EUROGRAPHICS 2017 / L. Barthe and B. Benes
(Guest Editors)

Volume 36 (2017), Number 2

Design Transformations for Rule-based Procedural Modeling

Stefan Lienhard[1] Cheryl Lau[2] Pascal Müller[2] Peter Wonka[3] Mark Pauly[1]

[1]EPFL [2]Esri R&D Center Zurich [3]KAUST

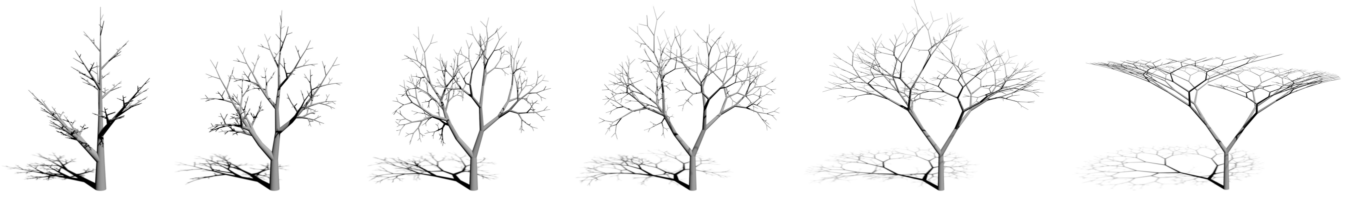

Figure 1: ***Tree L-Systems*** *A design transformation computed from two different L-systems (first and last images). Our method can compute a sequence of intermediate results transforming one design into the other by combining and merging the L-systems rather than the geometry of the two trees.*

Abstract

We introduce design transformations for rule-based procedural models, e.g., for buildings and plants. Given two or more procedural designs, each specified by a grammar, a design transformation combines elements of the existing designs to generate new designs. We introduce two technical components to enable design transformations. First, we extend the concept of discrete rule switching to rule merging, leading to a very large shape space for combining procedural models. Second, we propose an algorithm to jointly derive two or more grammars, called grammar co-derivation. We demonstrate two applications of our work: we show that our framework leads to a larger variety of models than previous work, and we show fine-grained transformation sequences between two procedural models.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.5]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems

1. Introduction

Architectural design over centuries gives us many examples for the reuse, evolution, and the combination of designs [SMR04, Kni94]. We study these *design transformations* and present algorithms for their technical realization in the context of rule-based procedural modeling.

Rule-based procedural modeling systems, such as L-systems and shape grammars, provide an efficient method for the automatic creation of scenes with rich geometric detail. The rules specify how to iteratively evolve a design from a crude initial shape to a highly detailed model.

The simplest way to realize design transformations is to switch rules between different procedural models, e.g., to replace the window rule of one building with the window rule of another building. *Rule switching* can be used to combine two or more procedural models using the framework of Bayesian model merging [TYK*12].

We improve on this initial idea. Rule switching is the simplest method to obtain design transformations, but it only provides coarse-grained control and a limited number of new designs. In our framework we complement rule switching with *rule merging*, i.e., combining the effect of two rules. In contrast to rule switching, rule merging generally leads to an infinite number of variations. For our solution we need to overcome two major challenges. First, we need to be able to merge rules that are structurally different, e.g., they generate a different number of shapes and place them in different spatial arrangements. Second, grammars also exhibit structural differences globally. As a consequence, it is often not possible to establish simple one to one correspondences between shape (rule) labels. In such cases we can only establish sparse correspondences and it is not sufficient to merge individual rules. We propose a solution to this problem by merging shape trees that are the result of (partial) derivations using multiple rules. We call this new derivation framework *grammar co-derivation*.

To summarize, we present the following main contributions. On the technical side, we propose algorithms for rule merging for rule-based procedural modeling. On the application side, we can generate a larger space of design variations than competing methods (Fig. 2) and we demonstrate fine-grained transformation sequences between different procedural designs (Figs. 1 and 3). Transformation se-

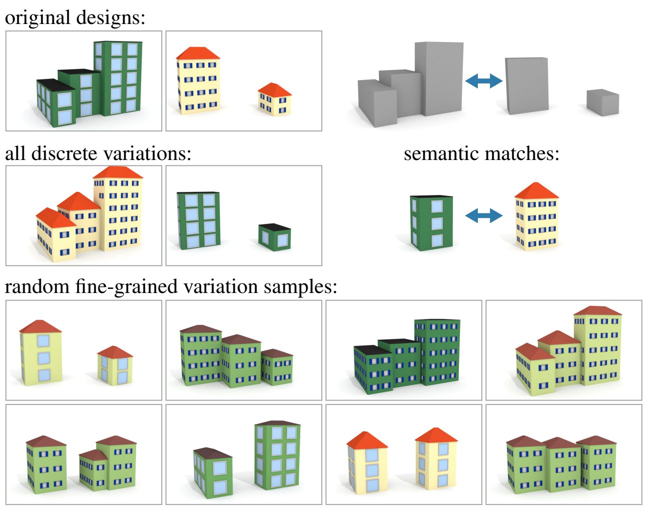

Figure 2: ***Coarse-Grained Versus Fine-Grained Transformations*** *Two simple grammars have correspondences (called semantic matches) for the production rules that generate the mass models and the building style (top right). Discrete rule switching spans only a very limited shape space with four designs, the two original designs (top left) plus two variations (middle left). Fine-grained rule merging can span a shape space with infinitely many designs (bottom).*

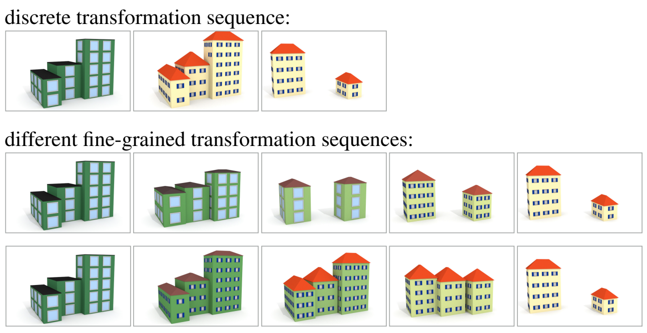
discrete transformation sequence:
different fine-grained transformation sequences:

Figure 3: ***Transformation Sequences*** *Discrete switches can only create very limited transformation sequences while fine-grained rule merging enables the generation of different interpolation paths that morph one design into another.*

quences are gradual transformations of one design into another, and they can be shown as computer animations. The generation of such transformation sequences is difficult to achieve using existing work.

2. Related Work

Grammar-based Procedural Modeling Shape grammars were invented by Stiny [Sti75]. In their original form, the derivation was too difficult to control and typically done with the user in the loop. Existing methods resort to very simple shapes or use graphs to represent shapes and subgraph matching to identify the rules, e.g., Grape [GE11]. A simplification of shape grammars are *set grammars* [Sti82] where rules act on a set of labeled shapes rather than on an arrangement of lines. Another milestone are L-systems which use a parallel replacement strategy. They have been successfully applied for plant modeling [Pru86, PL90] and extended to query and interact with their environment during derivation [PJM94, MP96]. Several shape grammar frameworks have been developed for modeling streets and buildings [PM01, WWSR03, MWH*06, KPK10].

