Inverse Procedural Modeling of Branching Structures by Inferring L-Systems

JIANWEI GUO, NLPR, Institute of Automation, CAS
HAIYONG JIANG, UCAS and NTU Singapore
BEDRICH BENES, Purdue University
OLIVER DEUSSEN, SIAT Shenzhen and University Konstanz
XIAOPENG ZHANG, NLPR, Institute of Automation, CAS
DANI LISCHINSKI, The Hebrew University of Jerusalem
HUI HUANG*, Shenzhen University

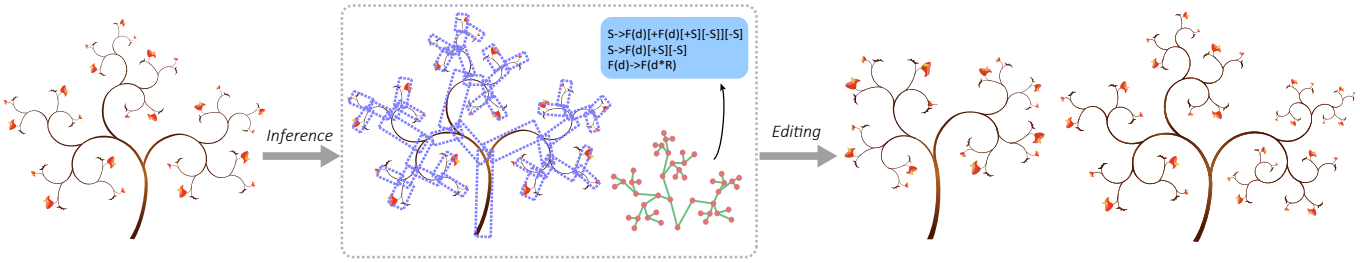


Fig. 1. Our algorithm analyzes the input image and infers a parametric L-system that represents it (d is the size of the detected structure and R is the scaling factor). The L-system represents the input and can be also used to produce its variations (right).

We introduce an inverse procedural modeling approach that learns L-system representations of pixel images with branching structures. Our fully automatic model generates a compact set of textual rewriting rules that describe the input. We use deep learning to discover atomic structures such as line segments or branchings. Orientation and scaling of these structures are determined and the detected structures are combined into a tree. The initial representation is analyzed, and repeating parts are encoded into a small grammar by using greedy optimization while the user can control the size of the detected rules. The output is an L-system that represents the input image as a simple text and a set of terminal symbols. We apply our approach to a variety of examples, demonstrate its robustness against noise and blur, and we show that it can detect user sketches and complex input structures.

CCS Concepts: • **Computing methodologies** → **Shape analysis**; • **Theory of computation** → *Grammars and context-free languages*; Rewrite systems.

*Corresponding author: Hui Huang (hhzhiyan@gmail.com)

Authors' addresses: Jianwei Guo, NLPR, Institute of Automation, CAS; Haiyong Jiang, UCAS, Beijing, NTU Singapore; Bedrich Benes, Purdue University; Oliver Deussen, SIAT Shenzhen and University Konstanz; Xiaopeng Zhang, NLPR, Institute of Automation, CAS; Dani Lischinski, The Hebrew University of Jerusalem; Hui Huang, College of Computer Science & Software Engineering, Shenzhen University.

0730-0301/2020/4-ARTxx $15.00
https://doi.org/0000001.0000001

Additional Key Words and Phrases: L-Systems, Grammar Induction, Procedural Generation

ACM Reference Format:
Jianwei Guo, Haiyong Jiang, Bedrich Benes, Oliver Deussen, Xiaopeng Zhang, Dani Lischinski, and Hui Huang. 2020. Inverse Procedural Modeling of Branching Structures by Inferring L-Systems. *ACM Trans. Graph.* xx, xx, Article xx (April 2020), 13 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Procedural modeling is one of the most powerful means for the automatic creation of digital content in computer graphics. A particular advantage of procedural modeling is in model amplification; the procedural rules may consist of only a small number of symbols and a few parameters, but the generated geometry can be millions of structures. While the geometry itself is large, its procedural representation is concise and easy to reuse. Moreover, the rules allow for generation of many variants of the geometry.

However, achieving a desired user intent, *i.e.,* obtaining a procedural model that would generate a specific geometry, is a difficult problem. A small change of parameters can exacerbate during the repeated applications of the procedural rules and quickly diverge in large modifications of the generated geometry. These intricate changes are difficult to understand and typically leave users working in a tedious trial-and-error loop often with parameterized procedural rules that are difficult to understand and control.

