [Koc06]. The traditional way of creating a whole snowflake based on this curve is to apply this production rule to an equilateral triangle and let it grow from there.

Given the form of its definition, the Koch Snowflake lends itself to being implemented with formal grammars. Indeed, looking at this definition, one could even say that Koch essentially defined the curve based on an informal shape grammar, in the original meaning of the term. The most common means of defining the snowflake in a grammar is a set of L-System rules like the one below:

```
Alphabet: F, +, -
Axiom: F--F--F
Production rules:
F -> F+F--F+F
```

Which is then interpreted either by a Logo-style turtle as: F moves the turtle forward, + turns the turtle 60° to the left and - turns the turtle 60° to the right; or by interpreting it as vector graphics with each L-system symbol being associated with a fixed vector displacement [ODA03].

A graph grammar implementation of the same process, which would also end up being interpreted by turtle graphics to view the result, can be found in [Kni08]:

```
public void rules() [
    Axiom ==> F(10) RU(120) F(10) RU(120) F(10);
    F(x) ==> F(x/3) RU(-60) F(x/3) RU(120) F(x/3) RU(-60) F(x/3);
]
```

The production rules are essentially the same as for L-system, but now instead of producing a string made up of 'F', '+' and '-' characters, a graph daisy-chaining 'F(x)' and 'Rotate(φ)' nodes is created.

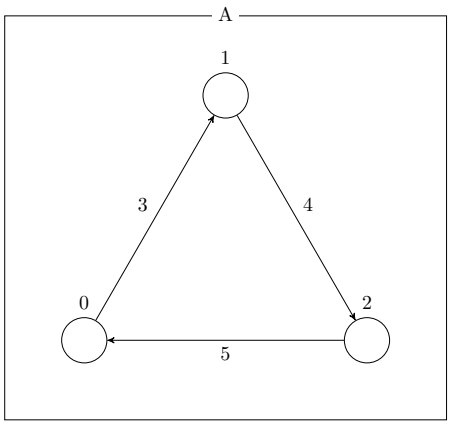

Figure 7.1: Axiom graph of the Koch Snowflake derivation.

Element	Attribute	Value
0-2:	`.svg_stroke_width`	0
3-5:	`.directed`	True
	`level_of_detail`	0

Table 7.1: Attribute definitions of the Axiom Graph of the Koch Snowflake.

The graph grammar presented in this paper is capable of forming such a graph which could be interpreted by a Logo-style turtle, but it is also capable of working on this problem entirely with connected geometric primitives. This is the approach presented here. It starts with a triangle as it's axiom graph as seen in Figure 7.1. The Table 7.1 shows what attributes the elements of the axiom graph have. The nodes have an SVG attribute, which makes them invisible in the output, since we are only interested in the lines. All edges are given the `.directed` attribute to mark them as directed edges. The direction in which they point is shown in the diagram by arrowheads.

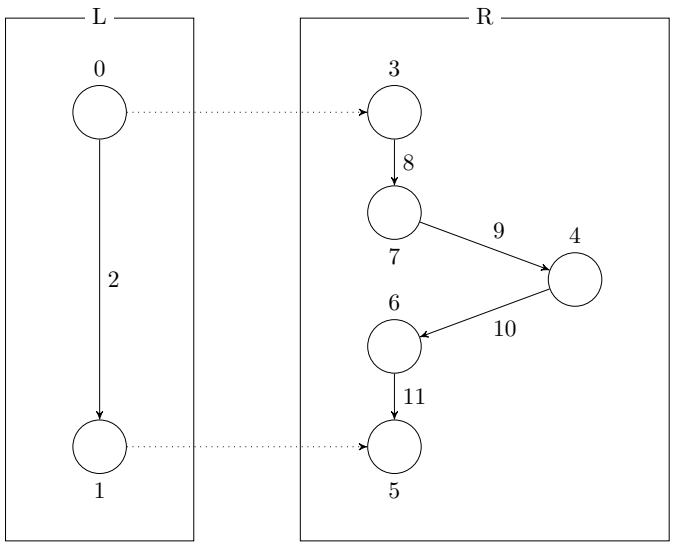

Figure 7.2: The single production of the Koch Snowflake derivation.

The solution's sole production rule is shown in Figure 7.2, with Table 7.2 showing the corresponding attribute definitions. It takes a straight line, splits it apart, and adds the protruding spike on the outside of the snowflake. To determine which side is the outside, directed edges are used, as shown by the arrows in the diagram. The newly added vertices have their SVG `stroke-width` set to zero, as in the axiom graph, to make them invisible in the export. Additionally, to assure that the Koch Snowflake is correctly constructed, the new positions of added vertices are calculated by hand rather than automatically. To simplify the position calculations two vectors are defined, as specified in the bottom of Table 7.2. Vector **A** is a point vector at the position of node 3 and **v1** is a directional vector going from node 3 to node 5. The function `perp_left()` creates a perpendicular vector pointing to the left of the vector passed as an argument. The option `max_level_of_detail` controls how often a single line will get subdivided, ensuring that all lines will end up with the same length. It's effect is equivalent to the number of production application in an L-system, where a single application is applied to all possible matches in parallel. The result of the derivation is found in Figure 7.3.

Element	Attribute	Value
2:	`.directed`	True
	`level_of_detail`	attr < max_level_of_detail
4:	`.new_pos`	A + v1/2 + 1/3∗(perp_left(v1))
	`.svg_stroke_width`	0
6:	`.svg_stroke_width`	0
	`.new_pos`	A + v1 ∗ 2/3
7:	`.svg_stroke_width`	0
	`.new_pos`	A + v1 ∗ 1/3
8-11:	`.directed`	True
	`level_of_detail`	edge.attr["level_of_detail"] + 1

Vector Name	Definition
A	Point 3
v1	Line from 3 to 5

Table 7.2: Attribute and vector definitions of the Production of the Koch Snowflake.

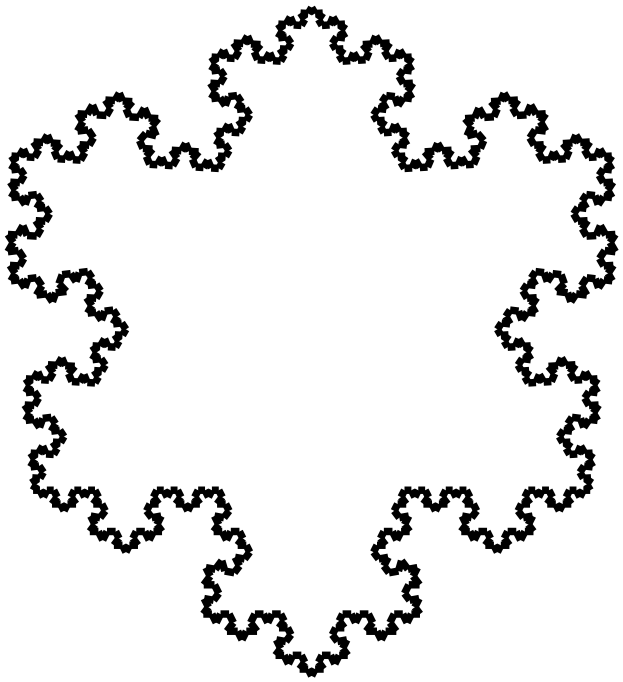

Figure 7.3: Result of a Koch Snowflake derivation.

7.2 Modelling of Patterns

Creation of patterns can be an interesting subject of study for grammars, because frequent repetition and the symmetric nature of patterns lends itself well to being expressed in grammars. In this subsection two different kinds of patterns are created by the proposed grammar. One is an infinitely tiling square pattern, the other is a self-contained circular pattern which is used to show the grammars capabilities for creating a varied set of interesting outputs from the same set of productions using randomness in attribute calculations.

7.2.1 A Tiling Square Pattern

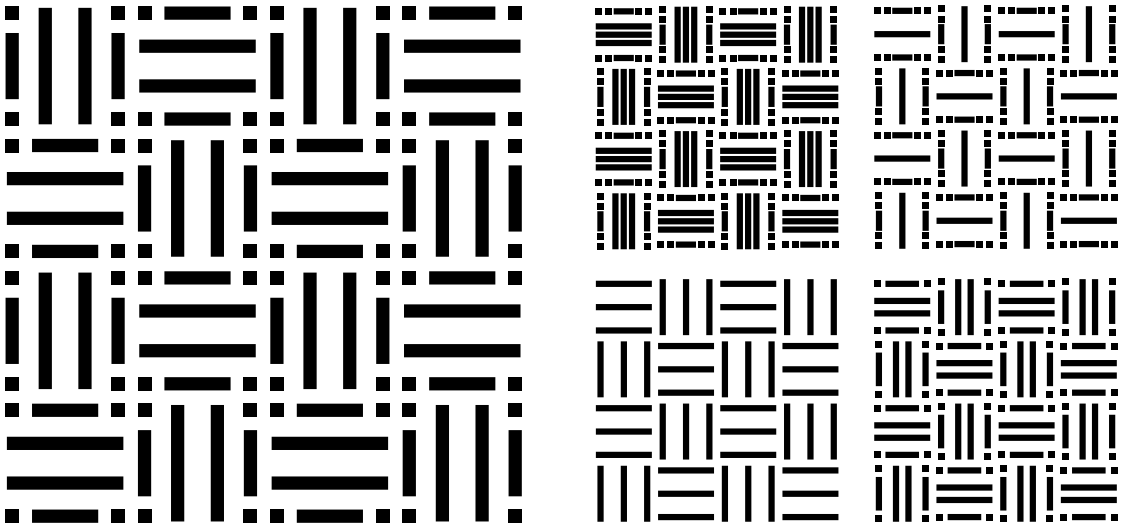

Figure 7.4: To the left is the result of the tiling square pattern derivation discussed in this chapter. To the right are different variations on this pattern generated with the described grammar.

As an example of a deterministic generation of a tiling square pattern, the result from Figure 7.4 is presented. The size of the result, and therefore the number of tiles, can be decided by an attribute in the axiom graph displayed in Figure 7.5 and Table 7.3. The `length` attribute is understood to be a length in centimetres for the purposes of the svg export, where a single tile of the pattern will have a length varying between 1cm and 2cm. If the length set in the axiom is greater than 2cm the resultant square will be split apart into four smaller squares, each of them being filled with the square pattern.

Element	Attribute	Value
0:	`size`	4

Table 7.3: Attribute definitions of the axiom graph of the square pattern.

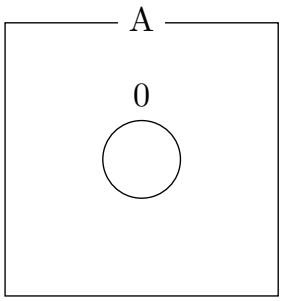

Figure 7.5: The axiom graph of the square pattern.

The detailed function of the productions is as follows. The production „Grow Square“, shown in Figure 7.6 and Table 7.4, extends the single starting node into a square of the appropriate length, as defined in the `size` attribute of the starting node.