An inspiration for this project was Terry Knight's book *Transformations in Design* [Kni94]. It describes how certain architectural styles evolved over time. The book goes further than just defining the styles with shape grammars; it introduces *meta rules* that do not operate on the shapes but rather modify the grammar's production rules. The evolution of a style can thus be represented by a sequence of meta rules that are applied to the shape grammar. Knight's work is based on the archetypal shape grammars by Stiny. While she discusses architectural concepts, our work focuses on the technical aspect of automatically computing design transformations. We demonstrate our idea of co-derivation both in the context of grammars similar to CGA shape [MWH*06] and L-systems [Pru86].

There are many other extensions for grammar-based procedural modeling, e.g., for interactive rule editing [LWW08], for guiding the derivation process of stochastic grammars [BŠMM11, TLL*11, RMGH15], for extending scopes to arbitrary convex polyhedra [TKZ*13], for a more expressive and context-sensitive grammar syntax combined with elements of functional programming [SM15], etc. There also exist evolutionary techniques for L-systems [McC93, Jac01, Bur13] that are similar to our method. They, however, apply changes to the grammar at random while we always transform between two or more designs.

Inverse Procedural Modeling & Grammar Induction In computer graphics there are several recent approaches that study how to generate design variations given a single existing design. Bokeloh et al. [BWS10], Stava et al. [ŠBM*10], and Talton et al. [TYK*12] all propose methods to compute simple shape grammars from input designs to generate variations of those input models. The main difference between our work and previous work is the granularity of the combination. While previous work mainly relies on rule switching, our grammar co-derivation and rule merging operations lead to a much larger shape space.

Algorithms that try to learn grammars tend to build on very simple context-free grammars, or they only learn rule parameters for predefined grammars. Most often this is done for building façades since their hierarchical split patterns render the induction process easier [MZWG07, STKP11, MMWG11, WRPG13, MG13].

Morphing & Style Transfer Our design transformations combine elements of volume morphing and style transfer — two successful modeling methods in computer graphics. Volume morphing was pioneered by Lerios et al. [LGL95] and Kanai et al. [KSK97]. An overview over different morphing methods is given by Alexa [Ale02]. Similar to our work are the methods by Alhashim et al. [ALX*14, AXZ*15] that do not only blend shapes geometrically but also topologically.

Style transfer is the process of applying the style of one exemplar model to another target model. Often geometric high frequency details are re-targeted and applied to another mesh [BIT04, MHS*14]. The same idea has also been applied to images [HJO*01].

Structure Preserving Editing There are structure-aware methods that automatically keep the finer geometric details intact while the coarse shape of a 3D model is changed [CLDD09, BWKS11, BWSK12]. Our method also has to deal with geometry changes at the structural and detail levels. We have the advantage to work with rule-based procedural models; if a design changes at a coarse level, the structure of the detail shapes can be kept consistent by reevaluating their production rules. Our challenge is to keep that consistency when combining elements from different grammars.

Design Exploration The space of all possible design transformations for a set of grammars could be interpreted as a parametric model that spans a shape space. That space could be navigated with an interactive exploration tool, e.g., the work by Talton et al. [TGY*09] or Dang et al. [DLC*15].

3. Overview

3.1. Grammar Definitions

As our proposed concepts are applicable to a broader range of rule-based procedural modeling systems, we describe our framework for a generalized set grammar, similar to CGA shape [MWH*06] and L-systems [Pru86]. We define a grammar G as a four-tuple:

$$G = \langle NT, T, \omega, P \rangle.$$

The grammar operates on shapes where each shape is assigned exactly one label, or *symbol*. The symbols stem from two disjoint sets: non-terminals (NT) and terminals (T). A shape can have a list of geometric and non-geometric attributes. The most important attributes are encoded by the *scope* [MWH*06], i.e., a local coordinate frame and the shape's size defining a (bounding) box in space. The grammar derivation starts with a single shape labeled with the special symbol $\omega \in NT$, the *axiom* or *starting symbol*. During the grammar derivation, a shape is selected and a *production rule* (or just *rule*), is applied to it. P is the set of rules of the form:

$$\textit{predecessor} : \textit{cond} \rightarrow \textit{successor},$$

where *predecessor* is a symbol $\in NT$ and *successor* is a general procedure (or program) that generates zero, one, or more successor shapes labeled with symbols ($\in NT \cup T$). A rule can only be applied if its condition *cond* is met. A condition can depend on shape attributes (e.g., the scope's size, position, etc.), the rule's parameters (in case of parametric grammars), and other shapes. In our examples, we use conditions for occlusion queries [MWH*06] in building grammars and as a recursion counter in plant grammars (the max number of derivation steps is initialized to n and decreased during derivation so that a rule can stop the recursion when $n < 0$). We use the term *deterministic* grammar if there is only a single possible derivation for all possible starting shapes. Otherwise, a grammar is called *stochastic*. Procedural modeling systems differ in the way a *successor* can be defined. Typically, commands like translation, scaling, rotation, instantiation of mesh and texture assets, and splitting rules are used.

The derivation generates a *shape tree*. After applying a rule to a non-terminal shape, all the shapes generated by *successor* will be added as children. We assume the derivation order to be breadth-first.

3.2. Framework Overview

Our framework consists of multiple components. In a preprocess we establish sparse correspondences between rules of the input grammars (Sec. 4.1). These correspondences help our proposed *co-derivation* (Sec. 4.2) to synchronize the simultaneous derivation of two grammars. The co-derivation algorithm uses two different strategies for rule merging: 1) a general shape matching and blending algorithm (Sec. 4.3) and 2) a specialized algorithm for split rules (Sec. 4.4). We further describe the extensions necessary for our two applications together with their results: co-derivation using multiple grammars is important for generating variations of designs (Sec. 5.1), and we introduce a user interface to control transformation sequences (Sec. 5.2).

4. Co-Derivation of Shape Grammars

4.1. Sparse Correspondences

We require a sparse set of correspondences between the symbols (rules) of the input grammars. These correspondences are called *semantic matches*. In general, this is a modeling problem that does not have a single correct solution, and we let the users adjust the semantic matches according to their design intent. We use the tuple notation $\langle Sym_a, Sym_b \rangle$ to say that symbol Sym_a in one grammar matches symbol Sym_b in another.

Since the structure of the grammars can be quite different, not every rule will have a valid match. For the presented examples, only 24-54% of the rules have matches. We allow one-to-many matches, i.e., it is possible that a rule has more than one match in another grammar. For example, one grammar might have different rules for *GothicWindow* and *RoundWindow* which both match the only *Window* rule in the other grammar. The axioms of all grammars are always set to be a semantic match.

Finding semantic matches automatically is a very challenging problem and beyond the scope of this paper. We have the user annotate selected rules with tags coming from a set of architectural vocabulary (e.g., mass model, roof, façade, floor, window tile, window, etc.). Rules with the same tag are automatically matched while non-tagged rules are not matched. This assignment works in most cases. Otherwise the user can manually refine the semantic matches.

