A Graph Grammar for Modelling of 2D Shapes

Viktor Pogrzebacz*
Supervised by: Martin Ilčík†

Institute of Visual Computing & Human-Centered Technology
Vienna University of Technology
Vienna / Austria

Abstract

The creation of detailed models for computer graphics is a work intensive task, which limits projects in the graphical fidelity which can be achieved. Procedural modelling is an ongoing field of research which aims to alleviate this pressure. The most common systems for procedural modelling specialize in either modelling plants, as is the case with L-systems, or in modelling buildings, as do shape grammars. This paper aims to show a path for improving this situation, by describing the conception and implementation of a graph grammar for procedural modelling of both artificial (buildings and furniture) and organic (trees and flowers) objects in 2D space. The suitability of the proposed graph grammar is demonstrated by applying it to a variety of modelling tasks, such as a Koch snowflake, circular and square patterns, foliage and a building façade.

Keywords: graph grammar, procedural generation, modelling

1 Introduction

Modelling of detailed objects is a subject of growing importance, especially in the entertainment industry. Procedural modelling can play an important role in this process, by providing a means of creating detailed and varied models in a shortened time-frame. It has the potential to add more objects, and more variety in the same class of objects, without requiring more development resources.

Most methods of procedural modelling used today have a particular type of object they are good at producing. For example, shape grammars are used very successfully to create building façades, houses and cities, but they are not very successful at creating plants. L-systems mirror this state, being very good at modelling plants, but facing noticeable limitations when applied to modelling buildings.

This paper attempts to show how graph grammars can be used to model all kinds of objects, be they naturally grown or artificially crafted.

The next section describes the relevant literature in the fields of procedural modelling and graph grammars (Section 2). Afterwards, the next two sections give a formal definition of the new grammar proposed in this paper and a description of its actual implementation in Python (sections 3 and 4). Then, Section 5 will show how a production is defined and display the results of various derivations. Finally, Section 6 contains conclusions and a discussion on drawbacks and future work.

2 Related Works

A comprehensive introduction to L-systems and their application to the modelling of plants can be found in [17]. In general, models of plants are generated from the string produced by an L-system by interpreting them with a Logo-style turtle. An example of such an L-system for a simple 2D plant-like structure would be the following (taken from [17]):

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

The set of rules would be 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. 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 color of line segment.

L-systems are, however, not limited to the generation of plant models, they can also very successfully create fractal patterns and have even been applied to the modelling of buildings and cities [15].

For the modelling of buildings and façades, an approach based on shape grammars was developed since the turn of the millennium. The term shape grammar is somewhat unfortunate, because what it commonly refers to is not technically a shape grammar, as defined by [20], but a subset of set grammars.

*viktor@agerekg.at
†ilcik@cg.tuwien.ac.at

The split approach to shape grammars is the most common in procedural modelling. It was introduced by [22], inspired by L-Systems and the work done in [15]. Since then it has been improved and extended by multiple publications, such as the addition of scopes in [13], extending the use of non-terminal shapes in [11], allowing for a shapes scope to be any convex polyhedra in [21], allowing conditions on geometric information within productions in [19], and allowing for layers and SVG code inside 2D shapes in [8].

A typical example of a production in a shape grammar, taken from [13], would be:

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

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 [13] 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.

A different, but somewhat related approach to shape grammars is GML [5]. It is a stack-based imperative programming language, mirroring the syntax of PostScript, 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.

As to the subject of graph grammars, which are the underpinning of this paper, the standard work is the "Handbook of Graph Grammars and Computing by Graph Transformation" in three volumes [18, 3, 4]. A detailed, specialized treatment of double push out algebraic graph grammars can be found in [2]. A comparison between the double and the single push out approaches to algebraic graph grammars can be found in [16].

Applications of graph grammars to procedural modelling are not yet particularly well explored. There is [1], which describes a graph grammar for general purpose procedural modelling, but deviates from the usual practice of having labelled graph elements and places big limits on what productions can look like. For plants there are [9], which mirrors L-systems using graph grammars, and [6], which is an extension of the graph based XFrog software. Another work of interest is [12], which uses graph grammars for evolutionary design. So there is active research in this area, but as of yet there is no generally agreed upon approach to using graph grammars for procedural modelling, as is the case with L-systems or shape grammars.