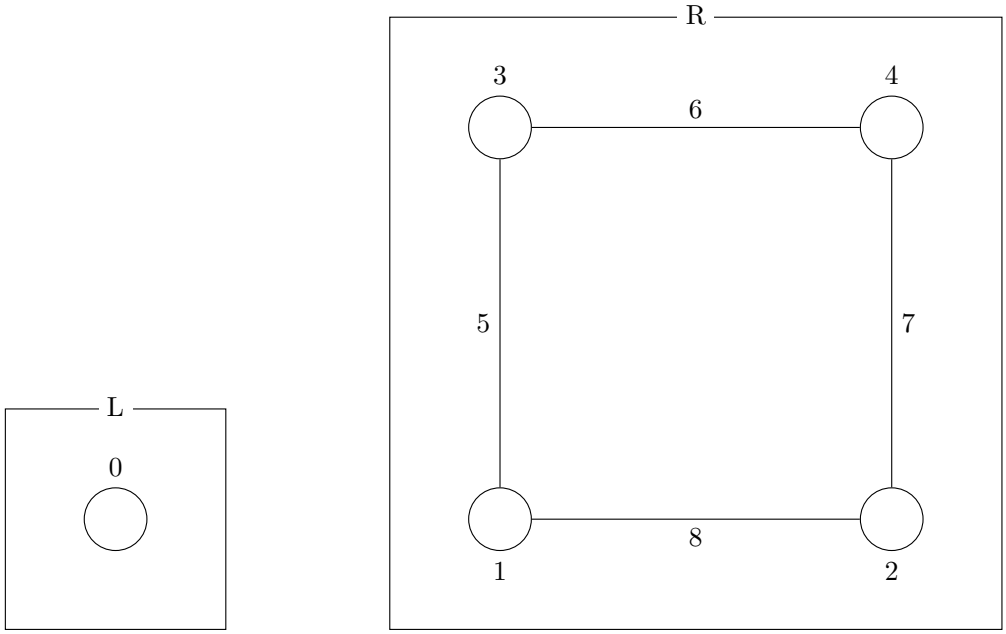

Figure 7.6: The production „Grow Square“ of the square pattern.

The „Subdivide Square“ production, Figure 7.7 and Table 7.5, is only applied if there is a square of length greater or equal to four. If there is such a square than it is split apart into four unconnected squares of equal length with some overlapping elements. This is the reason why the diagram in Figure 7.7 shows multiple numbers for some elements. There are in fact multiple edges and vertices on the same spot.

The production „Define Directions“, as displayed in Figure 7.7 and Table 7.6, makes the same structural changes to the graph as the production „Subdivide Square“, but gives

Element	Attribute	Value
0:	`size`	True
1:	`new_x`	$-$**float**(arg0.attr['size'])/2
	`new_y`	$-$**float**(arg0.attr['size'])/2
2:	`new_x`	**float**(arg0.attr['size'])/2
	`new_y`	$-$**float**(arg0.attr['size'])/2
3:	`new_x`	$-$**float**(arg0.attr['size'])/2
	`new_y`	**float**(arg0.attr['size'])/2
4:	`new_x`	**float**(arg0.attr['size'])/2
	`new_y`	**float**(arg0.attr['size'])/2
5-8:	`length`	arg0.attr['size']

Table 7.4: Attribute definitions of the production „Grow Square“.

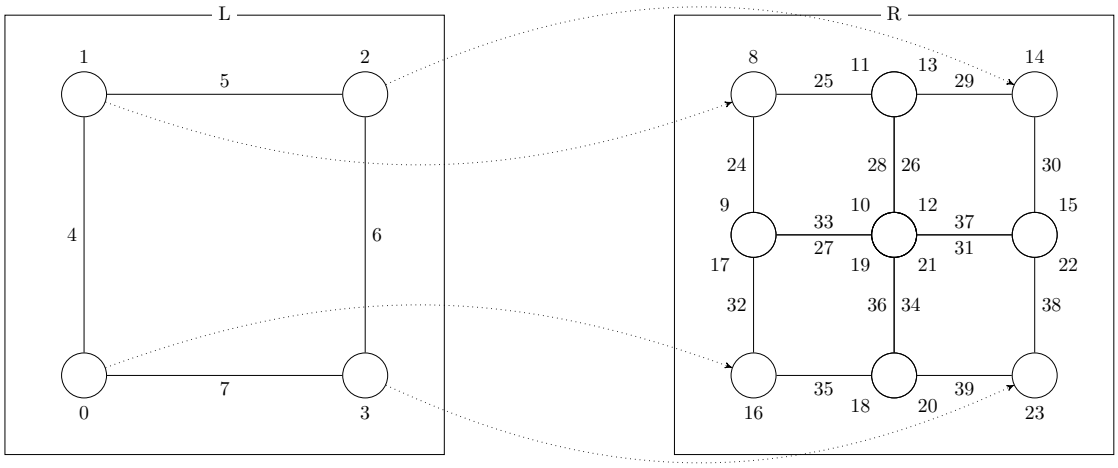

Figure 7.7: The productions „Subdivide Square“ and „Define Directions“ of the square pattern. They both use the same graphs with different attributes.

the edges different attributes. In particular it sets the attribute `label` to define which squares will be filled with a horizontal and which with a vertical pattern.

Now come the two production actually creating the information which gets exported to the SVG file: the production „Vertical Pattern“, as seen in Figure 7.8, Table 7.7, and the production „Horizontal Pattern“ in Figure 7.9, Table 7.8. They both define the same attributes for the elements of the graph, only differentiating themselves with the positioning of the newly added elements. Since there is no mapping from the left to the right side of the production all elements matched to L are discarded, which is why it was not necessary to set the lines or nodes used by the previous productions to be invisible in the SVG export. In the completed derivation, only the elements added by these two

Element	Attribute	Value
4-7:	`length`	**float**(attr)>=4
24-39:	`length`	**float**(arg0.attr['length'])/2

Table 7.5: Attribute definitions of the Production „Subdivide Square“.

Element	Attribute	Value
4-7:	`length`	**float**(attr)>=2 **and float**(attr)<4
24-27:	`label`	'vertical'
	`length`	**float**(arg0.attr['length'])/2
28-35:	`label`	'horizontal'
	`length`	**float**(arg0.attr['length'])/2
36-39:	`label`	'vertical'
	`length`	**float**(arg0.attr['length'])/2

Table 7.6: Attribute definitions of the production „Define Directions“.

productions remain. Setting the attribute `.svg_tag` to `none` means that the element in question will not be exported, setting it to `rect` produces a rectangle. The edges do not require any special SVG configuration, as the default export settings already fit the purpose.

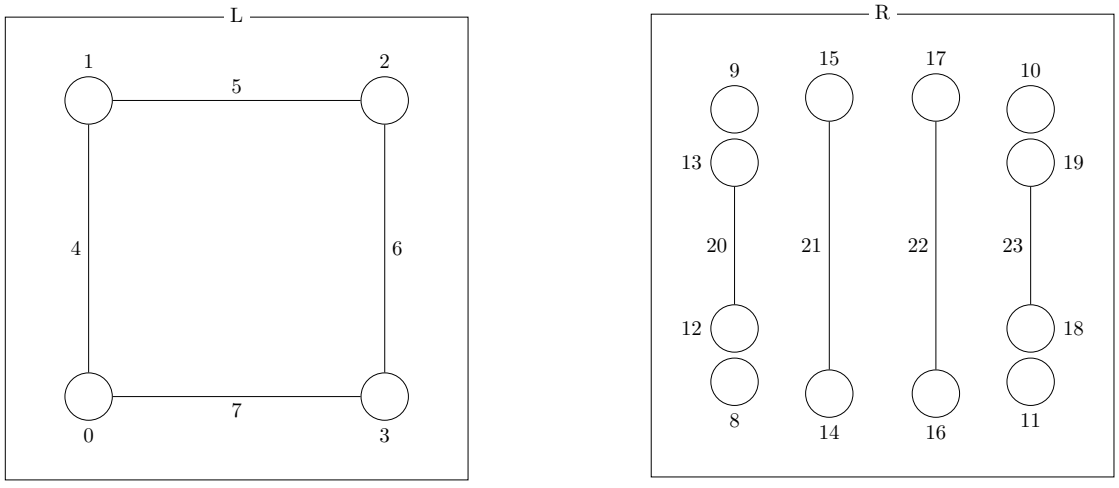

Figure 7.8: The production „Vertical Pattern“ of the square pattern.

Element	Attribute	Value
4-7:	`label`	attr==’vertical’
	`.svg_fill`	black
8-11:	`.svg_height`	0.1
	`.svg_tag`	rect
	`.svg_width`	0.1
12-19:	`.svg_tag`	none

Table 7.7: Attribute definitions of the production „Vertical Pattern“.

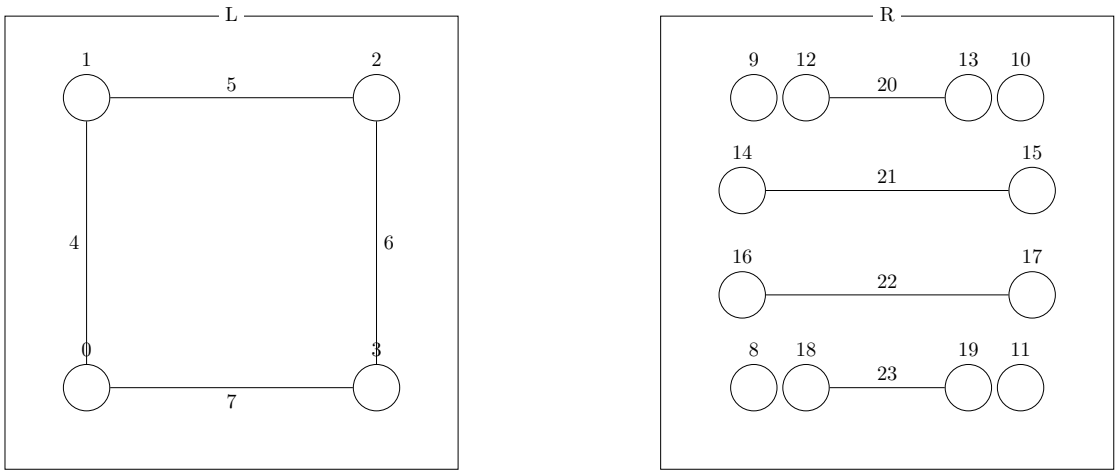

Figure 7.9: The production „Horizontal Pattern“ of the square pattern.

Element	Attribute	Value
4-7:	`label`	attr==’horizontal’
	`.svg_fill`	black
8-11:	`.svg_height`	0.1
	`.svg_tag`	rect
	`.svg_width`	0.1
12-19:	`.svg_tag`	none

Table 7.8: Attribute definitions of the production „Horizontal Pattern“.

7.2.2 Circular Patterns

To display the ability of this grammar to introduce randomness into the result of productions and to show the flexibility gained by allowing the export of any SVG tags, nine different results of the same set of circular pattern productions are shown in Figure 7.10.