4.2. Grammar Co-Derivation

Grammar co-derivation is an extension of traditional grammar derivation to two grammars. We first look at the problem of ex-

Algorithm 1 coDerivationStep_1(*shape*, *grammar*)

```
1:  derive(shape, grammar)
2:  repeat
3:      $done = true$
4:      for each $l \in$ leaves(shape_tree) do
5:          $sym$ = symbol($l$)
6:
7:          # keep deriving non-terminal leaves without semantic matches
8:          if $\nexists$ semantic match for $sym$ && $sym \in NT$ then
9:              derive($l$, grammar)
10:             $done = false$
11: until done
```

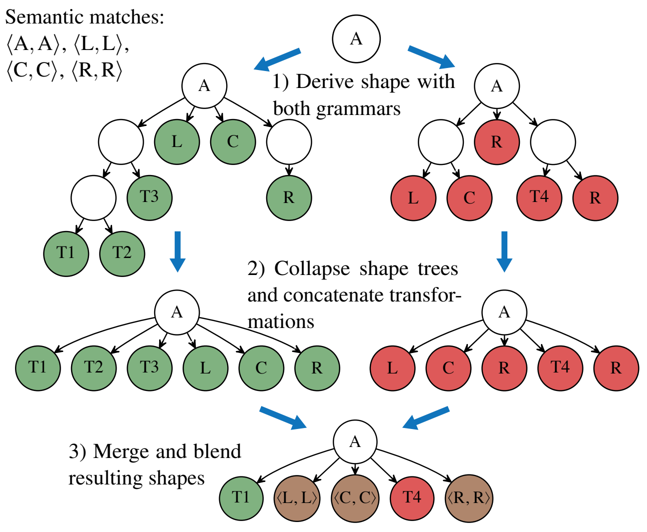
Semantic matches:
⟨A,A⟩, ⟨L,L⟩,
⟨C,C⟩, ⟨R,R⟩
1) Derive shape with both grammars
2) Collapse shape trees and concatenate transformations
3) Merge and blend resulting shapes
A
L
C
R
T1
T2
T3
T4
⟨L, L⟩
⟨C, C⟩
⟨R, R⟩

Figure 4: ***The Co-Derivation Step*** *1) A shape, whose symbol A has a semantic match, is separately derived with both grammars. 2) Both subtrees are collapsed and all leaf shapes are transformed into a common coordinate system. 3) The resulting shapes are combined in a merge step according to user preferences. Shapes can come from the first (green) or the second (red) grammar. If a shape's symbol has a semantic match, it can also be blended together with matching shapes from the other subtree (brown). Blended shapes are labeled with the match tuple.*

tending a single derivation step, and then use our proposed solution to design a complete grammar co-derivation algorithm.

The elementary step of a traditional shape grammar is to select a shape, select a rule, and replace the shape by its successor by adding newly generated shapes as children in the shape tree. We call the elementary step of grammar co-derivation, used to derive shapes with semantic matches, a *co-derivation step*. The co-derivation step generates separate partial derivations of a shape x for each grammar and merges both outputs. The fundamental challenge of designing a co-derivation step is the following: if we only apply a single rule from each of the two input grammars to x, we likely end up with sets of incompatible shapes. Some of these will have semantic matches and others will not. However, good merging results can only be achieved if we work with two sets of shapes where all of them either have semantic matches or are terminal shapes. A co-derivation step therefore needs to synchronize the derivation. Each grammar needs to derive shape x until only terminal shapes or shapes with semantic matches exist. Our co-derivation step has three parts (also see Fig. 4).

1. Derive x with both grammars. Similar to traditional grammar derivation, we also use breadth-first order, but stop at shapes that have a semantic match or that are terminals. See Alg. 1.
2. Collapse the two subtrees. For each of the two subtrees, we bring all shapes into a common coordinate system and delete intermediate nodes.
3. Merge all shapes of both collapsed subtrees into a final shape set. The merging can either combine shapes from the two derivations, discard shapes, or copy shapes unmodified. The resulting shapes are appended to the shape tree as x's children. The merging operation is more involved and we will explain two merging algorithms in Secs. 4.3 and 4.4.

To define a complete co-derivation, we recursively apply co-derivation steps until all the shape tree leaves become terminal shapes. See Alg. 2 for pseudo code.

4.3. Rule Merging

The input to this step are two sets of shapes, one from grammar G_a and one from grammar G_b. The output of this step is a single set of shapes that are either directly copied from G_a or G_b and/or shapes that are combined by blending pairs of shapes. In general, this is an underspecified problem with many possible solutions. In order to manage this large design space, we propose the following method to parameterize the solutions. For each semantic match, we use a parameter $p_s \in [0,1]$. $p_s = 0$ means that all shapes are directly copied from grammar A while $p_s = 1$ means all shapes are directly copied from grammar B. In addition, we use a single threshold th (default 0.5) to facilitate certain discrete decisions, e.g., when to switch assets.

To compute the resulting set of shapes, we need three ingredients: 1) a distance metric that can be computed between each pair of shapes, 2) a matching algorithm that computes matches based on the distance metric, and 3) an interpolation algorithm that combines matched (and unmatched) shapes.

We propose a distance metric d that is the sum of two components, a semantic distance d_s and a geometric distance d_g. The semantic distance d_s is zero if there is a semantic match between the labels of the two shapes and infinity otherwise. That means that two shapes can only be matched if there also exists a semantic match between their corresponding symbols. The distance-metric d_g compares shape positions and scope sizes by summing up the point-to-point distances of the eight scope corners.

We propose two automatic ways to compute matches based on the metric d. (We allow the user to override the automatic assignment for additional artistic control, but this option is not used in any

Algorithm 2 Co-Derivation Process

```
1:  $shape\_tree$ = new node($\langle Axiom, Axiom \rangle$)
2:
3:  repeat
4:      for each $l \in$ leaves($shape\_tree$) do
5:          if symbol($l$) $\notin T$ then
6:              # derive twice and collapse shape trees to height 1
7:              $st_1$ = copy($l$)
8:              coDerivationStep($st_1$, $grammar_1$)
9:              collapse($st_1$)
10:             $st_2$ = copy($l$)
11:             coDerivationStep($st_2$, $grammar_2$)
12:             collapse($st_2$)
13:
14:             # calculate resulting set of shapes
15:             $res$ = mergeAndBlend(leaves($st_1$), leaves($st_2$))
16:             $l$.append($res$)
17: until $shape\_tree$ does not grow anymore
```

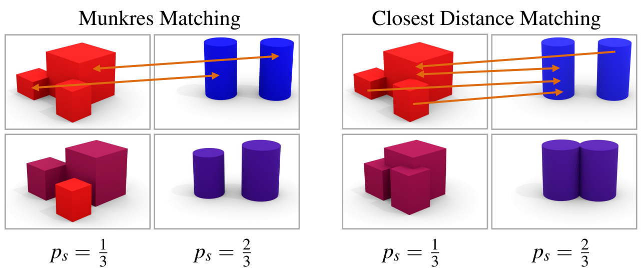

Figure 5: ***Shape Matching & Blending*** *The two automatic strategies for shape matching are Munkres assignment (top left) or closest distance matching (top right). Munkres matches are always bidirectional. (bottom) The merged and blended resulting shape sets are shown for the parameter values of* $\frac{1}{3}$ *and* $\frac{2}{3}$ *with a threshold of* 0.5.

of our results.) 1) The first strategy uses the Munkres (or Hungarian) algorithm [Mun57] for finding a minimum weight maximum matching in a weighted bipartite graph. It results in only one-to-one and one-to-none matches. 2) The second matching algorithm assigns each shape from G_a's output the closest shape derived by G_b and vice versa. Note that this assignment, unlike the result of the Munkres algorithm, is not necessarily symmetric. All pairs of shapes that mutually map to each other are included in the set of pairs. The Munkres matching method might leave remaining single (unmatched) shapes while the closest distance approach will likely end up with some unidirectional matching pairs.