Another way to obtain a procedural model is by reverse engineering an existing geometry; the task is commonly referred to as inverse procedural modeling (IPM). Such methods would allow a significant

compression of given structures and an easy creation of variations. Previous approaches to IPM worked with input structures such as facades [Martinovic and Van Gool 2013], 2D arrangements [Ellis et al. 2018; Šťava et al. 2010], biological trees [Šťava et al. 2014], and urban modeling [Nishida et al. 2016]. However, most of the existing methods work with an existing procedural model and only adapt its parameters. The generation of a procedural model is an important open problem [Aliaga et al. 2016].

An important class of procedural models is Lindenmayer systems (L-systems) [Lindenmayer 1968], a mathematical formalism based on parallel string rewriting. L-systems generate linear branching structures and they have been used in a wide variety of applications (Section 2) with the prevailing domain being the simulation of biological branching structures [Prusinkiewicz and Lindenmayer 1990]. L-systems, however, share the common problem of procedural models: it is difficult to create the procedural rules in a goal-oriented way and to control the expansion process.

We present a novel algorithm that infers an L-system from a given image of a branching structure. We use a deep neural network to detect basic branching elements and their orientations, which range from line segments to simple branching patterns. Location, scale and orientation of these atomic structures are used to infer an initial grammar. Since branching angles and lengths might vary, we also detect the parameters of the procedural rules to describe the input. In the initial step of our approach the discovered grammar is a plain, parametric description of the image content. Higher-level branching rules are then generated by reducing the initial grammar to a compact representation. This process is performed by a greedy optimization and can be controlled by the user who can select between creating rules of larger length or a higher frequency of repetition. We claim the following main contributions:

- a novel algorithm to generate an L-system from an input image,
- a novel algorithm that uses a deep neural network to locate and describe branching structures by textual grammars, and
- a grammar compression framework that allows to infer a compact grammar from the located branching structures.

Figure 1 shows the main steps of our process. An input pixel image is analyzed, atomic structures are detected, and by inference an initial grammar is generated. This grammar is reduced by optimization and used to create variants of the generated structure.

2 RELATED WORK

2.1 Procedural Modeling, L-systems, and Vegetation

Procedural modeling is an important way to create complex objects such as trees and landscapes [Deussen and Lintermann 2010], façades [AlHalawani et al. 2013; Bao et al. 2013; Li et al. 2011; Martinovic et al. 2012; Müller et al. 2007; Shen et al. 2011; Wu et al. 2014; Zhang et al. 2013], buildings [Müller et al. 2006; Vanegas et al. 2010; Whiting et al. 2009], or whole cities [Parish and Müller 2001]. We refer the readers to a recent survey of procedural modeling [Smelik et al. 2014] and to a review of inverse modeling methods [Aliaga et al. 2016] for further information.

Lindenmayer [1968] invented the L-system as a parallel string rewriting system for describing cellular subdivision. Their theoretical properties have been outlined by Rozenberg and Salomaa [1980] and they were expanded by Prusinkiewicz [1986] by graphical interpretation using a logo-like turtle and a generation of branching structures. The L-systems is a robust formal framework for describing vegetation [Prusinkiewicz and Lindenmayer 1990] that was expanded to allow for continuous growth animation using parametric differential L-systems [Prusinkiewicz et al. 1993], or external control of the rewriting by the environment using Open L-systems [Měch and Prusinkiewicz 1996; Prusinkiewicz et al. 1994].

In recent years a number of publications focused on learning-based procedural modeling, where different aspects of such systems are learned. Yumer et al. [2015] use autoencoder networks to learn representative samples of procedural generation to guide the user in high-dimensional design spaces. Nishida et al. [2016] used a deep neural network to train simple 3D procedural parameterized volumetric models. During an interactive session, the user sketches models, the system recognizes them and combines them into 3D volumetric buildings with detailed façades, while keeping the underlying procedural representation. Huang et al. [2017] use networks to fit procedural models to user sketches for modeling buildings.

2.2 Inverse Procedural Modeling

The task of inverse generation of L-systems relates to grammar inference that is a well-studied problem in computer science [de la Higuera 2010]. Closely-related to our work is [McQuillan et al. 2018] with polynomial algorithms inferring context free and context sensitive grammars from input atoms. However, not much work addresses inverse procedural modeling of branching structures.