3 Formal Definition

The proposed *grammar* is a tuple of the form (A, P) where A is the *axiom graph*, or the 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, 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.

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 is the partial graph morphism.

The *partial graph morphism* p is a morphism of some subgraph of M to D, i.e. $S \subset M$, $p : S \rightarrow D$. 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 new 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, i.e. a node may only be mapped to a node, an edge to an edge and so on.

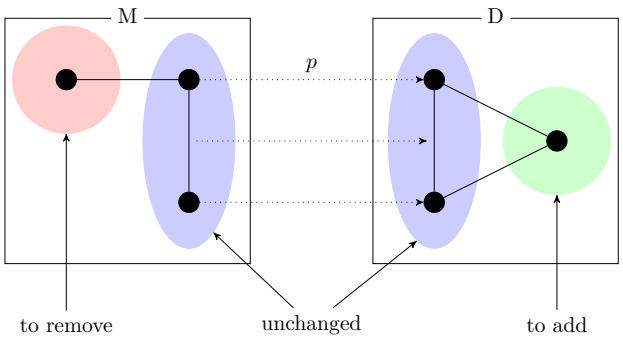

Figure 1: Example of a production definition. The dotted arrows going from M to D represent the partial graph morphism p. Elements which will be added are circled in red, elements to be added in green and elements kept unchanded in blue.

Figure 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 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 an 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 2, which applies the production from Figure 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.

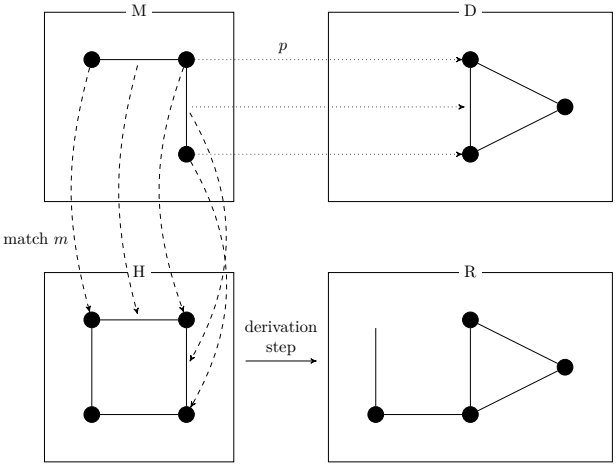

Figure 2: An example application of the rule from Figure 1. The dashed lines show the match found between M and H. R shows the result graph of applying the production.

A *derivation* is a sequence of derivation steps, defined in the usual manner. A derivation ends and produces a terminal graph when none of the productions in P can be applied. This definition allows for the creation of infinite derivations. This is not seen as an issue since, in practical application, the number of derivation steps will always need to be bound, because of factors such as computer memory or time constraints.

The above definition marks the introduced grammar as an algebraic approach graph grammar, using the gluing approach to embedding. 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 production, 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 [2].

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.

3.1 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 in which they are applied 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 only be applied 100 times, following which productions of priority one will be applied up to 400 times. 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 5.1 and the tree models of Section 5.3.

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 $(M, \{(p, D, w)\})$ where 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 production level by setting the priority to choose the match with the oldest elements.

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.

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 than fixed values. These calculation instructions have access to any attribute values of any element matched in H, i.e. to any element of the set $\{m(e) | e \in M \wedge e \in dom(p)\}$.

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, can be queried in the mother graph and have new values 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 5.1.

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

3.2 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 of the different means by which variation between results can be achieved in the proposed grammar.

There are two types of variations 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 calculating attributes that no additional functionality was necessary.

As for structural variations, there are multiple ways by which they 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 this priority from being applied to the graph, then this can be used to produce structural variations. In addition to this method, it is possible to define a single mother graph with multiple daughter graphs, each of which have a weight attached to them. 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.

4 Implementation

In this section the implementation and the relevant algorithms used therein are briefly discussed. The grammar

was implemented in Python and the source code can be found online[1]. It is split apart into a graph grammar library doing all the calculations and a GUI used to interact with the library.