The basic idea behind the construction of this set of productions is that, in the first step we will calculate four variables which will control the look of the resulting pattern. Then,

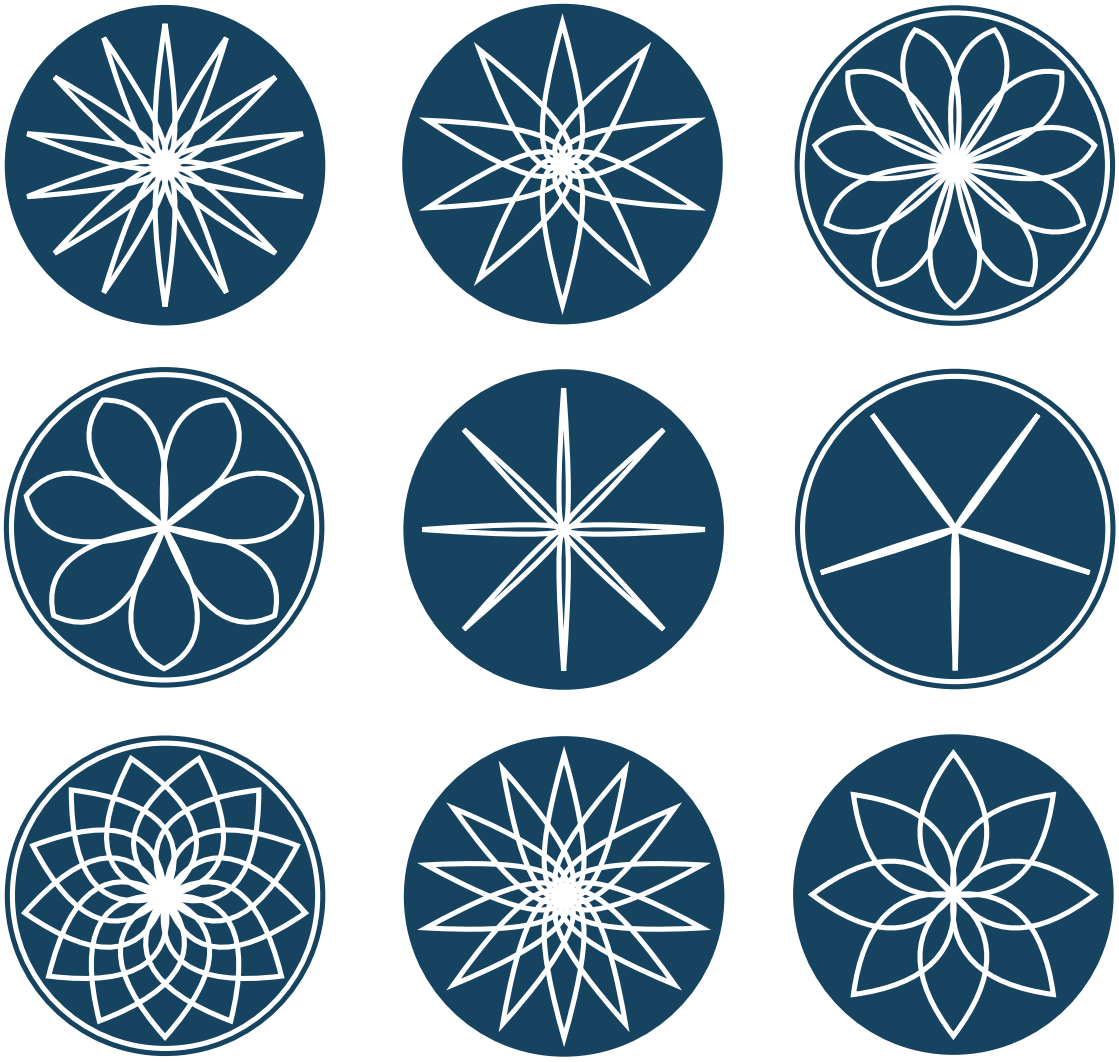

Figure 7.10: Nine different results of deriving a circular pattern.

one production adds a couple of edges, which will each represent a pair of curves going from the centre of the circle to the circumference. Another production rotates those edges around the centre of the circle, so that they are positioned at an equal distance to each, and lastly a finishing production sets the SVG attributes necessary to draw the curves.

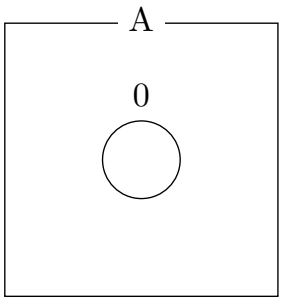

Figure 7.11: Axiom graph of the circular pattern

Element	Attribute	Value
0:	`.svg_fill`	*#154360*
	`.svg_r`	3.1cm
	`.svg_stroke`	*#154360*
	`finished`	False
	`label`	center
	`num_leaves`	0

Table 7.9: Attribute definitions of the axiom graph of the circular pattern.

In detail, the axiom graph from Figure 7.11, Table 7.9 defines the SVG attributes which will create the blue background circle in the export. It also sets some attributes which will be used to control the following productions such as the number of currently connected edges, here called „leaves“.

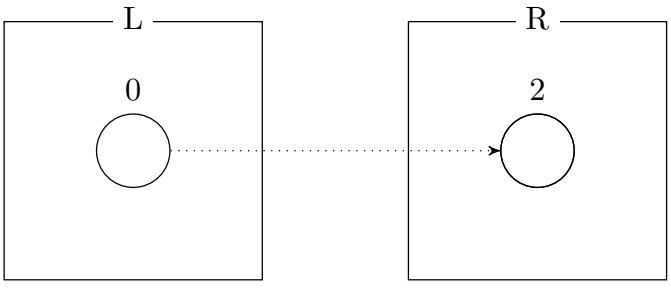

Figure 7.12: Production „Introduce Randomness“ of the circular pattern.

Element	Attribute	Value
0:	`finished`	attr==False
	`label`	attr=='center'
	`control_position`	random.uniform(0.05, 0.95)
1:	`control_degree`	random.uniform(0.01, 0.3)
	`finished`	True
	`max_leaves`	random.randint(5,15)
	`.svg_fill`	'#154360'
	`.svg_r`	'3cm'
2:	`.svg_stroke`	'white'
	`.svg_fill_opacity`	0.0
	`.svg_stroke_opacity`	random.choice((0.0, 1.0))

Table 7.10: Attribute definitions of the production „Introduce Randomness“.

The first production to be executed, given the priority 0, is called „Introduce Randomness“, see Figure 7.12 and Table 7.10. This production introduces four different elements of randomness. First, the opacity of the ring which can surround the leaves is chosen between zero and one, in effect either making it visible or invisible. Then, the attributes `control_position` and `control_degree` are calculated. These are used to position the control point of the quadratic Bézier curve in the SVG export. Lastly the number of leaves is randomly chosen between 5 and 15.

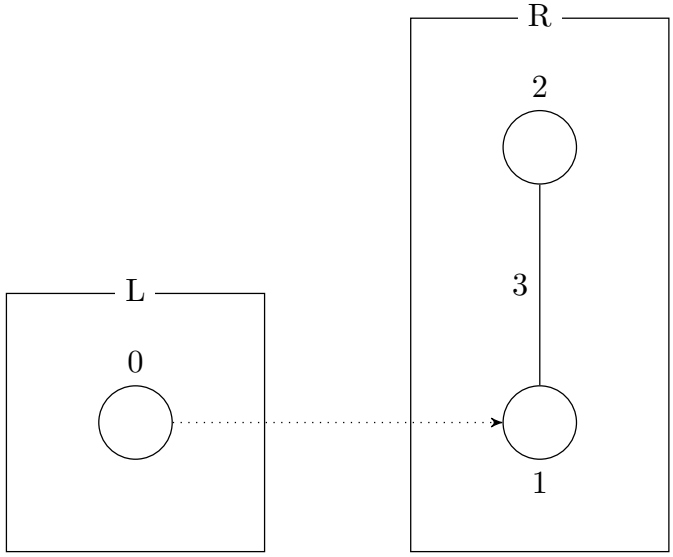

Figure 7.13: Production „Add Leaves“ of the circular pattern.

The next production to be applied is „Add Leaves“, Figure 7.13 and Table 7.11. This production does nothing more than adding an edge going from the centre to a point one

Element	Attribute	Value
0:	`label`	attr==’center’
	`num_leaves`	**int**(attr) < **int**(attrs[’max_leaves’])
1:	`num_leaves`	**int**(arg0.attr[’num_leaves’])+1
2:	`positioned`	False
3:	`finished`	False
	`number`	**int**(arg0.attr[’num_leaves’])

Table 7.11: Attribute definitions of the production „Add Leaves“.

unit above the centre. No special calculations are taking place. This production is given the priority 1.

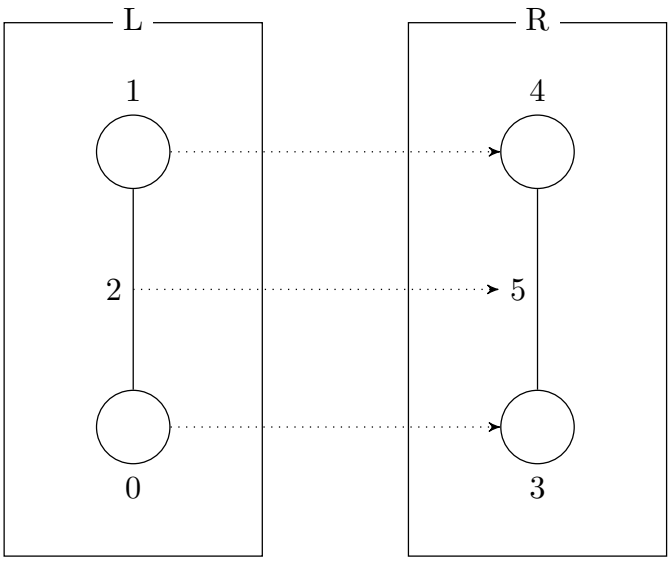

Figure 7.14: Production „Position Leaves“ of the circular pattern.

Element	Attribute	Value
0:	`label`	attr==’center’
1:	`positioned`	attr==False
4:	`.new_pos`	c + rotate(v1, 2 ∗ pi ∗ (**int**(leave.attr[’number’]) / **int**(center.attr[’max_leaves’])), c)
	`positioned`	True

Table 7.12: Attribute definitions of the production „Position Leaves“.

Following the addition of all leaves, the production „Position Leaves“ (Figure 7.14, Table 7.12) is applied to position all edges in an equal distance to each other around the centre

of the circle. Here in calculating the new position of the edge, the function `rotate(v, rad, c)`, which rotates the given vector `v` by an angle expressed in radians around a centre point `c`, is used. This production could technically be given the same priority as „Add Leaves“, but for debugging purposes it was more practical to give it the priority 2, so that each production is exhaustively applied in sequence.

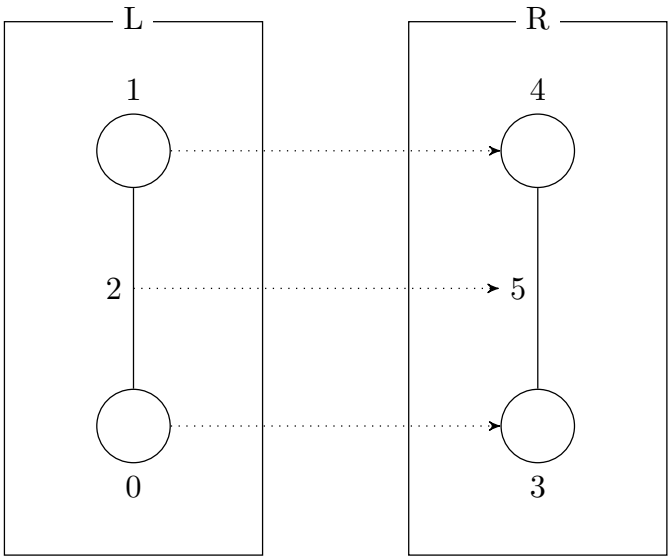

Figure 7.15: Production „Configure Curves“ of the circular pattern.