Our proposed interpolation algorithm works as follows. Singles from G_a and unidirectional pairs pointing from G_a to G_b are kept if the corresponding parameter p_s is below the threshold th and vice versa. The advantage of this approach is that it also allows one-to-many matches. A shape matching and blending example is given for both proposed strategies in Fig. 5. The shapes of matching pairs are blended by interpolating their scope attributes according to the corresponding parameter p_s. Position, size, and color are linearly interpolated, while we use quaternion slerp to interpolate the shapes' orientations with respect to their centers. For overlapping blended shapes, an additional parameter decides if they are kept as is or if they are merged together (which enables interesting scenarios such as the one with multiple houses merging into one building shown in Fig. 12). Currently, we do not support the morphing of asset geometry attached to the shapes. We simply use the threshold to switch between the assets from G_a and G_b. The same applies to all textures. In the output, all blended shapes are labeled with tuple names.

4.4. Rule Merging for Split Rules

Split rules [WWSR03, MWH*06] are a particular type of rule to define the *successor* of a shape. They are most commonly used for architectural modeling. A split rule subdivides a shape along an axis into a set of smaller shapes that are tightly aligned. For example, a split rule along the y-axis can partition a façade into individual floors. An advantage of split rules is that they can be written to adapt to the size of shapes (encoded by the scope). For example, a split rule can generate a reasonable number of floors for façades of any height. An important aspect of split rules is that they partition the scope, i.e., none of the successor shapes should overlap, and all the space defined by the scope should be filled by successor shapes. The previously described solution cannot guarantee these two conditions, because the shapes produced by a split rule cannot be transformed independent of each other. For example, a shape in the split pattern can only become larger if other shapes shrink. Our method for handling split rules guarantees that the output is also a valid split pattern. We again use a parameter for semantic matches to parametrize the solution space.

Our proposed algorithm has two steps, structure computation and geometry computation. The first step computes a sequence of symbols (thereby deciding on the number, type, and relative position of symbols). The second step computes the shape attributes, such as size.

Structure Computation For example, a string for a split pattern of a building floor consisting of windows (W), pillars (P), and wall pieces (ω) could be:

$$\{\omega, W, \omega, P, \omega, W, \omega, P, \omega, W, \omega, P, \omega, W, \omega, P, \omega, W, \omega\}.$$

In building designs the split patterns consist of relevant shapes such as ornaments or windows. For our split transformation algorithm that is tailored towards such architectural patterns, the wall pieces that make up the spacing between the important elements are irrelevant. The user indicates the symbol that is used for spacing (default *Wall*). The example pattern is reduced to:

$$\{W, P, W, P, W, P, W, P, W\}.$$

The removed wall shapes will be reinserted again in the geometry computation step.

We propose an algorithm based on an edit distance between two strings. We optimize the edit distance and take the lowest cost edit sequence to compute intermediate strings. The algorithm takes two reduced strings as input (*src* and *dst*) and produces a sequence of intermediate strings. A single parameter value p_r will define what string of the sequence is used at the current state.

The edit distance is computed by summing the cost of elementary edit operations:

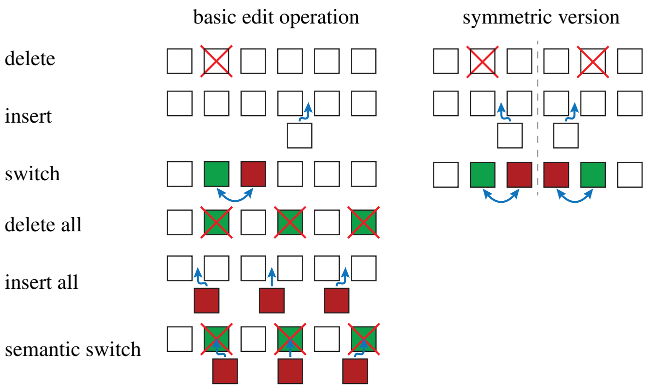

Figure 6: ***Edit Operations*** *Overview of all edit operations for split transformations. For details see Sec. 4.4.*

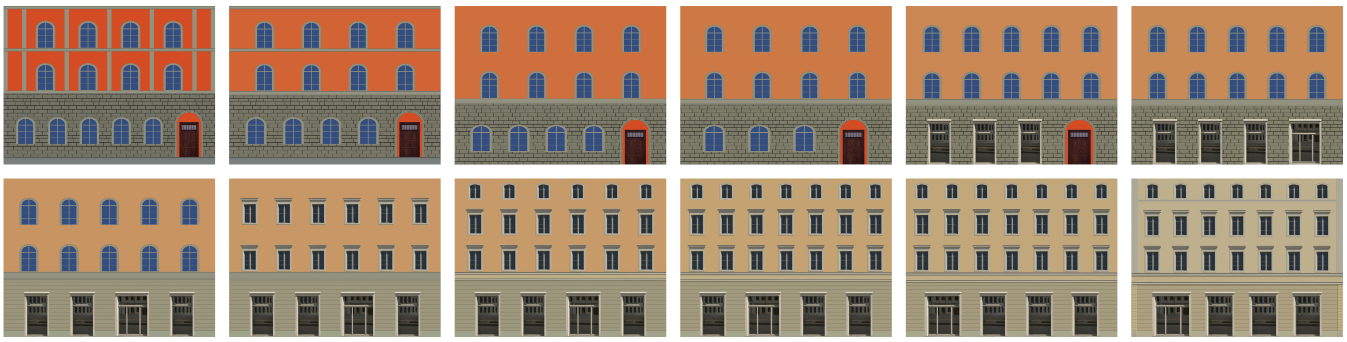

Figure 7: ***Façade Split*** *A transformation between two hierarchical façade split patterns. The first (top left) and last (bottom right) images show the two inputs to the algorithm, and the remaining eight images show intermediate steps. The entire transformation applies 18 gradual changes to façade structure over the timeline. For elements with semantic matches, scope sizes and shape colors are smoothly interpolated.*

- *delete* – Removes a single symbol from the string. Only applicable if the symbol (or a semantic match) appears in *src* and *dst* but more often in *src*.
- *insert* – Inserts a single symbol. Can be used if the symbol (or a semantic match) appears in *src* and *dst*, but more often in *dst*.
- *switch* – Exchanges the positions of two different, neighboring symbols.
- *delete all* – Removes all symbols with a given label that only appears in *src*.
- *insert all* – Inserts all symbols with a given label that only appears in *dst*.
- *semantic switch* – Replaces all symbols of a label with semantically matching symbols.