4.1 Matching Algorithm

Finding isomorphisms of two graphs is an NP-hard problem. 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 going along the graph structure of the mother graph, trying to find a match for each element using a depth first approach. 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 is certainly possible and even necessary for working with the grammar on a greater scale, but outside the scope of the current work.

4.2 Replacement Algorithm

The replacement algorithm follows, as stated before, a single push-out approach. In the implementation it uses five graphs with mappings between them to apply a production.

The basic steps taken to apply a production, after a match has been found and selected, are:

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.

3. Add the new elements, which will automatically connect them as necessary.

4. Calculate the new value of attributes for those elements where this is necessary. This includes calculating the new position of elements, if no user-defined calculation instructions are present.

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 3. M and D are the mother or left-hand-side and daughter or right-hand-side graphs of the production. The mapping between them is decided 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 partial isomorphism as described in the Section 4.1. 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. When a production adds a new element to the result graph R, the element is simply moved from C to R and referenced in neighbourhood lists appropriately. This way there is no need for a special function to create new elements, while copying only the relevant attributes and also calculating the correct new position.

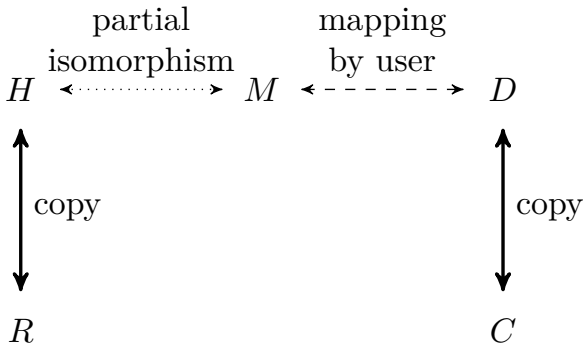

Figure 3: Overview of the hierarchy of graphs in a production application. M and D are the left- and right-hand sides of the production. H is the hostgraph to which the production is applied. R is a deep-copy of H and C a deep-copy of D. R becomes the result graph when all elements to be added are moved from C to R.

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 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 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 are deleted.

4. Elements of D without a mapping from M to D are added to the result. References to elements from point 2 are translated to references to elements in R, the copy of H.

[1] https://github.com/D4id4los/Python-Graph-Grammar

4.3 Calculation of Attributes and Position of Elements

The attribute calculation instructions can be arbitrary Python instructions allowing, in particular, to use 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 based on nodes, which will be available for use in the argument calculation formulas. For an example see Section 5.

The automatic calculation of new positions currently calculates the barycentre of the daughter graph D and the partial 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 depending on what subgraph of H they are matched to.

4.4 Export

The derivation result of the grammar can be saved to a YAML 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 5.

5 Examples

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 to which either L-systems or shape grammars find the most successful application. Due to space considerations the examples only constitute a showcase of the new grammars capabilities, the complete definitions can be found online as YAML files[1].

For each of the examples a derivation runtime is given. To inform interpretation of that number it is important to keep in mind that no performance optimisation was undertaken.