Šťava et al. [2010] present a framework for inverse procedural modeling of L-systems from vector data. They use a fixed set of parameters that are matched from 2D models by using symmetry detection and a voting scheme. The most frequently repeated patterns are encoded into a single rule that is converted into a hierarchical structure. While the algorithm of Šťava et al. [2010] requires a predefined set of rules and only finds the parameters of these rules, our algorithm is capable of generating new rules and their parameters. Moreover, their work requires the input to be in a vectorized form and the terminal symbols are defined a priory. Our work is more flexible in that it can be trained to detect terminal symbols from pixel images by using deep learning. In short, we seek to solve a more general problem of inverse procedural modeling in a broader way while the algorithm in [Šťava et al. 2010] detects only the parameters of a given branching model.

Bokeloh et al. [2010] analyze complex 3D shapes and describe their symmetries by using a novel inverse procedural model that connects the basic building blocks in a way that allows for quick variations. Talton et al. [2012; 2011] use a Markov Chain Monte Carlo (MCMC) optimization to produce a more general rule system from a specific one that describes given input scenes. Martinovic et al. [2013] use Bayesian learning to infer layout rules for creating certain types of buildings. Ritchie et al. [2015] improve the MCMC by introducing Sequential Monte Carlo (SMC) optimization, which allows to get feedback on complete and also partial models.

Inverse procedural modeling of façades has been introduced in [Xiao et al. 2008]. Structure from motion provides a rough shape that is proceduralized into rectilinear patches that are further organized into a regular procedural model. In [Wu et al. 2014] a split grammar is generated by using dynamic programming for an input façade and in [Zhuo et al. 2015] human intuition is used in a semi-interactive approach for inverse procedural modeling of façades.

Vanegas et al. [2010] use an L-system that incorporates only a single universal rule for the 3D reconstruction of Manhattan-style buildings. They match a floor pattern to the input data and represent the building as a floor-wise stack of procedural rules. Vanegas et al. [2012] create a procedural model of a city from user-defined high-level constraints such as the visibility of landmarks by optimizing the city geometry using MCMC. Šťava et al. [2014] create a parameterized procedural tree model including growth simulation. The parameters of the model (not the rules) are estimated by MCMC and optimized to fit a carefully developed metric that compares two tree models.

Sharma et al. [2018] combine a convolutional neural network and a recurrent neural network to predict constructive solid geometry or their operators. Tian et al. [2019] introduce *3D shape programs* to automatically infer 3D programs from given images or shapes by using a block LSTM and a step LSTM, the former is used to infer program blocks and the latter to infer the statements of each program block. Pairwise relations in scenes, *e.g.,* symmetry and repetitions, are explored with a group representation by [Liu et al. 2019]. Kalojanov et al. [2019] map shapes to strings and leverage a variational auto-encoder to learn variations of the strings rather than of complex 3D shapes. Most of these works aim to reconstruct programs from an input image or a 3D shape, and do not learn to generalize content. Furthermore, we focus on recursive structures, which may not be properly handled by these methods.

2.3 Neural Networks for Object Detection

CNN based object detection can be mainly split into one-stage and two-stage detectors. A seminal work about two-stage detectors is R-CNN [Girshick et al. 2014], such networks consist of a candidate proposal stage and a CNN-based region classification plus refinement stage. Later works boost the detection performance by adopting CNN-based region proposal networks [Ren et al. 2015], or using a feature pyramid network [Lin et al. 2017]. In general, two-stage detectors seem to achieve a high detection accuracy at high computational costs. Thus, one-stage detectors, *e.g.,* OverFeat [Sermanet et al. 2013] or SSD [Liu et al. 2016], YOLO [Redmon et al. 2016], have been proposed to speed up detection by directly classifying and refining bounding boxes of different scales and aspect ratios that are generated for tiled regions of the input image. The above works focus on the detection of axis-aligned bounding boxes. The work of Ellis et al. [2018] employs a neural network for detecting objects in a sketch and automatically creates a computer program that outputs the sketch. While this is not a procedural description in the classical sense, the approach creates a compact representation of the input.

In our work, however, we need to detect oriented bounding boxes to predict transformation parameters of the atomic structure of our rule-based description by adopting an improved version of the Faster R-CNN on oriented bounding boxes [Xia et al. 2018].

3 METHODOLOGY OVERVIEW

Before we describe how to infer an L-system that represents a branching structure from a given image, we shortly explain some of the main underlying concepts of our approach.