Since split rules are predominantly used for regular patterns with repeating elements or with a clear structure, we also want edit operations that keep such regularities intact. For strings that are symmetric, i.e., they read the same from the left and the right (like a palindrome), we define symmetric versions of the delete, insert, and switch operations. These symmetrically apply the same edit operation twice. Further, if a string is symmetric, edits that result in a symmetric string again will be preferred over edits that break the symmetry. All edit operations are summarized in Fig. 6. Two simple examples of what is possible with such edits are shown in Fig. 8.

Each possible edit operation has an assigned cost that reflects how drastic the change of the string is. The task of finding a smooth sequence of strings that gradually changes from *src* to *dst* can be

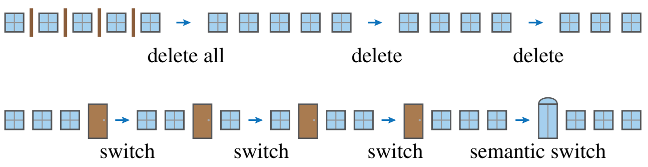

Figure 8: ***Edit Sequences*** *Two simple sequences that apply edit operations. (top) First all pillars are removed, then the number of windows is gradually reduced to three. (bottom) A door is moved from the left to the right with subsequent switches, then a semantic switch changes the door style.*

cast as a shortest path problem on a graph that can be solved with Dijkstra's algorithm. Graph nodes represent strings while each edit operation and its cost correspond to a weighted edge. We never store the entire graph in memory since we do not know it upfront. We expand it on the fly; when the search arrives at a specific string, we grow the graph by adding all remaining neighbors of the current node. All edit operations except the symmetric ones have a cost of 1. If an edit breaks the string's symmetry its cost is increased by 100 to assure that a non-symmetry breaking sequence is given priority (if one exists). Symmetric edit operations cost 2.5. They are less gradual than two normal edit operations because they change two symbols at once. However, it is often not possible to apply two subsequent normal edit operations without breaking the symmetry. To prevent the search from exploding, we also penalize edits that result in strings that are longer or shorter than the two input strings. This speeds up the search without changing its outcome.

Geometry Computation This step of the algorithm sets the shape sizes and adds spacing between the elements by reinserting walls. Our observation to make this algorithm work is that we not only need to consider the sizes of shapes in the two input sets but also the average sizes of shapes occurring in the model in general. We fully derive all input grammars and compute the average sizes of all shapes with a specific symbol and the average distances between neighboring shapes. If a symbol in the output has a semantic match, the average shape sizes from grammars G_a and G_b are interpolated according to the parameter p_r. The average distances are used to reinsert *Wall* shapes of that size. If two symbols do not appear as neighbors in the input, the average spacing between one symbol and any other symbol is used instead. The sum of the sizes of all resulting shapes might not exactly match the size of the parent scope. Therefore we uniformly rescale the *Wall* shapes to match the size. In Fig. 7 we show an example result of our method for transforming split rules applied to two hierarchical patterns consisting of several nested splits.

5. Applications & Results

We describe how we use and adapt the basic design transformation framework to enable our two modeling applications, and we present their results.

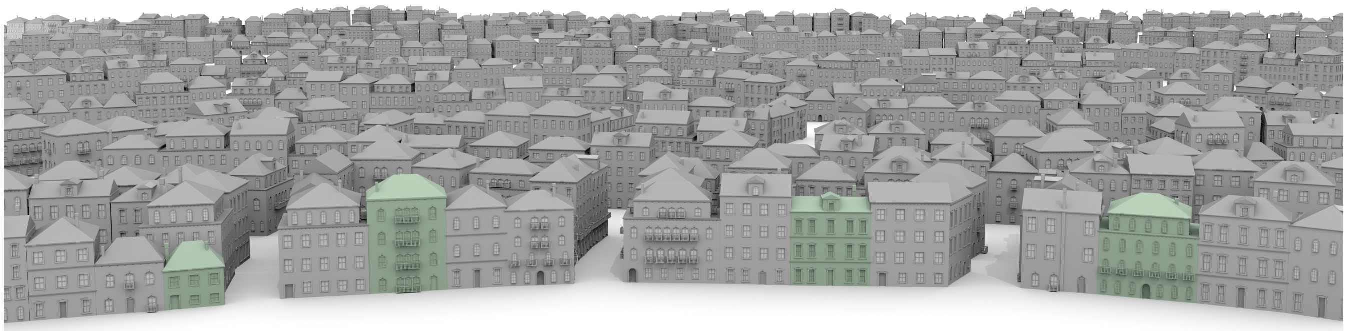

Figure 9: ***Venice Variations*** *Design transformations enable the generation of new variations from a small set of exemplars. The four original Venice buildings are highlighted in green. Random design transformations combine and blend elements from the different exemplars to generate arbitrary new building variations to populate the rest of a virtual Venice.*

5.1. Variety Generation with Multiple Grammars

To compute a co-derivation of multiple input grammars, we extend our framework. When working with n grammars, a semantic match tuple can have a maximum size n. To derive shapes with tuple labels, we randomly pick two of the participating grammars that define a rule for that symbol and apply the co-derivation step described in Sec. 4.2. To generate a random variety of shapes, we can randomly sample parameters for the semantic matches and their thresholds during the derivation.

We explore the concept of generating a variety of designs in the context of urban modeling. The idea behind our approach is that it is much easier to write deterministic grammars depicting a few designs (for example by recreating designs from photographs) rather than writing a stochastic grammar describing a larger shape space. Design transformations can then be used to generate a city consisting of new unique models of the same architectural style. The input to our example are four deterministic exemplar grammars depicting Venice style buildings. The output is a city consisting of about 400 buildings (Fig. 9). The buildings are the result of randomly combining and transforming parts of the different Venice grammars.

To compute the design transformations we use the shape blending algorithm from Sec. 4.3 for the block and roof rules and the split rule merging algorithm from Sec. 4.4 for the façade rules.

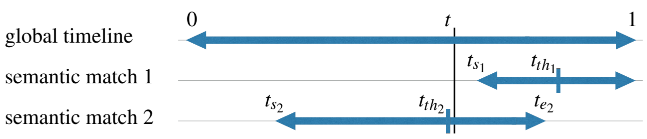

Figure 10: ***User Interface*** *A sketch of the user interface for making transformation sequences. The transformation of the first semantic match is gradually executed over the time range $[t_{s_1}, 1]$ and has not started at t. The transformation for the second semantic match is active at t since $t \in [t_{s_2}, t_{e_2}]$. t_{th_1} and t_{th_2} represent the thresholds that control discrete decisions. For semantic match 2, asset geometry and textures are taken from the second grammar since $t > t_{th_2}$.*

5.2. Transformation Sequences

One application of our work is to compute a transformation sequence between the design of one grammar and the design of another. There are two conflicting goals for designing transformation sequences. First, all intermediate designs should be valid. Second, the transitions between intermediate designs should be as smooth as possible. For example, a valid building design needs to have windows of a certain minimum size, which requires discontinuities in the transformation sequence. However, these should be limited as much as possible to favor smoothness. Similar animations are often used as special effects in the entertainment and movie industry. We control transformation sequences by a global timeline parameter $t \in [0, 1]$. All individual transformation parameters for each semantic match are derived from the global timeline. The user has the possibility to change the default mapping by adjusting start and end times and thresholds. By default all parameters span the full range and all thresholds are set to 0.5. Fig. 10 illustrates the user interface.