To finish the derivation, the production „Configure Curves“ with priority 3, as seen in Figure 7.15, Table 7.13, is applied. The important part is the calculation of the values making up the SVG curve in the attribute `.svg_d`. It creates an SVG path made up of two quadratic Bézier curves, one going from the centre to the outside edge of the circle, and one going the other way, so that the two are a mirror of each other. The shape of these curves is controlled by the attributes `.control_degree`, which defines how far from the edge the control point is rotated away, and `control_position`, which defines how close to or far from the centre of the circle the control point lies.

Element	Attribute	Value
0:	`label`	attr==’center’
2:	`finished`	attr==False
4:	`.svg_tag`	’none’
5:	`.svg_d`	f’M␣s.x∗35␣s.y∗35␣Q␣(s␣+␣rotate(v1,␣2∗pi∗float(arg0.attr["control_degree"]),␣s)∗float(arg0.attr["control_position"])).x∗35␣(s␣+␣rotate(v1,␣2∗pi∗float(arg0.attr["control_degree"]),␣s)∗float(arg0.attr["control_position"])).y∗35␣e.x∗35␣e.y∗35␣Q␣(s␣+␣rotate(v1,␣−2∗pi∗float(arg0.attr["control_degree"]),␣s)∗float(arg0.attr["control_position"])).x∗35␣(s␣+␣rotate(v1,␣−2∗pi∗float(arg0.attr["control_degree"]),␣s)∗float(arg0.attr["control_position"])).y∗35␣s.x∗35␣s.y∗35’
	`.svg_fill_opacity`	’0.0’
	`.svg_stroke`	’white’
	`.svg_stroke_width`	’1mm’
	`.svg_tag`	’path’
	`finished`	True

Table 7.13: Attribute definitions of the production „Configure Curves“.

7.3 Modelling of a Tree

As part of the effort to show the versatility of the graph grammar approach, the following part will show how plants can be modelled easily and how the grammar can be used for more artistic productions.

Figure 7.16 shows the result of the tree derivation in the same style as is using in Prusinkiewicz and Lindenmayer [PL96], as a black and white tree skeleton. The productions used to create these tree skeletons are explained in detail in the remainder of this sub-section. The results from Figure 7.17 are obtained by adding two additional productions which are run after the trees seen in Figure 7.16 are finished generating. One production places brush strokes along the trunks and branches and the other places brush strokes at the end of branches. Each brush stroke is a scaled and rotated grey-scale image of a brush stroke, added as an SVG `image` element to the export. The different colours are obtained by applying various `feColorMatrix` filters. This is another display of how the versatility of SVG can be put to elegant use within proposed graph grammar.

The productions used in creating these trees where inspired by and adapted from the descriptions of trees by Honda [Hon71] as demonstrated in Prusinkiewicz and Linden-

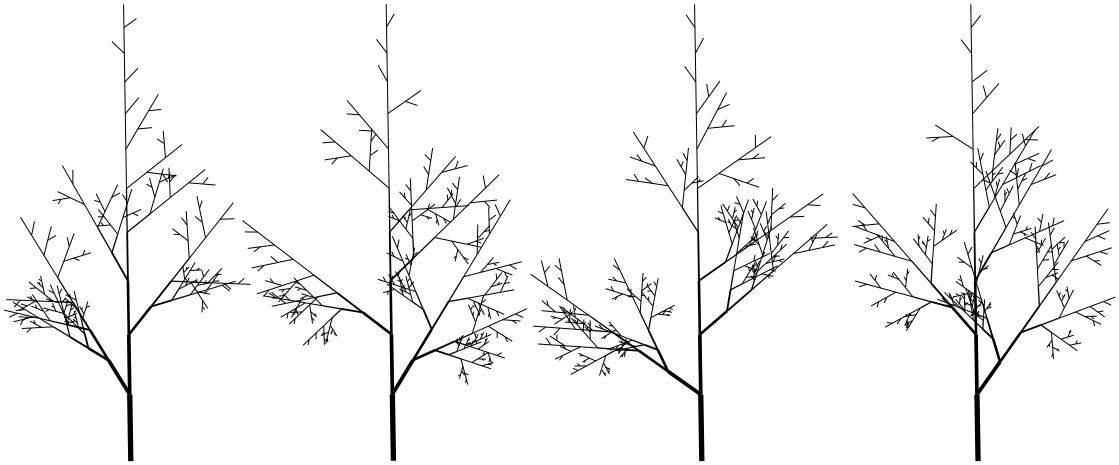

Figure 7.16: Four possible results of running the tree derivation.

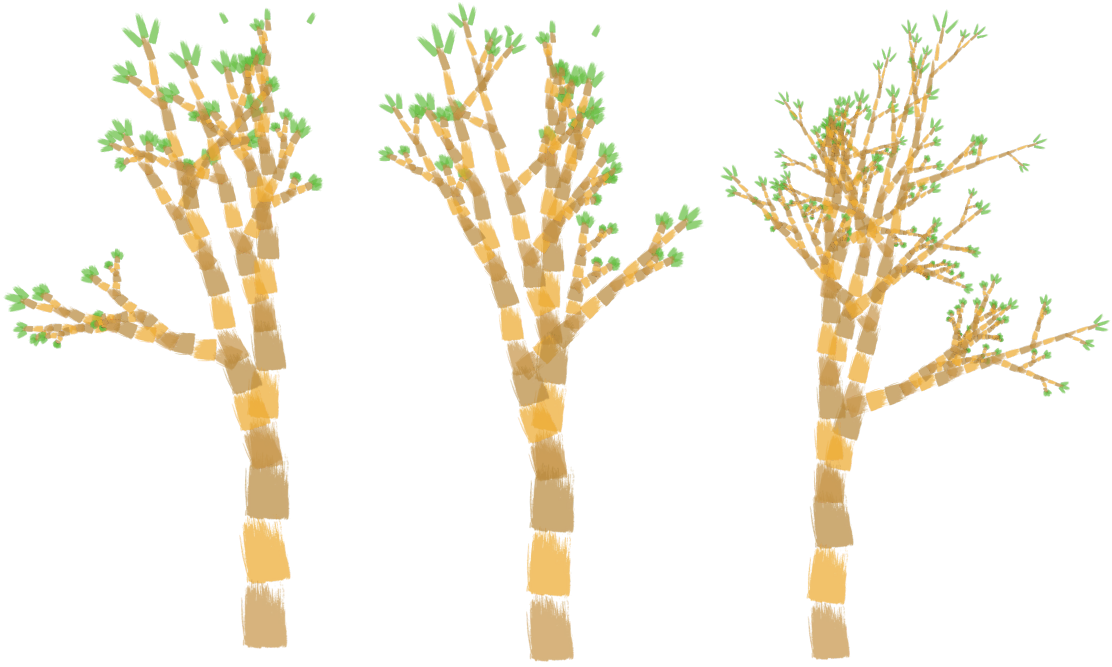

Figure 7.17: Three possible results of the painted tree derivation.

mayer [PL96]. It is a fairly simple set of two productions which grow the tree, while decreasing the width and length of additional segments with each step.

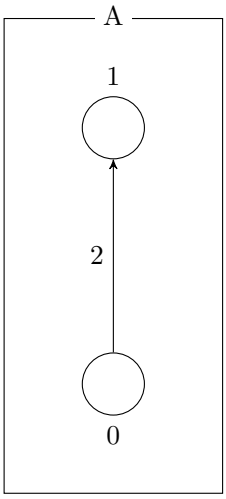

Figure 7.18: Axiom graph of the tree productions.

Element	Attribute	Value
0:	`.svg_tag`	None
1:	`.svg_tag`	None
2:	`.directed`	True
	`.svg_stroke_width`	10
	`label`	Trunk
	`section_height`	1

Table 7.14: Attribute definitions of the axiom graph of the tree example.

The axiom graph, seen in Figure 7.18 and Table 7.14, defines the starting piece of a tree-trunk and the starting width of the tree.

The production „Grow Trunk“ from Figure 7.19 and Table 7.15 is used to grow the tree trunk in a vertical direction. Each step reduces the width and the length of the next segment piece by a constant factor, while adding a new branch going out to the side at a randomly calculated angle. The attribute `segment_height`, which is incremented for each application of this production, is used to stop the vertical growth of the tree once a particular point has been reached.

The production „Grow Branch“ from Figure 7.19, Table 7.16 is used to grow the branches of the tree. The calculations done are almost the same as for the „Grow Trunk“ production with the only differences being that it has no limit on the number of applications and that it produces two new edges labeled as `Branch` rather than one `Branch` and one `Trunk` edge.

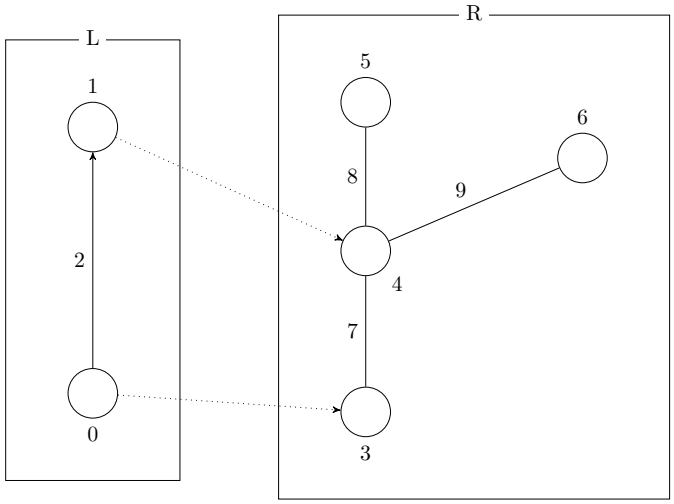

Figure 7.19: The productions „Grow Trunk“ and „Grow Branch“ of the tree generation example. They both use the same graphs with different attributes.

Element	Attribute	Value
	.directed	True
2:	label	attr=='Trunk'
	section_height	**int**(attr) < 11
3:	.new_pos	A
4:	.new_pos	B
5:	.new_pos	B + v1 ∗ 0.9
	.svg_tag	None
6:	.new_pos	B + Vec(vec1=A, vec2=rotate(B, pi/2 ∗ random.choice((−1,1))∗ (random.random()/ 3 + 0.3), A))∗ 0.6
	.svg_tag	None
7:	.svg_stroke_width	trunk.attr['.svg_stroke_width']
	.svg_stroke_width	**float**(trunk.attr['.svg_stroke_width'])∗ 0.7
8:	label	'Trunk'
	section_height	**int**(trunk.attr['section_height'])+1
9:	.svg_stroke_width	**float**(trunk.attr['.svg_stroke_width'])∗ 0.7
	label	'Branch'

Vector Name	Definition
A	Point **0**
B	Point **1**
v1	Line from **0** to **1**

Table 7.15: Attribute definitions of the production „Grow Trunk“ of the tree generation example.