3.1 Lindenmayer Systems

L-systems [Lindenmayer 1968; Prusinkiewicz 1986; Prusinkiewicz and Lindenmayer 1990] are parallel string rewriting systems. An L-system $\mathcal{L}$ is a tuple

$$\mathcal{L} = \langle M, \omega, R \rangle \,, \tag{1}$$

where M is the L-system alphabet, ω is the axiom, and R is a set of *rewriting or production rules*. The alphabet contains parameterized *modules* $M = \{A(P), B(P), \dots\}$, where $P = p_1, p_2, \dots, p_n$ are module parameters such as translation, rotation, or scaling. The *axiom* $\omega \in M^+$ is a non-empty sequence of modules and M^+ is the set of all non-empty strings from M. The production rules in R have the following form:

$$id_1 : A(P) : \text{cond} \quad \rightarrow \quad x, x \in M^*, \tag{2}$$

$$id_2 : B(P) : \text{cond} \quad \rightarrow \quad x, x \in M^*, \tag{3}$$

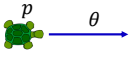
$$\dots$$

where M^* is the set of all possible strings from M including the empty string ε. A rule id_i rewrites the left-hand side character from the alphabet $A(P)$ by the sequence of letters of the right-hand side iff the *cond* is true. A module that does not appear on any left-hand side of a rule is called a *terminal symbol*, all other modules are called *non-terminals*.

While each letter has its production rule, the *derivation* of a string of modules is done by a parallel execution of applicable rules from set R to each letter it contains. We denote the derivation by $a \Rightarrow b$, where $a \in M^+$ and $b \in M^*$. The production rules rewrite the starting symbol by a sequence of modules and continue in successive derivations $\omega \Rightarrow m_1 \Rightarrow m_2 \Rightarrow \dots$ until either no module can be rewritten any more, *i.e.,* the string ends with a set of terminal symbols, the string of modules is empty due to the application of the epsilon rule or the execution is stopped after a user-specified number of iterations.

Recursion in L-system occurs if a symbol from a left-hand side of a rule occurs on the right-hand side of the same rule (even indirectly). **Non-determinism** allows multiple rules per character of the alphabet. It requires to additionally specify the probability of their application.

To create a geometry from a string of modules, each string is *interpreted* by a logo-like turtle that produces geometric symbols such as lines or even 3D geometry [Prusinkiewicz 1986].



In 2D the turtle has its state $S(\mathbf{p}, \theta)$ where $\mathbf{p} = [x, y]$ is its position and θ is the heading vector that identifies the direction of its motion.

The turtle reads the letters of the interpreted string of modules sequentially from the beginning and each letter is interpreted as a command. The letter $F(d)$ is interpreted as "move forward from