We show transformation sequences for building and plant models. All sequences can have alternative interpolations paths. See Fig. 3 and the Sternwarte example described in the following.

Farm $\rightarrow$ Castle In Fig. 11 a farm transforms into a castle. Both consist of several mass models that are matched as follows: *Shed* and *MainBuilding* symbols match *MainHall*, *CastleWall*, and *Gate* symbols, and *Silo* corresponds to *Tower*.

Residential Houses $\rightarrow$ High-Rise Fig. 12 shows the transition of a group of several small residential houses into fewer but larger apartment blocks. From there the scene transforms into a high-rise building. Merging the buildings together in such a fashion is facilitated by one-to-many matches.

Chain of Transformations We use four procedural buildings to create transformations between them: the Sternwarte building designed by Semper, two modern residential buildings, and a Hausmannian building from Paris. Transforming the buildings in the given order leads to three transformation sequences that can be concatenated together as shown in Fig. 13. An alternative transformation of the first part, in which all façade elements change before the mass models, is shown in Fig. 14.

Tree L-Systems The first plant transformation sequence (Fig. 1) is based on two procedural models presented by Honda [Hon71] and by Aono and Kunii [AK84]. The example illustrates how a monopodial branching pattern smoothly changes into a sympodial one. In both cases, an apex will always create exactly two offspring branches.

Flower L-Systems A more complex plant example (Fig. 15) is inspired by the L-systems in Figs. 1.26 and 3.14 in Prusinkiewicz and Lindenmayer's book [PL90]. Not only does this example feature leaves and blossoms whose colors, shapes, and sizes are interpolated, but the branching structure is also more complicated. In each iteration, the first grammar grows each apex into two subsequent internodes and three new apexes, one between the internodes and two at the end of the outer internode. The other grammar grows an apex into an internode followed by one younger and one older apex. At the next iteration the older apex recursively repeats the branching, while the younger apex grows into an older one (which will only expand later). The rules have to be carefully matched so that the merge and blend steps happen after every iteration for the first L-system but only after every other iteration for the second L-system.

5.3. Quantitative Results

Our implementation of design transformations builds on a newly designed procedural modeling system that combines elements of CGA Shape, G^2 [KPK10], and L-systems. We implemented the framework in C++ and used a 2012 MacBookPro to measure the running times. See Tab. 1 for the timings of our algorithm and other quantitative information of the examples in the paper.

Name	Figs.	Time [s]	#Terminals ×10^3	#Rules	#Matches	%NT w. Match	#Params	#Edits
Tree L-Systems	1	0.11	0.5	23	6	39	10	0
Fine-grained Variations	2, 3	0.05	0.6	34	4	24	10	0/6
Façade Split	7	0.02	0.2	51	15	54	30	0
Venice Variations	9	0.06	0.8	108	15	46	25	0
Farm → Castle	11	0.18	3.3	117	20	24	50	22
Houses → High-rise 1	12	0.20	1.8	48	9	38	14	3
Houses → High-rise 2	12	0.23	2.9	49	9	37	14	4
Sternwarte Chain 1	13, 14	0.15	1.0	107	21	36	39	17/23
Sternwarte Chain 2	13	0.06	0.7	60	12	40	22	4
Sternwarte Chain 3	13	0.10	1.5	85	13	27	23	7
Flower L-Systems	15	0.04	0.3	42	8	39	18	4

Table 1: ***Results*** *We list the following results (and their corresponding figures): the average computation time per transformed model (mean over 200 frames or variations), the average number of terminals shapes in the design, the number of rules of all involved grammars, the number of semantic matches, the average percentage of rules with matches (%NT w. Match), the number of parameters, and the number of parameters adjusted by the user (#Edits). If a result has two different interpolation paths, we show two different numbers of user edits. The percentage of rules with matches is an indicator of the sparsity of the correspondences. It cannot simply be inferred from the number of rules and semantic matches since certain rules are part of more than one match. The number of parameters hints at the large size of the shape spaces. Comparatively few user edits are usually sufficient to get a reasonable result.*

6. Discussion

Our two applications clearly show how design transformations facilitate the creative process of combining and merging parts of different grammars. It is important to note that our framework does not need stochastic grammars as input but works directly with deterministic grammars. While we can also apply our method to stochastic grammars, modeling deterministic grammars is much easier for the user. Therefore, our framework can help a user to accelerate modeling larger environments because the time-consuming modeling step of extending deterministic grammars to stochastic ones can be omitted.

Difference to L-Systems Our design transformations are not directly applied to L-systems. Instead we recreate the procedural plant descriptions using our grammar. The main reason is because traditional L-systems derive a complete model as a string in a first pass and then interpret the string as a geometric model in a second pass. In contrast, our grammar interpreter provides a geometric interpretation at every intermediate step, as required by our transformation framework. Plant models also require two minor changes in the derivation. First, rotations are not interpolated around shape centers, but are interpolated around the attachment points at the scope origins. Second, the output of a merging operation needs to be a valid branching structure. This can be achieved by enforcing connectivity as a constraint in the merging step.

Limitations & Future Work Our work has several limitations and possibilities for improvements. First, it would be better if semantic matches could be found fully automatically without any manual annotations. This is a very hard problem since one single mismatch can impair the result. Some initial results for a related problem were presented by Alhashim et al. [ALX*14, AXZ*15]. However, this problem becomes significantly more difficult for our inputs since semantic matches might exist between parts of the design that are geometrically and structurally different. Second, we aim for transformation sequences that are as smooth as possible, but there are still noticeable discrete changes when transforming one model into another. The reason is that many objects, such as windows, cannot realistically grow from zero size, but they need to start appearing with a realistic minimum size. This leads to some discontinuities in the transformation.

7. Conclusions

We present design transformations as a modeling tool for rule-based procedural modeling. On the application side, we show two improvements of the state of the art. First, we show how design transformations can create a larger set of variations from a set of input designs than previous work. Second, we show how to compute fine-grained transformation sequences between two input designs that cannot be generated with previous work. We describe two main components that enable design transformations: *grammar co-derivation* and *rule merging*. The results show example design transformations for a variety of building and plant models.

Acknowledgements We thank Gaspard Zoss for rendering all the figures and the anonymous reviewers for their valuable comments.