Element	Attribute	Value
2:	`.directed`	True
	`label`	attr==’Branch’
3:	`.new_pos`	A
4:	`.new_pos`	B
5:	`.svg_tag`	None
	`.new_pos`	B + v1 ∗ 0.9
6:	`.svg_tag`	None
	`.new_pos`	B + Vec(vec1=A, vec2=rotate(B, pi/2 ∗ random.choice((−1,1))∗ (random.random()/ 3 + 0.3), A))∗ 0.6
7:	`.svg_stroke_width`	trunk.attr[’.svg_stroke_width’]
8-9:	`label`	’Branch’
	`.svg_stroke_width`	**float**(trunk.attr[’.svg_stroke_width’])∗ 0.7

Vector Name	Definition
A	Point **0**
B	Point **1**
v1	Line from **0** to **1**

Table 7.16: Attribute definitions of the production „Grow Branch“ of the tree generation example.

7.4 Modelling of Façades

The modelling of building façades is a typical procedural modelling task for which shape grammars appear to be the tool of choice. They lend themselves to a subdivision approach, where the surface of a building is continuously divided into smaller and smaller parts until all important elements of a façade, such as windows, doors, ledges and the like, are placed [Mül+06]. In the usual approach this subdivision does not result in a finished 3D model, but rather in a „building plan“ of the façade, into which scaled and rotated 3D models, created in external applications, are loaded at the appropriate positions [JCS16].

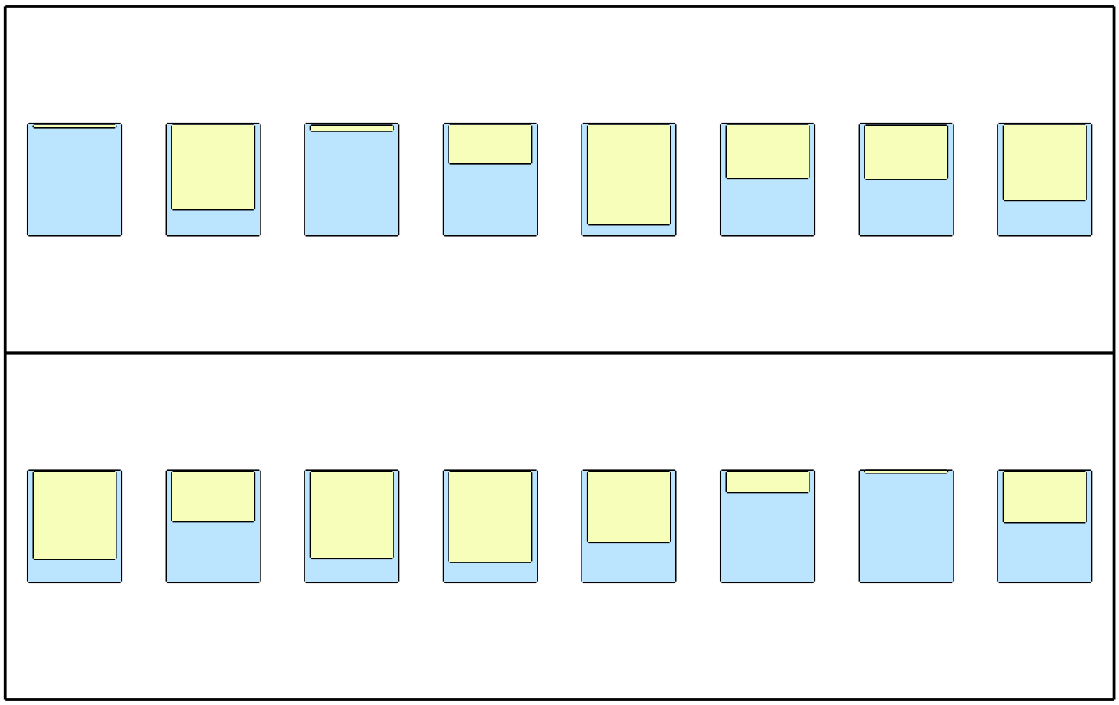

Figure 7.20: Result of a very simple building façade generation.

The result of a simple example of using the proposed graph grammar to model a building's façade, which will be discussed in detail in this sub-section, is shown in Figure 7.20. It is a simplified, schematic view of a façade with two floors, the windows having blinds which are placed at random states of unrolling. A slightly more detailed variation of such a schematic façade is shown in Figure 7.21, containing window-sills and a randomly placed door on the ground-floor. The starting point is a single node as shown in Figure 7.22, Table 7.17, containing within its attributes information on the length and height of the façade.

The `height` attribute is interpreted first by the production „Grow House“ (Figure 7.23, Table 7.18) to create an edge of appropriate length, and then by the follow-up production „Split House to Floors“ (Figure 7.24, Table 7.19), which will split the edge with the label

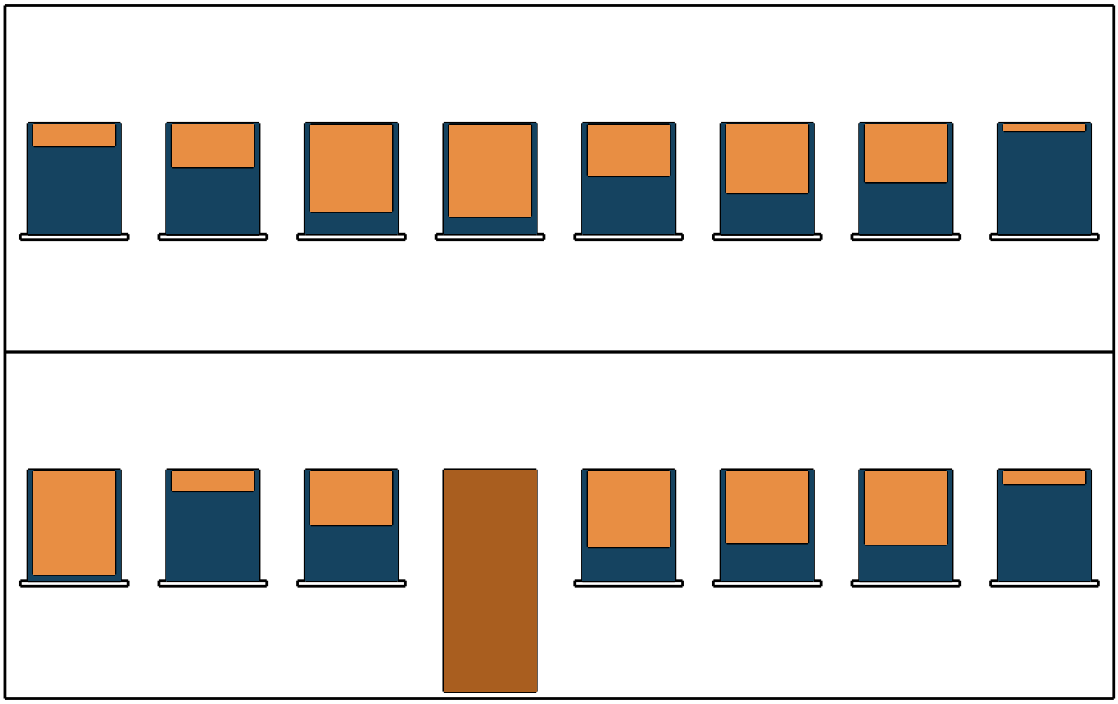

Figure 7.21: Result of a slightly more varied façade generation.

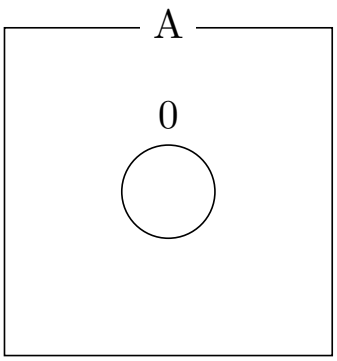

Figure 7.22: Axiom graph of the façade example.

Element	Attribute	Value
	`height`	25
0:	`label`	House
	`width`	40

Table 7.17: Attribute definitions of the axiom graph of the façade example.

`House` into multiple edges with the label `V-Floor`, standing for vertical floor piece. The resulting height of each floor piece lies between 10 and 20, with the remainder of the division of the attribute `height` by 10 being spread equally amongst all created floors. The exact calculation function for this can be seen in the definition of the attribute `.new_pos` of element **4** and of the attribute `height` of element **7** in Table 7.19.

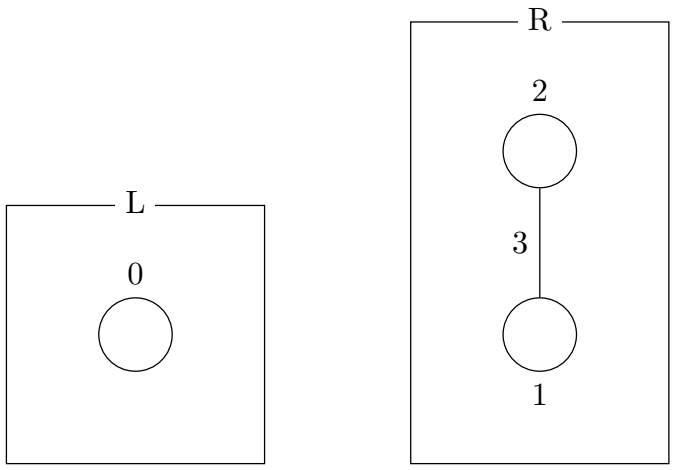

Figure 7.23: The production „Grow House“ of the façade example.

After this, the production „Grow Floor“ (Figure 7.25, Table 7.20) will extend the `V-Floor` edge into a horizontal edge with the label `H-Floor` of appropriate length and position. Following this the production „Add Features“ (Figure 7.26, Table 7.21) will split the horizontal floor apart into equal length chunks of `Feature` edges, similar to the „Split House to Floors“ production explained above. These features can the be used to place visual elements in a regular distance to each other.

Since this is only a simple example of the capabilities of this grammar, the only feature present is added in the production „Add Windows“ (Figure 7.27, Table 7.22). It will simply add a `Window` node in the centre of the `Feature` edge.