5.1 Modelling of a Koch Snowflake

The Koch Snowflake or Koch curve defined by Helge von Koch in [10] is a classical task in procedural modelling. 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
Production rules:
F -> F+F--F+F
+ -> +
- -> -
```

Which is then interpreted either by a Logo 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 [14].

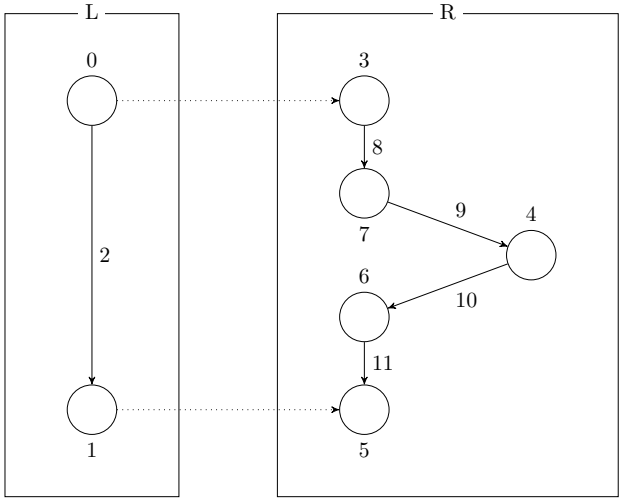

Figure 4: The single production of the Koch Snowflake derivation.

To define a Koch Snowflake in this grammar a different approach was chosen. Starting with a graph in the shape of an equilateral triangle the production presented in Figure 4 is applied to every edge of the graph (or triangle). To simplify the calculation of the new positions of the nodes two vectors are defined. **A**, a point vector at the position of node 3, and **v1**, a directional vector going from the node 3 to the node 5. With that the function to calculate the new position of node 4 becomes `A + v1 / 2`

`+ 1/3*(perp_left(v1))`, where `perp_left` is a function returning a vector perpendicular and pointing to the left of a directional vector. This calculation instruction is saved as an attribute of the vertex with the name `.new_pos`.

The final result is shown in Figure 5. It is produced after 192 derivation steps, taking an average of 9 seconds. The result graph contains 1158 elements.

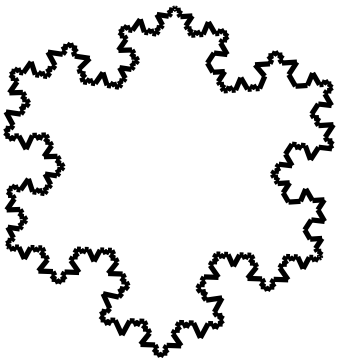

Figure 5: Result of a Koch Snowflake derivation. ~9 seconds runtime, 192 derivation steps resulting in 1158 elements.

Of particular note is how this production mimics the functionality of a string or graph interpreted by a Logo turtle, but makes these changes directly on the geometrical structure. 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, as where discussed above. Contrary to the above approach, the solution in this example has the visual and geometric information already present in the result graph, without requiring additional interpretation.

5.2 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.

The ability of the proposed grammar to introduce randomness into the result of production is shown in both the square pattern in Figure 6 and the circular pattern of Figure 7. The circular pattern additionally displays the flexibility

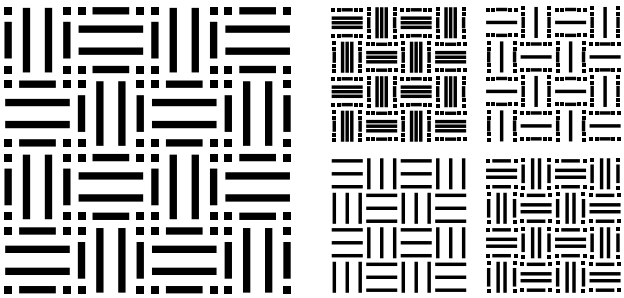

Figure 6: Different variation of a tiling square pattern generation. Average 12 seconds runtime, 128 derivation steps resulting in an average of 294 elements.

gained by allowing the export of arbitrary SVG tags, making use of the `<path>` tag to export Bézier curves. The square patterns are the result of 9 rules taking an average of twelve seconds to complete the derivation consisting of an average of 128 steps, producing 295 graph elements. The circle patterns only require four rules and the derivations are very fast at a tenth of a second for 28 derivation steps. The resulting graphs average 20 elements.

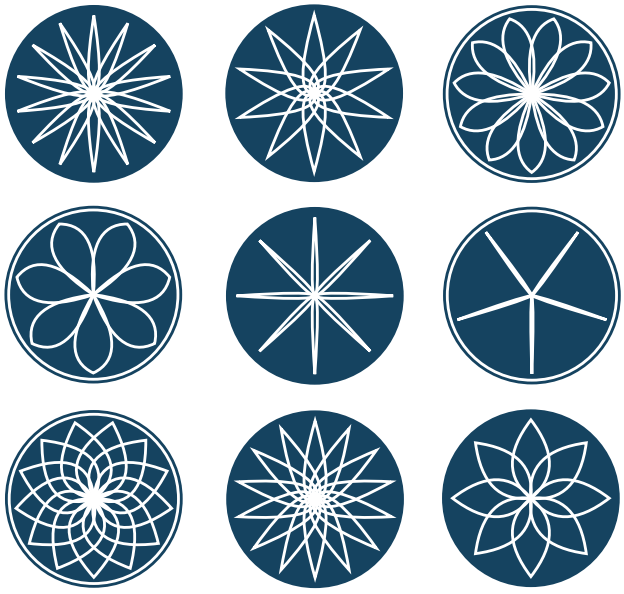

Figure 7: Nine different results of deriving a circular pattern. Average 0.1 seconds runtime, 28 derivation steps resulting in about 20 graph elements.

5.3 Modelling of a Tree

As part of the effort to show the versatility of the graph grammar it was applied to the modelling of trees shown in Figure 8. This grammar consists of two rules, which when left to run for 200 steps produce 803 graph elements in an average of five seconds.