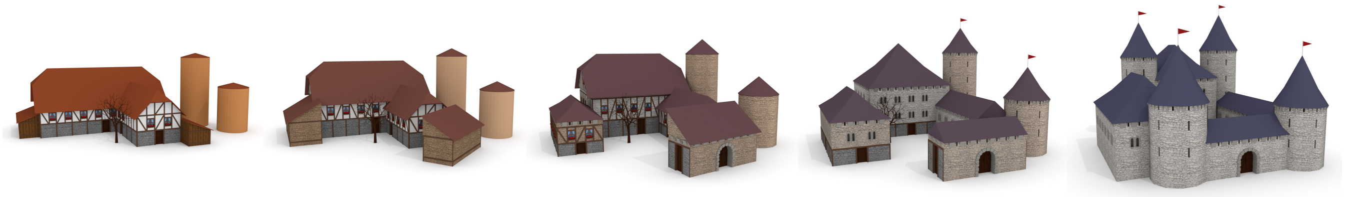

Figure 11: ***Farm*** $\rightarrow$ ***Castle*** *A farm gradually transforms into a castle. The farm consists of six mass model components (house parts, sheds, and silos) and the castle of eight (towers, walls, and main halls).*

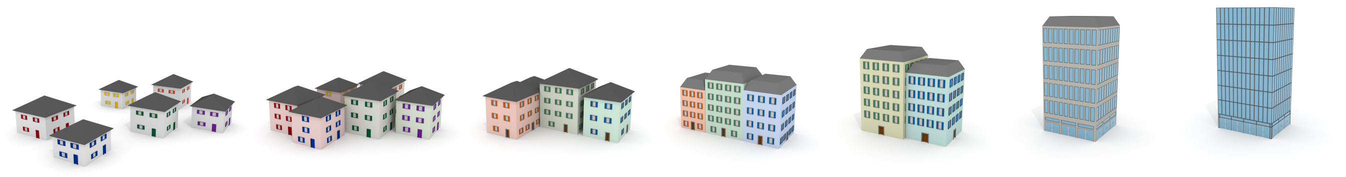

Figure 12: ***Houses*** $\rightarrow$ ***High-rise*** *A transformation from several small residential houses into apartment blocks that eventually converge into a high-rise office building. For this scene we allow one-to-many shape matches, and we decide to collapse intersecting mass model shapes.*

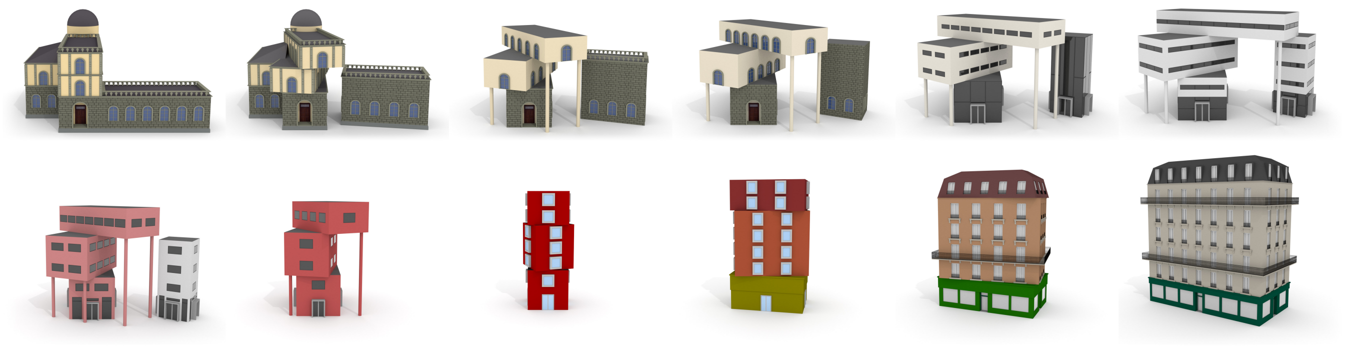

Figure 13: ***Sternwarte Chain*** *An animation sequence that consists of several sequential transformations. Semper's Sternwarte design is first changed into a white modern building, then into a red modern building, and finally into a Hausmannian building.*

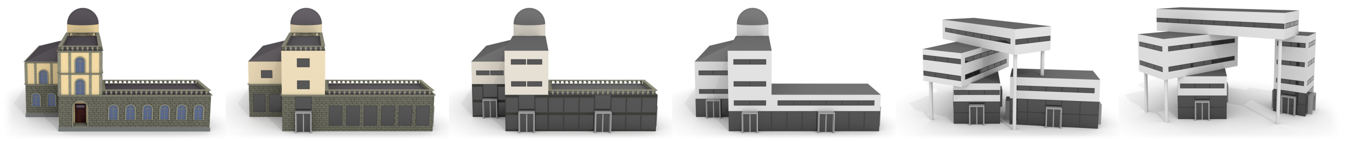

Figure 14: ***Sternwarte Chain 1 Alternative*** *An alternative interpolation path for the same transformation as shown in the first row of Fig. 13. This time all façade rules are transformed before any of the mass model rules.*

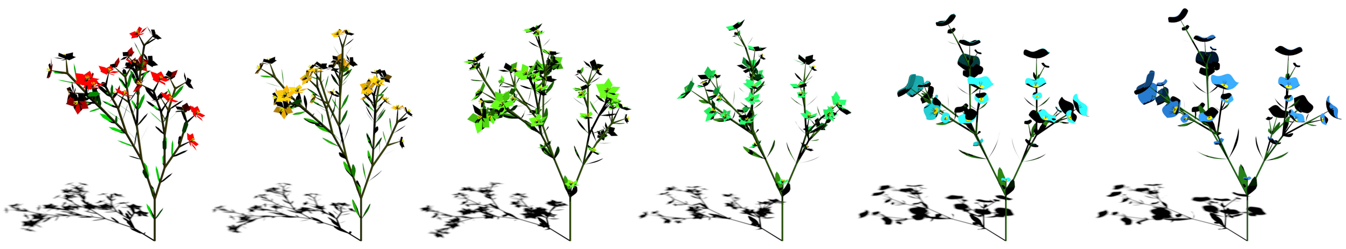

Figure 15: ***Flower L-Systems*** *A more complicated transformation sequence that gradually changes the output of one flower L-system into another. Design transformations do not only blend the overall tree structure but also the smaller leaf and blossom shapes. What further complicates the transformation is that the growing rules for both flowers are structurally very different.*

References

[AK84] Aono M., Kunii T. L.: Botanical Tree Image Generation. *IEEE Computer Graphics and Applications* (1984). 8

[Ale02] Alexa M.: Recent Advances in Mesh Morphing. *Comp. Graph. Forum* (2002). 2

[ALX*14] Alhashim I., Li H., Xu K., Cao J., Ma R., Zhang H.: Topology-Varying 3D Shape Creation via Structural Blending. *ACM Trans. Graph.* (2014). 2, 8

[AXZ*15] Alhashim I., Xu K., Zhuang Y., Cao J., Simari P., Zhang H.: Deformation-driven Topology-varying 3D Shape Correspondence. *ACM Trans. Graph.* (2015). 2, 8

[BIT04] Bhat P., Ingram S., Turk G.: Geometric Texture Synthesis by Example. *Comp. Graphics Forum* (2004). 3