The node with the label `Window` is then used by the production „Expand Window“ (Figure 7.28, Table 7.23) to create an actual visual representation of a Window. It will replace the `Window` node and the two connected nodes with an unconnected subgraph representing a Window, placed and sized appropriately, depending on the positioning

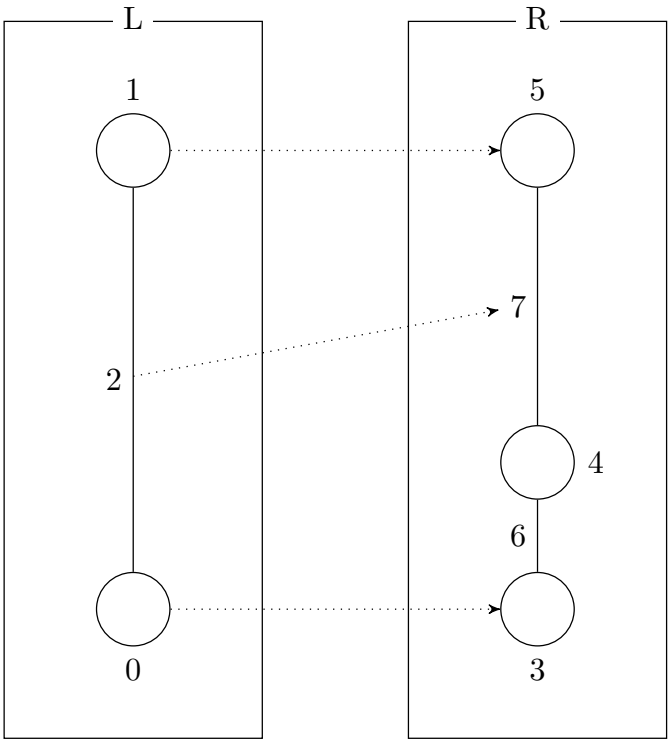

Figure 7.24: The production „Split House to Floors“ of the façade example.

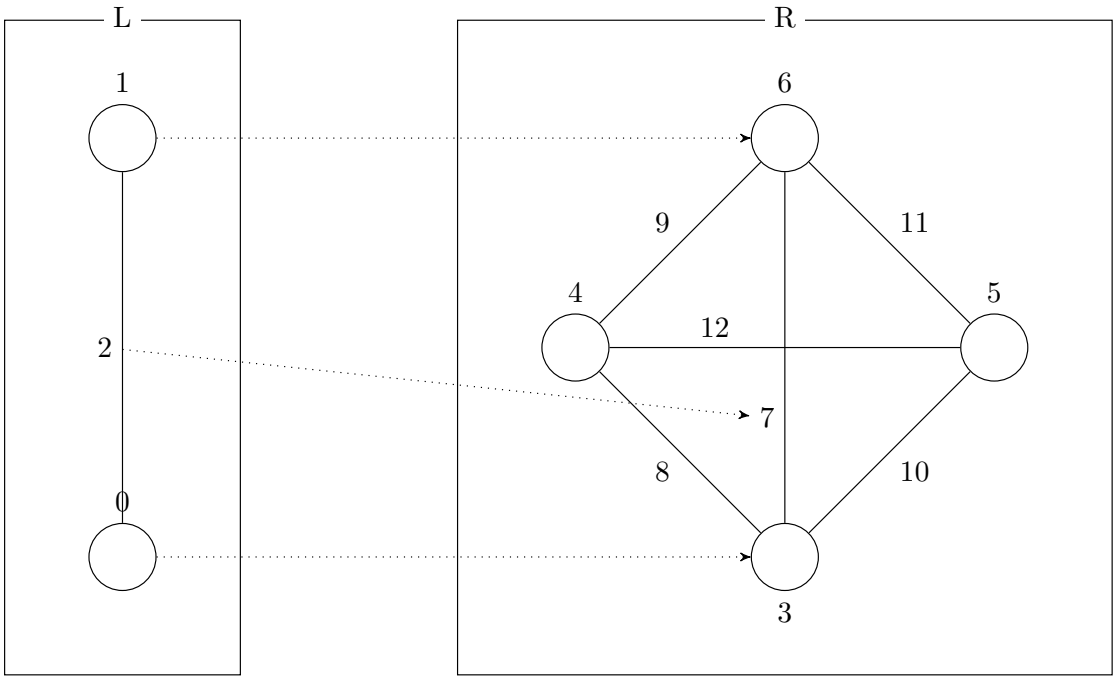

Figure 7.25: The production „Grow Floor“ of the façade example.

Element	Attribute	Value
0:	`label`	attr=='House'
1:	`.new_pos`	A
	`.svg_tag`	None
2:	`.new_pos`	A + Vec(x1=0, y1=**float**(house.attr['height']))
	`.svg_tag`	None
	`.svg_tag`	None
3:	`floors`	0
	`height`	house.attr['height']
	`label`	'House'
	`width`	house.attr['width']

Vector Name	Definition
A	Point **0**

Table 7.18: Attribute definitions of the production „Grow House“.

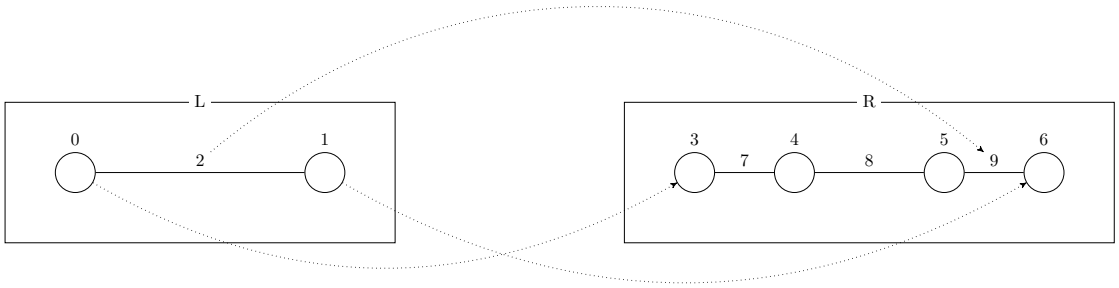

Figure 7.26: The production „Add Features“ of the façade example.

of the two nodes framing the `Window` node. The edges added by this production are the first elements which will actually appear in the SVG output of the derivation. In addition this production also calculates a random value between 0 and 1 and saves it in the attribute `blind_portion`. This attribute will later be used to define how much of the window is covered by a blind.

All windows extended by the production above will then be colored through the „Color Windows“ production (Figure 7.29, Table 7.24). It will place a single new node in the middle of the window with the SVG tag `rect` associated with it. Then the appropriate width and height of the rectangle are calculated based on the positioning of the windows vertices. In the SVG export this will result in a square of light blue colour filling the window.

The production „Expand Floors“ (Figure 7.30, Table 7.25) has a very similar function to that of the „Expand Window“ function, which is why they share a similar name. This

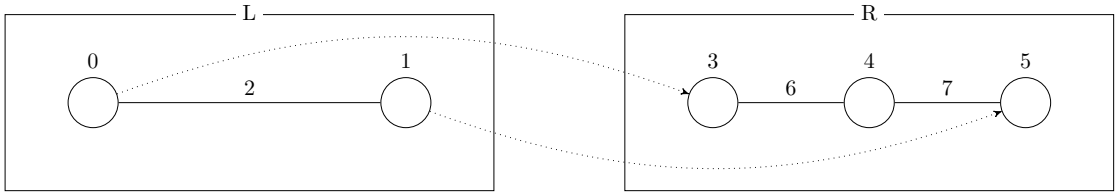

Figure 7.27: The production „Add Windows“ of the façade example.

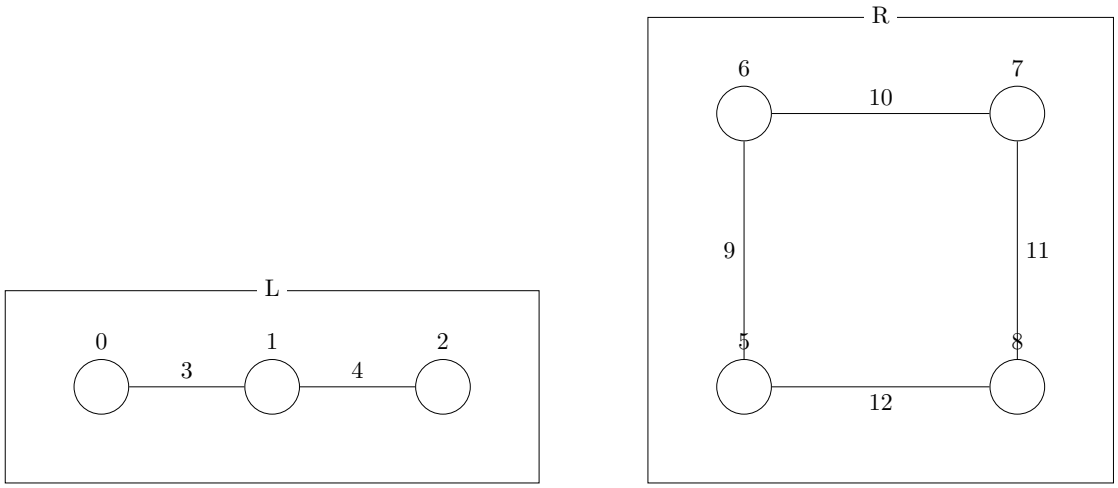

Figure 7.28: The production „Expand Windows“ of the façade example.

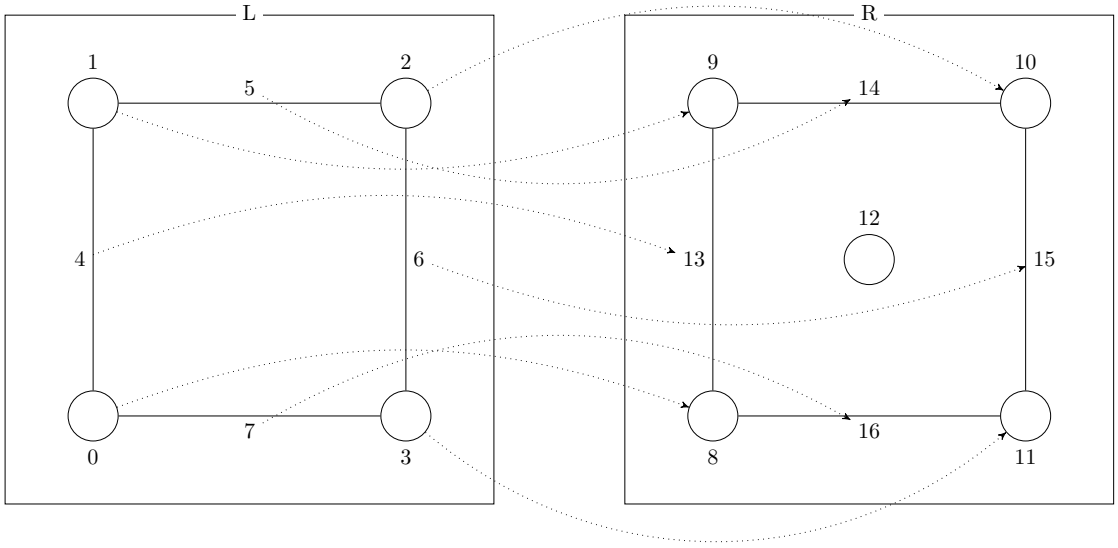

Figure 7.29: The production „Color Windows“ of the façade example.