The productions used in creating these trees where inspired by and adapted from the descriptions of trees by [7]

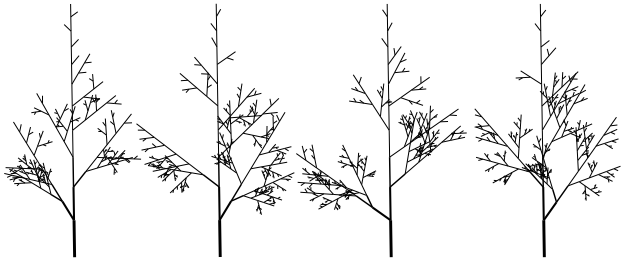

Figure 8: Four possible results of running the tree derivation. Average 5 seconds runtime, 200 derivation steps resulting in 803 graph elements.

as demonstrated in [17]. It is a fairly simple set of two productions which grow the tree while decreasing the width and length of additional segments with each steps.

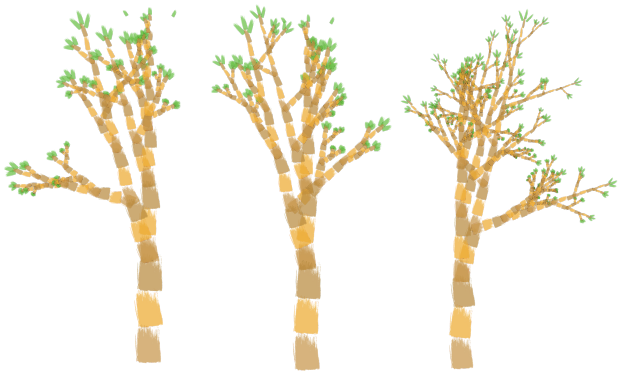

Figure 9: Three results of the painted tree derivation.

Using the SVG capability of adding images with filters to the output, it is possible to create a stylised visualisation of the tree generation as seen in Figure 9. This result was obtained by adding two additional productions which are run after the trees seen in Figure 8 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 image of a black brush stroke added as an SVG `image` element to the export. The different colours are obtained by applying various `feColorMatrix` filters.

5.4 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 [13]. 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 places [8].

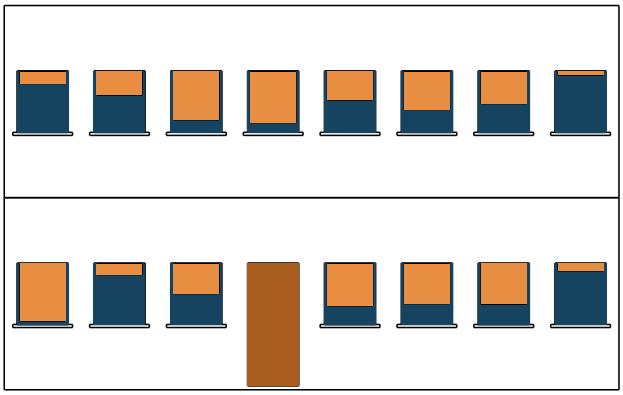

Figure 10: Result of a very simple building façade generation. Average 11 seconds runtime, 116 derivation steps ressulting in 468 graph elements.

The result of a simple example of using the proposed graph grammar to model a building's façade is shown in Figure 10. It is only a simplified view of a façade with two floors, the windows having blinds which are placed at random states of unrolling. For this result fourteen rules where used, performing 116 derivation steps in about 11 seconds to produce a graph with 468 elements.

6 Conclusio and Future Work

Given the level of expressiveness available to graph grammars and past works on specialised applications, such as [12] for geometric objects and [9] or [6] for plants, the ability of graph grammars to be used for general purpose procedural modelling was never in question. What remains to be answered is whether or not a single class of graph grammars can 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 tool-chains to combine e.g. houses with foliage. Another contribution is the application of Logo-like rules directly on graphs which represent geometric information of obreadjects, as discussed in Section 5.1.

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 state of the UI. As the creation of the GUI was not the sole nor the prime objective of this work, a relatively simple GUI sufficient for this work 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. Aside from better matching algorithms and low-level optimization of the matching code, coupled with a rewrite in a language like C++, one should also consider additional approaches, such as defining subgraphs which would act like boundaries for the matching algorithm, so that it does not need to match against the entire graph at all times. The second direction of inquiry are further improvements to the graph grammar proposed in this work. Due to the limited scope it only contains basic functionality 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
- A derivation hierarchy which could be queried within productions