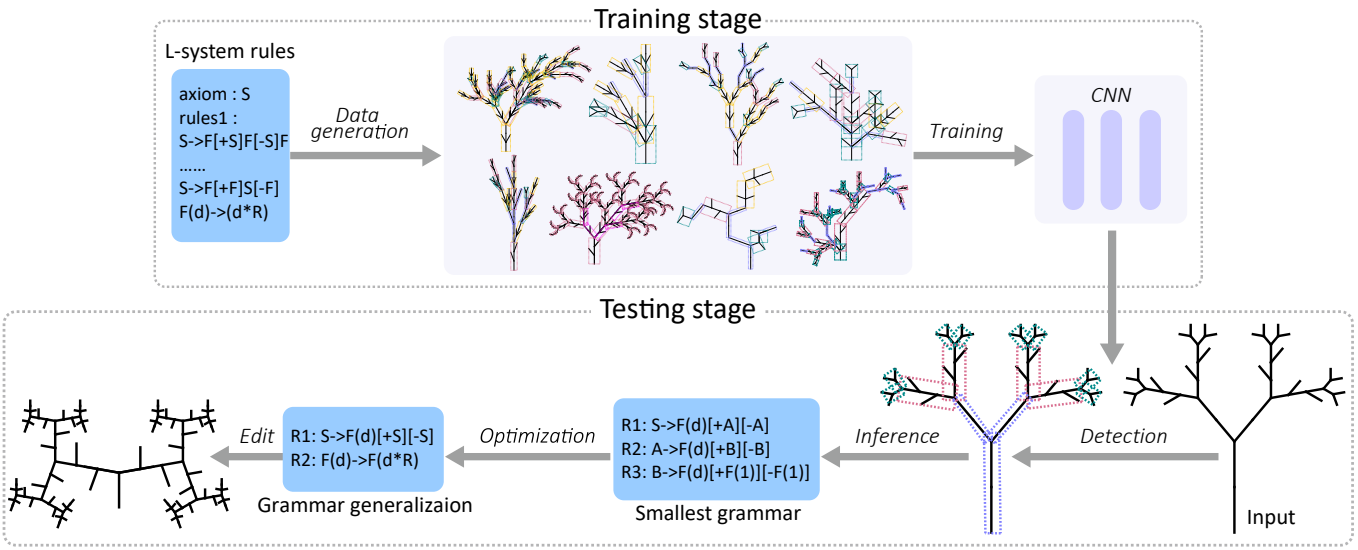


Fig. 2. Overview: During training, we generate large sets of data to train a CNN in order to detect atomic structures in an input image. During the subsequent inference based on the input data, the CNN detects atomic elements in an image, their transformations are determined and organized into a tree-like structure that is then combined into an output grammar using optimization.

p with heading θ by distance d and draw a line segment between the old and the new position", commands $+(\alpha)$ and $-(\alpha)$ change the heading of the turtle by turning to the left and right by α, respectively. Moreover, the turtle has a *stack*, the letter "[" pushes the state of the turtle to the stack and "]" pops the state back from the stack. Anything that is between the brackets $[M^+]$ is geometrically interpreted as a branch of the generated structure. Not all symbols from the alphabet need to have a geometric interpretation and if they do not the turtle ignores them.

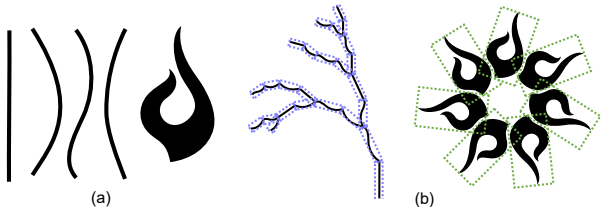


Fig. 3. Examples of templates (a) and their instances in input images (b).

3.2 Inferring L-Systems

The L-system inference is a three step process : 1) detection of atomic structures, 2) inference of an initial, and later compact, grammar, and 3) grammar generalization (see Fig. 2).

We first train a convolutional neural network (R-CNN) to detect atomic branching structures from 2D images. The training data is generated from predefined L-systems that produce a large number of training images. We produce linearly translated, scaled, as well as rotated branching elements and label them automatically. Once trained, the R-CNN detects the instances of atomic structures from the input test images. Along with that we detect their transformation parameters such as translation, scaling, and rotation.

In the next step, the detected atomic elements are organized into a structural pattern with higher complexity, which is encoded by a small grammar. This starts with constructing a tree-like data structure from the detected atomic elements by looking for their pair-wise distances. Each tree node corresponds to an atomic branch and each edge encodes the spatial transformation (translation and rotation) of each node relative to its parent. Our goal is to find an L-system that has a small number of modules by finding repetitions of identical topological structures within the tree. Moreover, we allow the user to modify this objective by specifying a preference to create rules of larger length or with a higher frequency of repetition.

In the last step, we introduce a cost function to merge similar rules in order to generalize the obtained L-system with non-deterministic and recursive rules. The cost function considers both the grammar length and grammar distance, where the grammar distance is computed by the string editing distance between the right hand sides (RHS) of the rules. A greedy strategy is employed to approximately minimize this cost function.

4 METHOD

Here we first describe how we detect atomic structures, then we explain our algorithm to infer the grammar, and finally the method to generalize it.

4.1 Detection of Atomic Structures

We regard our input images to consist of a collection of different atomic structures that we call *templates* (Fig. 3(a)). Each template occurrence denotes one of its *instances* (Fig. 3(b)), which may be a result of rotation, scaling, and translation of the template. A template can be a simple line corresponding to the F symbol (the basic step of

the turtle from Section 3.1) or a higher-level structure, *e.g.,* a simple branching pattern such as $F[+F][-F]$. Our aim is to automatically detect all instances as shown in the second stage of Fig. 2.

Data preparation: Existing object detection solutions require a large number of labeled examples to train a neural network in order to capture variants of an object and its surrounding scenes including overlaps. It is tedious and expensive to manually annotate such a large corpus of data. For our problem, we also assume to have no knowledge about the L-system rules.