Element	Attribute	Value
2:	`height`	**float**(attr) >= 10
	`label`	attr=='House'
3:	`.new_pos`	A
4:	`.new_pos`	A + normalize(v1)* (10 + (**float**(house.attr['height']) % 10) / (**float**(house.attr['height']) // 10))
	`.svg_tag`	None
5:	`.new_pos`	B
	`.svg_tag`	None
	`expanded`	False
6:	`floor`	house.attr['floors']
	`grown`	False
	`label`	'V−Floor'
	`width`	house.attr['width']
7:	`floors`	**int**(old.attr['floors'])+1
	`height`	**float**(old.attr['height']) − (10 + (**float**(old.attr['height']) % 10) / (**float**(old.attr['height']) // 10))

Vector Name	Definition
A	Point **0**
B	Point **1**
v1	Line from **0** to **1**

Table 7.19: Attribute definitions of the production „Split House to Floors“.

production will create a square, drawn in the SVG output, surrounding the entire floor. It does this by matching a floor by the four nodes which define the maximum extent of the floor in the horizontal and vertical directions. The actual position of the newly added elements is calculated automatically based on the positioning of the matched elements.

As a last step, blinds are added to the windows and then coloured in a beige tone. This is done in the productions „Add Blinds“ (Figure 7.31, Table 7.26) and „Color Blinds“ (Figure 7.32, Table 7.27). They are mostly equal to the productions adding and colouring a widow described before. The difference is that the degree to which the blinds cover the windows is random, based on the value `blind_portion` calculated in the „Extend Window“ production. The exact calculations can be found in the calculation of `.new_pos` in Table 7.26 for the elements **12** and **15**.

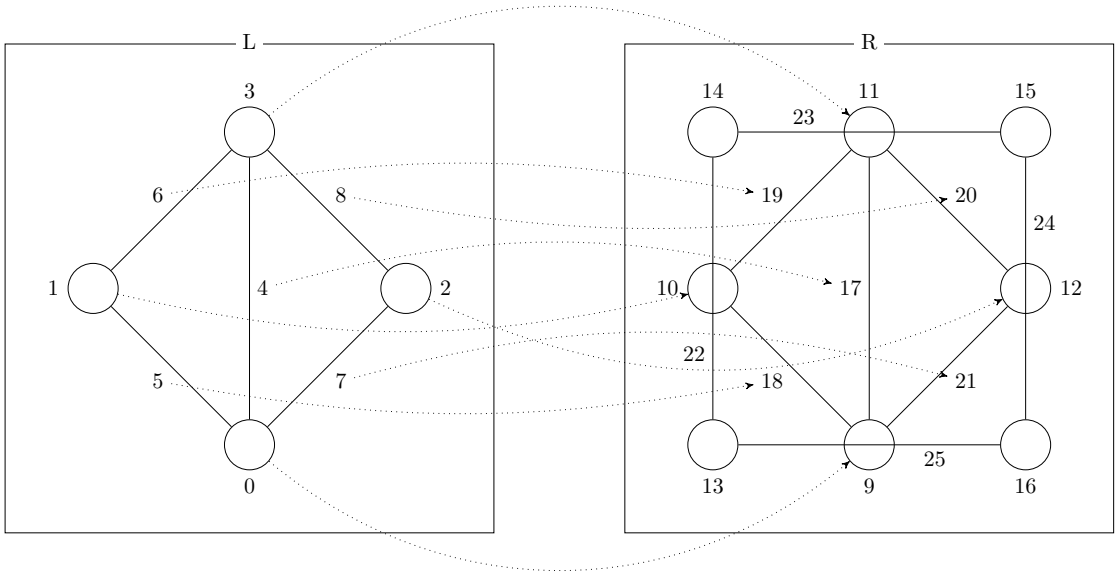

Figure 7.30: The production „Expand Floors“ of the façade example.

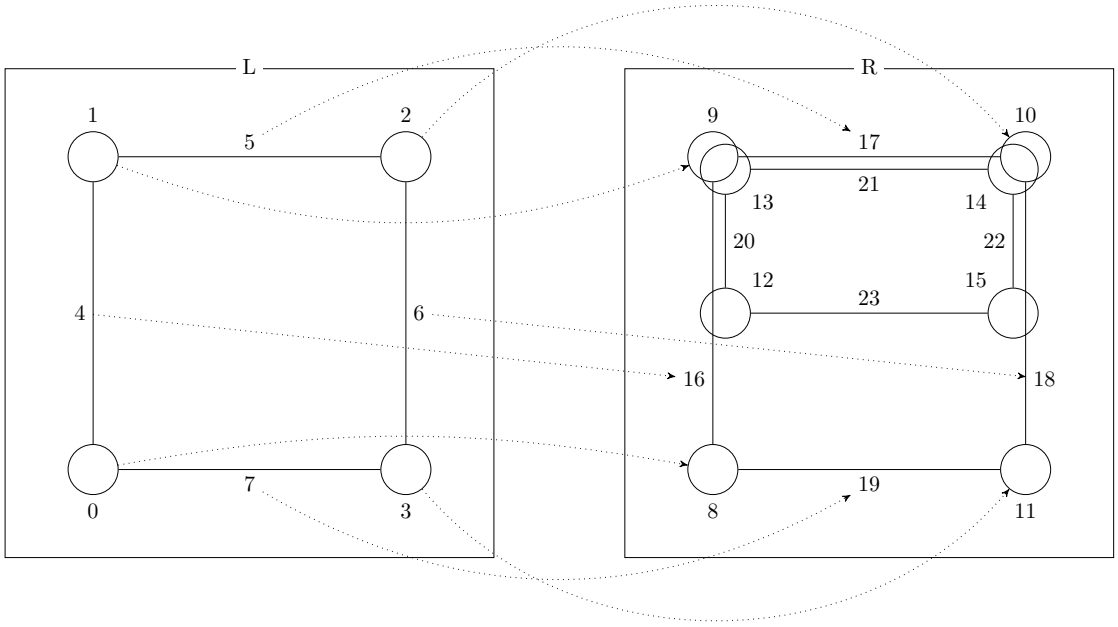

Figure 7.31: The production „Add Blinds“ of the façade example.

Element	Attribute	Value
2:	grown	**not** attr
	label	attr=='V−Floor'
4:	.svg_tag	None
	new_x	−**float**(v_floor.attr['width']) / 2
5:	.svg_tag	None
	new_x	**float**(v_floor.attr['width']) / 2
7:	grown	True
8-11:	.svg_tag	None
	.svg_tag	None
12:	floor	v_floor.attr['floor']
	label	'H−Floor'
	width	v_floor.attr['width']

Vector Name	Definition

Table 7.20: Attribute definitions of the production „Grow Floor“.

Element	Attribute	Value
2:	label	attr=='H−Floor'
	width	**float**(attr) >= 5
3:	.new_pos	A
4:	.new_pos	A
	.svg_tag	None
5:	.new_pos	A + normalize(v1)∗ (5 + **float**(h_floor.attr['width']) % 5)
	.svg_tag	None
7:	.svg_tag	None
8:	.svg_tag	None
	label	'Feature'
9:	width	**float**(h_floor.attr['width']) − (5 + **float**(h_floor.attr['width']) % 5)

Vector Name	Definition
A	Point **0**
v1	Line from **0** to **1**

Table 7.21: Attribute definitions of the production „Add Features“.

Element	Attribute	Value
2:	`label`	attr==’Feature’
4:	`.svg_tag`	None
	`label`	’Window’
6-7:	`.svg_tag`	None

Vector Name	Definition

Table 7.22: Attribute definitions of the production „Add Windows“.

Element	Attribute	Value
1:	`label`	attr==’Window’
5-8:	`.svg_tag`	None
9:	`label`	’Window’
10:	`blind_portion`	random.random()
	`has_blind`	False
	`is_colored`	False
	`label`	’Window’
11-12:	`label`	’Window’

Vector Name	Definition

Table 7.23: Attribute definitions of the production „Expand Windows“.

Element	Attribute	Value
4:	`label`	attr==’Window’
5:	`is_colored`	**not** attr
	`label`	attr==’Window’
6-7:	`label`	attr==’Window’
12:	`.svg_fill`	’#bbe5ff’
	`.svg_height`	norm(v1)
	`.svg_tag`	’rect’
	`.svg_width`	norm(v2)
14:	`is_colored`	True

Vector Name	Definition
v1	Line from **1** to **0**
v2	Line from **1** to **2**

Table 7.24: Attribute definitions of the production „Color Windows“.

Element	Attribute	Value
4:	`expanded`	**not** attr
	`label`	attr=='V−Floor'
13-16:	`.svg_tag`	None
17:	`expanded`	True

Vector Name	Definition

Table 7.25: Attribute definitions of the production „Expand Floors“.

Element	Attribute	Value
4:	`label`	attr=='Window'
5:	`has_blind`	**not** attr
	`label`	attr=='Window'
6-7:	`label`	attr=='Window'
12:	`.new_pos`	A + Vec(x1=0.2, y1=0)+ v1 ∗ **float**(top_edge.attr['blind_portion'])
	`.svg_tag`	None
13:	`.new_pos`	A + Vec(x1=0.2, y1=−0.05)
	`.svg_tag`	None
14:	`.new_pos`	B + Vec(x1=−0.2, y1=−0.05)
	`.svg_tag`	None
15:	`.new_pos`	B + Vec(x1=−0.2, y1=0)+ v2 ∗ **float**(top_edge.attr['blind_portion'])
	`.svg_tag`	None
17:	`has_blind`	True
20:	`label`	'Blind'
21:	`label`	'Blind'
	`is_colored`	False
22-23:	`label`	'Blind'

Vector Name	Definition
A	Point **1**
B	Point **2**
v1	Line from **1** to **0**
v2	Line from **2** to **3**

Table 7.26: Attribute definitions of the production „Add Blinds“.

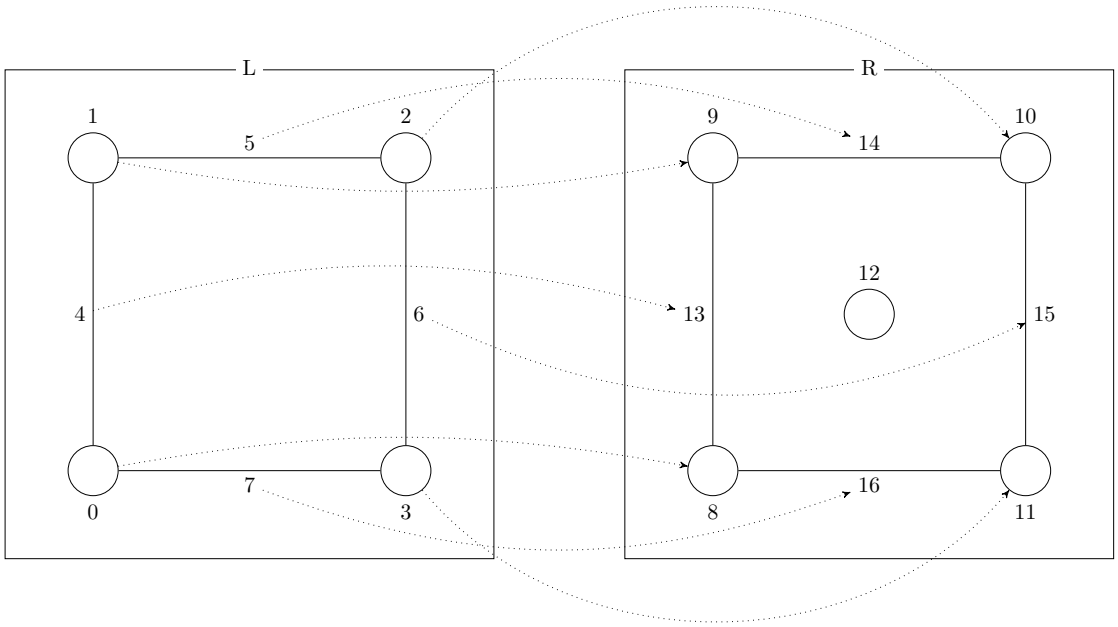

Figure 7.32: The production „Color Blinds“ of the façade example.