[BŠMM11] Beneš B., Šť'ava O., Měch R., Miller G.: Guided Procedural Modeling. *Comp. Graph. Forum* (2011). 2

[Bur13] Burt T.: *Interactive Evolution by Duplication and Diversification of L-systems*. Master's thesis, University of Calgary, 2013. 2

[BWKS11] Bokeloh M., Wand M., Koltun V., Seidel H.-P.: Pattern-aware Shape Deformation Using Sliding Dockers. *ACM Trans. Graph.* (2011). 3

[BWS10] Bokeloh M., Wand M., Seidel H.-P.: A Connection Between Partial Symmetry and Inverse Procedural Modeling. *ACM Trans. Graph.* (2010). 2

[BWSK12] Bokeloh M., Wand M., Seidel H.-P., Koltun V.: An Algebraic Model for Parameterized Shape Editing. *ACM Trans. Graph.* (2012). 3

[CLDD09] Cabral M., Lefebvre S., Dachsbacher C., Drettakis G.: Structure Preserving Reshape for Textured Architectural Scenes. *Comp. Graph. Forum* (2009). 3

[DLC*15] Dang M., Lienhard S., Ceylan D., Neubert B., Wonka P., Pauly M.: Interactive Design of Probability Density Functions for Shape Grammars. *ACM Trans. Graph.* (2015). 3

[GE11] Grasl T., Economou A.: GRAPE: A Parametric Shape Grammar Implementation. *Symposium on Simulation for Architecture and Urban Design* (2011). 2

[HJO*01] Hertzmann A., Jacobs C. E., Oliver N., Curless B., Salesin D. H.: Image Analogies. *Proc. of SIGGRAPH* (2001). 3

[Hon71] Honda H.: 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* (1971). 8

[Jac01] Jacob C.: *Illustrating Evolutionary Programming with Mathematica*. Morgan Kauffmann, 2001. 2

[Kni94] Knight T. W.: *Transformations in Design: A Formal Approach to Stylistic Change and Innovation in the Visual Arts*. Cambridge University Press, 1994. 1, 2

[KPK10] Krecklau L., Pavic D., Kobbelt L.: Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Comp. Graph. Forum* (2010). 2, 8

[KSK97] Kanai T., Suzuki H., Kimura F.: 3D Geometric Metamorphosis based on Harmonic Map. *Pacific Graphics. Computer Graphics and Applications* (1997). 2

[LGL95] Lerios A., Garfinkle C. D., Levoy M.: Feature-Based Volume Metamorphosis. *Proc. of SIGGRAPH* (1995). 2

[LWW08] Lipp M., Wonka P., Wimmer M.: Interactive Visual Editing of Grammars for Procedural Architecture. *ACM Trans. Graph.* (2008). 2

[McC93] McCormack J.: Interactive Evolution of L-system Grammars for Computer Graphics Modelling. *Complex Systems: From Biology to Computation* (1993). 2

[MG13] Martinovic A., Gool L. V.: Bayesian Grammar Learning for Inverse Procedural Modeling. *IEEE CVPR* (2013). 2

[MHS*14] Ma C., Huang H., Sheffer A., Kalogerakis E., Wang R.: Analogy-Driven 3D Style Transfer. *Comp. Graph. Forum* (2014). 3

[MMWG11] Mathias M., Martinovic A., Weissenberg J., Gool L. V.: Procedural 3D Building Reconstruction using Shape Grammars and Detectors. *Int. Conf. on 3D Imaging, Modeling, Processing, Visualization and Transmission* (2011). 2

[MP96] Měch R., Prusinkiewicz P.: Visual Models of Plants Interacting with Their Environment. *Proc. of SIGGRAPH* (1996). 2

[Mun57] Munkres J.: Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics* (1957). 5

[MWH*06] Müller P., Wonka P., Haegler S., Ulmer A., Gool L. V.: Procedural Modeling of Buildings. *ACM Trans. Graph.* (2006). 2, 3, 5

[MZWG07] Müller P., Zeng G., Wonka P., Gool L. V.: Image-based Procedural Modeling of Facades. *ACM Trans. Graph.* (2007). 2

[PJM94] Prusinkiewicz P., James M., Měch R.: Synthetic Topiary. *Proc. of SIGGRAPH* (1994). 2

[PL90] Prusinkiewicz P., Lindenmayer A.: *The Algorithmic Beauty of Plants*. Springer Verlag, 1990. 2, 8

[PM01] Parish Y. I. H., Müller P.: Procedural Modeling of Cities. *Proc. of SIGGRAPH* (2001). 2

[Pru86] Prusinkiewicz P.: Graphical Applications of L-systems. *Proc. on Graphics Interface/Vision Interface* (1986). 2, 3

[RMGH15] Ritchie D., Mildenhall B., Goodman N. D., Hanrahan P.: Controlling Procedural Modeling Programs with Stochastically-Ordered Sequential Monte Carlo. *ACM Trans. Graph.* (2015). 2

[ŠBM*10] Šť'ava O., Beneš B., Měch R., Aliaga D. G., Krištof P.: Inverse Procedural Modeling by Automatic Generation of L-systems. *Comp. Graph. Forum* (2010). 2

[SM15] Schwarz M., Müller P.: Advanced Procedural Modeling of Architecture. *ACM Trans. Graph.* (2015). 2

[SMR04] Semper G., Mallgrave H., Robinson M.: *Style: Style in the Technical and Tectonic Arts; or, Practical Aesthetics*. Getty Research Institute, 2004. 1

[Sti75] Stiny G. N.: *Pictorial and Formal Aspects of Shape and Shape Grammars and Aesthetic Systems*. PhD thesis, University of California, Los Angeles, 1975. 2

[Sti82] Stiny G. N.: Spatial Relations and Grammars. *Environment and Planning B* (1982). 2

[STKP11] Simon L., Teboul O., Koutsourakis P., Paragios N.: Random Exploration of the Procedural Space for Single-View 3D Modeling of Buildings. *IJCV* (2011). 2

[TGY*09] Talton J. O., Gibson D., Yang L., Hanrahan P., Koltun V.: Exploratory Modeling with Collaborative Design Spaces. *ACM Trans. Graph.* (2009). 3

[TKZ*13] Thaller W., Krispel U., Zmugg R., Havemann S., Fellner D. W.: Shape Grammars on Convex Polyhedra. *Computers & Graphics* (2013). 2

[TLL*11] Talton J. O., Lou Y., Lesser S., Duke J., Měch R., Koltun V.: Metropolis Procedural Modeling. *ACM Trans. Graph.* (2011). 2

[TYK*12] Talton J., Yang L., Kumar R., Lim M., Goodman N., Měch R.: Learning Design Patterns with Bayesian Grammar Induction. *ACM Symp. User Interface Software and Technology* (2012). 1, 2

[WRPG13] Weissenberg J., Riemenschneider H., Prasad M., Gool L. V.: Is there a Procedural Logic to Architecture? *IEEE CVPR* (2013). 2

[WWSR03] Wonka P., Wimmer M., Sillion F., Ribarsky W.: Instant Architecture. *ACM Trans. Graph.* (2003). 2, 5