- Automatic level of detail control for distant models

References

[1] Asger Nyman Christiansen and Jakob Andreas Bærentzen. Generic graph grammar: A simple grammar for generic procedural modelling. In *Proceedings of the 28th Spring Conference on Computer Graphics*, SCCG '12, pages 85–92, New York, NY, USA, 2013. ACM.

[2] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[3] Hartmut Ehrig, Gregor Engels, Hans Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[4] Hartmut Ehrig, Hans Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 3: Concurrency, Parallelism, and Distribution*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[5] Sven Havemann and Dieter W. Fellner. Towards a new shape description paradigm using the generative modeling language. In Cristian S. Calude, Grzegorz Rozenberg, and Arto Salomaa, editors, *Rainbow of Computer Science - Dedicated to Hermann Maurer on the Occasion of His 70th Birthday*, volume 6570 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011.

[6] Michael Henke, Ole Kniemeyer, and Winfried Kurth. Realization and extension of the xfrog approach for plant modelling in the graph-grammar based language XL. *Computing and Informatics*, 36(1):33–54, 2017.

[7] Hisao 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. *Journal of Theoretical Biology*, 31(2):331 – 338, 1971.

[8] Diego Jesus, António Coelho, and António Augusto Sousa. Layered shape grammars for procedural modelling of buildings. *The Visual Computer*, 32:933–943, 2016.

[9] Ole Kniemeyer. *Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling*. PhD thesis, Brandenburgische Technische Universität Cottbus, 2008.

[10] Helge Koch. Une méthode géométrique élémentaire pour létude de certaines questions de la théorie des courbes planes. *Acta Math.*, 30:145–174, 1906.

[11] Lars Krecklau, Darko Pavic, and Leif Kobbelt. Generalized use of non-terminal symbols for procedural modeling. *Computer Graphics Forum*, 29(8):2291–2303, 2010.

[12] James McDermott. Graph grammars for evolutionary 3d design. *Genetic Programming and Evolvable Machines*, 14(3):369–393, Sep 2013.

[13] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, July 2006.

[14] Alfonso Ortega, Abdel Latif Abu Dalhoum, and Manuel Alfonseca. Grammatical evolution to design fractal curves with a given dimension. *IBM J. Res. Dev.*, 47(4):483–493, July 2003.

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

[16] Francesco Parisi-Presicce. Single vs. double pushout derivations of graphs. In Ernst W. Mayr, editor, *Graph-Theoretic Concepts in Computer Science*, volume 657 of *Lecture Notes in Computer Science*, pages 248–262. Springer Berlin Heidelberg, 1993.

[17] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

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

[19] Michael Schwarz and Pascal Müller. Advanced procedural modeling of architecture. *ACM Trans. Graph.*, 34(4):107:1–107:12, July 2015.

[20] George Stiny, James Gips, George Stiny, and James 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.

[21] Wolfgang Thaller, Ulrich Krispel, René Zmugg, Sven Havemann, and Dieter W. Fellner. Shape grammars on convex polyhedra. *Computers & Graphics*, 37(6):707 – 717, 2013. Shape Modeling International (SMI) Conference 2013.

[22] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Trans. Graph.*, 22(3):669–677, July 2003.