Element	Attribute	Value
4:	`label`	attr==’Blind’
5:	`label`	attr==’Blind’
	`is_colored`	**not** attr
6-7:	`label`	attr==’Blind’
12:	`.svg_fill`	’#f6feba␣’
	`.svg_height`	norm(v1)
	`.svg_tag`	’rect’
	`.svg_width`	norm(v2)
14:	`is_colored`	True

Vector Name	Definition
v1	Line from **1** to **0**
v2	Line from **1** to **2**

Table 7.27: Attribute definitions of the production „Color Blinds“.

CHAPTER 8

Conclusion and Future Work

Given the level of expressiveness available to graph grammars and past works on specialised applications such as McDermott [McD13] for geometric objects and Kniemeyer [Kni08] or Henke, Kniemeyer, and Kurth [HKK17] for plants, the ability of graph grammars as a formalism to be used for general purpose procedural modelling was never in question. What remained to be answered is whether or not a single class of graph grammars could be an effective tool for all these purposes.

In this work it was shown that a single class of graph grammars can indeed cover disparate types of procedural modelling tasks effectively. This has the potential to increase the flexibility of modelling tools, no longer requiring the use of multiple separate toolchains to combine e.g. houses with foliage. Beyond that, a novel GUI for the creation of production rules, specialized for procedural modelling was introduced. As far as the author is aware, no other such GUI integrating the on-screen position of elements as part of the production definition process, has been proposed.

Another contribution is the application of Logo-like rules directly on graphs which represent geometric information of objects. Previous works on emulating and extending the functionality of L-systems with graph grammars focused on creating a graph containing the same information as is encoded in the result string of an L-system. To produce a visualisation, this graph would then have to be interpreted with methods similar to L-systems, such as a Logo-style turtle. Contrary to the above approach, the solution in this work has the visual and geometric information already present in the result graph, without requiring additional interpretation. To set itself apart from a previous attempt at the same goal in Christiansen and Bærentzen [CB13], the new graph grammar proposed in this work also supports the use of semantic information during the derivation process and the use of completely abstract graphs, whenever such an approach is found to be more suitable.

The main limitations of the current implementation are the lack of optimization, which limits the possibility of applying it to the creation of larger and more detailed models, and the unpolished state of the UI. As the UI was neither the sole nor a major objective of this work and the scope of a bachelor thesis has to be limited, a relatively simple GUI was implemented. However, through working with the GUI on producing the examples, the author has come to the believe that there is a lot of potential to ease the use of graph grammars and improve their intuitiveness, by giving appropriate visual feedback.

This leads to a discussion of future work, where there are two main directions of inquiry. The first is how the speed of the matching process can be improved to enable the generation of large scale models, such as entire cities within a reasonable time-frame. There is much active research on the subject of efficient subgraph isomorphism algorithms, as they find applicability in a wide variety of problems. For the subject of procedural modelling, it would be of particular interest to investigate which types of graphs are typically encountered and therefore which subgraph isomorphism algorithms are best suited for this type of application. In addition to investigating general purpose algorithms, one should also consider specialised approaches, such as defining regions within a graph, which would act like boundaries for the matching algorithm. For example, one could imagine that, when modelling an entire city, it would be sensible to restrict the matching algorithm of a tree production to a single plot of land, where the tree will be grown. Such an approach, if successful, could greatly reduce the search space of the graph matching algorithm. The second direction of inquiry are further improvements to the functionality of the graph grammar proposed in this work. Due to the limited scope, it only contains the basic necessities and there are many extensions which promise interesting results and new applications. A small list of possibilities the author has considered is:

- A 3rd dimension.
- Faces and volumes as graph elements. This could then allow for natural modelling of clipping restrictions.
- A derivation hierarchy which could be queried within productions.
- Automatic level of detail control for distant models.

List of Figures

List of Tables

List of Algorithms

Glossary

G^3 Generic Graph Grammar 13

CESCG Central European Seminar on Computer Graphics vii, ix

DPO double push out. One of two different approaches of defining productions in an algebraic graph grammar. 12, 69

GML Generative Modeling Language 5

SPO single push out. One of two different approaches of defining productions in an algebraic graph grammar. 12, 13, 69

SVG Scalable Vector Graphics 5, 29, 31, 35, 37, 41–43, 45, 46, 48, 49, 58, 60

YAML YAML Ain't Markup Language. A file-format for data serialisation with functionality similar to JSON, but with different syntax. 29, 31

Bibliography

[Car+18] V. Carletti, P. Foggia, A. Saggese, and M. Vento. „Challenging the Time Complexity of Exact Subgraph Isomorphism for Huge and Dense Graphs with VF3“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40.4 (Apr. 2018), pp. 804–818.

[CB13] A. N. Christiansen and J. A. Bærentzen. „Generic Graph Grammar: A Simple Grammar for Generic Procedural Modelling“. In: *Proceedings of the 28th Spring Conference on Computer Graphics*. SCCG '12. Smolenice, Slovakia: ACM, 2013, pp. 85–92.

[Cor+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. „Algebraic Approaches to Graph Transformation. Part I: Basic Concepts and Double Pushout Approach“. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Ed. by G. Rozenberg. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997, pp. 163–245.

[Ehr+06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[Ehr+99a] H. Ehrig, G. Engels, H. J. Kreowski, and G. Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999.

[Ehr+99b] H. Ehrig, H. J. Kreowski, U. Montanari, and G. Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 3: Concurrency, Parallelism, and Distribution*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999.

[EPS73] H. Ehrig, M. Pfender, and H. J. Schneider. „Graph-grammars: An Algebraic Approach“. In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*. SWAT '73. Washington, DC, USA: IEEE Computer Society, 1973, pp. 167–180.

[ER97] J. Engelfriet and G. Rozenberg. „Node Replacement Graph Grammars“. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Ed. by G. Rozenberg. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997, pp. 1–94.

[FB92] H. Fahmy and D. Blostein. „A survey of graph grammars: theory and applications“. In: *Pattern Recognition, 1992. Vol.II. Conference B: Pattern Recognition Methodology and Systems, Proceedings., 11th IAPR International Conference on*. Aug. 1992, pp. 294–298.

[FYA10] D. Fletcher, Y. Yue, and M. Al Kader. „Challenges and Perspectives of Procedural Modelling and Effects“. In: *Information Visualisation (IV), 2010 14th International Conference*. July 2010, pp. 543–550.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman and Co., 1979.

[HF11] S. Havemann and D. W. Fellner. „Towards a New Shape Description Paradigm Using the Generative Modeling Language“. In: *Rainbow of Computer Science - Dedicated to Hermann Maurer on the Occasion of His 70th Birthday*. Ed. by C. S. Calude, G. Rozenberg, and A. Salomaa. Vol. 6570. Lecture Notes in Computer Science. Springer, 2011, pp. 200–214.

[HKK17] M. Henke, O. Kniemeyer, and W. Kurth. „Realization and Extension of the Xfrog Approach for Plant Modelling in the Graph-Grammar Based Language XL“. In: *Computing and Informatics* 36.1 (2017), pp. 33–54.

[Hon71] H. Honda. „Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body“. In: *Journal of Theoretical Biology* 31.2 (1971), pp. 331–338.

[HS08] H. He and A. K. Singh. „Graphs-at-a-time: Query Language and Access Methods for Graph Databases“. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 405–418.

[JCS16] D. Jesus, A. Coelho, and A. A. Sousa. „Layered shape grammars for procedural modelling of buildings“. In: *The Visual Computer* 32 (2016), pp. 933–943.

[Kni08] O. Kniemeyer. „Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling“. PhD thesis. Brandenburgische Technische Universität Cottbus, 2008.

[Koc06] H. Koch. „Une méthode géométrique élémentaire pour létude de certaines questions de la théorie des courbes planes“. In: *Acta Math.* 30 (1906), pp. 145–174.

[KPK10] L. Krecklau, D. Pavic, and L. Kobbelt. „Generalized Use of Non-Terminal Symbols for Procedural Modeling“. In: *Computer Graphics Forum* 29.8 (2010), pp. 2291–2303.

[McD13] J. McDermott. „Graph grammars for evolutionary 3D design“. In: *Genetic Programming and Evolvable Machines* 14.3 (Sept. 2013), pp. 369–393.

[Mül+06] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. „Procedural Modeling of Buildings“. In: *ACM Trans. Graph.* 25.3 (July 2006), pp. 614–623.

[Nag79] M. Nagl. „A tutorial and bibliographical survey on graph grammars“. In: *Graph-Grammars and Their Application to Computer Science and Biology*. Ed. by V. Claus, H. Ehrig, and G. Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 70–126.

[ODA03] A. Ortega, A. L. A. Dalhoum, and M. Alfonseca. „Grammatical Evolution to Design Fractal Curves with a Given Dimension“. In: *IBM J. Res. Dev.* 47.4 (July 2003), pp. 483–493.

[Par93] F. Parisi-Presicce. „Single vs. double pushout derivations of graphs“. English. In: *Graph-Theoretic Concepts in Computer Science*. Ed. by E. W. Mayr. Vol. 657. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1993, pp. 248–262.

[PL96] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. New York, NY, USA: Springer-Verlag New York, Inc., 1996.

[PM01] 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. New York, NY, USA: ACM, 2001, pp. 301–308.

[PR69] J. L. Pfaltz and A. Rosenfeld. „Web Grammars“. In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI'69. Washington, DC: Morgan Kaufmann Publishers Inc., 1969, pp. 609–619.

[Roz97] G. Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997.

[Rud97] M. Rudolf. „Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation“. MA thesis. Fachbereich Informatik der Technischen Universität Berlin, 1997.

[Sch70] H. J. Schneider. „Chomsky-Systeme für partielle Ordnungen“. In: *Arbeitsberichte des Inst. f. Math. Masch. u. Datenver.* 3.3 (1970).

[SM15] M. Schwarz and P. Müller. „Advanced Procedural Modeling of Architecture“. In: *ACM Trans. Graph.* 34.4 (July 2015), 107:1–107:12.

[Sti+71] G. Stiny, J. Gips, G. Stiny, and J. Gips. „Shape Grammars and the Generative Specification of Painting and Sculpture“. In: *Segmentation of Buildings for 3DGeneralisation. In: Proceedings of the Workshop on generalisation and multiple representation , Leicester*. 1971.

[Tha+13] W. Thaller, U. Krispel, R. Zmugg, S. Havemann, and D. W. Fellner. „Shape grammars on convex polyhedra“. In: *Computers & Graphics* 37.6 (2013). Shape Modeling International (SMI) Conference 2013, pp. 707–717.

[Won+03] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. „Instant Architecture“. In: *ACM Trans. Graph.* 22.3 (July 2003), pp. 669–677.