We automatically synthesize a training dataset by using pre-defined templates and L-system rules. The templates include frequently-used atomic structures such as line segments of tree images, curve segments of sketches or elements of graphical designs, and can be provided by an artist (see Fig. 3(a)). Most instances of these templates only differ by their transformation parameters, *i.e.,* scaling, translations, and rotations, and will have only few changes in their appearances when approximated by the transformed templates. Thus, it should be possible to generate similar instances as those in the test images by randomly transforming our pre-defined branch templates.

However, spatial relations between instances still may be different between the test images and the synthesized training dataset. Though most spatial relations do not influence on the detection, overlaps and adjacency of instances do affect its results. Adjacency can be simulated by using pre-defined rules, while overlaps are generated randomly during the derivation of the pre-defined rules. In our algorithm, we use eight pre-defined templates (shown in Fig. 4), which cover different kinds of branching structures. To decouple the training and test data, we use different sets of pre-defined rules to generate the synthetic images. We further add randomness by selecting random combinations of the templates. Although the rules used for the training data are different from those of the test cases, they help to model adjacency as well as overlaps between instances and therefore reduce data gaps between the training and test phases.

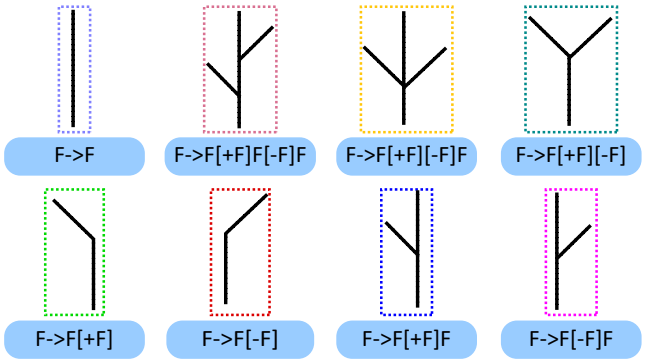


Fig. 4. Atomic structures: eight pre-defined rules together with their produced geometry used. The colors of the bounding boxes indicate the labels of the detected structures.

During data generation, we randomly select a subset of pre-defined rules and templates from our template basis. Training examples are generated by creating a branching structure from the rules at a random position. When we created the right hand side of a rule, we randomly rotated it around its parent node and scaled it with a random factor in the range $0.6-2.4$. A random transformation of the terminals produces enough variety for the training instances and furthermore creates a variety of random overlaps between them.

De-instancing: Next, we train a neural network to detect each instance and infer its corresponding geometric transformation. Inspired by [Šťava et al. 2010], we call this step de-instancing. We detect oriented bounding boxes which yield the rotation transformation (Fig. 3 (b)). The scale is obtained by comparing the detection results with its template, the translation is calculated from the center of the detected bounding box. The transformation is stored as a 3×3 matrix.

The detection of oriented instances is resolved by using a detection neural network based on Faster R-CNNs. This neural network is initialized by using the pre-training model ResNet-101 [He et al. 2016]. A Region Proposal Network (RPN) is applied to generate rotational bounding boxes. Then we use a multi-scale Region of Interest (ROI) align layer to extract pooled features, and through two fully connected layers, we conduct a position prediction and classification. The final result is obtained by an inclined non-maximum suppression. The novelty of our approach lies in the data synthesis and incorporation of a detection algorithm for L-system inference rather than the design of the detection network. It would be possible to replace the network with alternatives such as SSD [Liu et al. 2016] or YOLO [Redmon et al. 2016].

4.2 Inferring a Compact Grammar

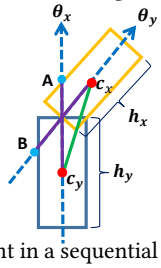
We define the *compact grammar* as the L-system of the shortest grammar length that generates only the given example. The grammar length is defined as the rule length and the number of used symbols.

The smallest grammar problem is NP-hard [Charikar et al. 2005], so proper heuristics have to be explored. Contrary to strings, the instances of a graphical L-system example are potentially scattered over the 2D space, and have to be properly grouped and represented for inferring a grammar. Moreover, some L-systems may have random parameters, *e.g.,* the heading direction θ in Sec. 3.1, so attention must be paid to make the process robust against noise. We developed a greedy optimization algorithm that attempts to always join nearby instances of the template to a rule.

During the grammar generation step we first group different instances as a tree structure by their distance. Then, a compact grammar is extracted while controlling the rule complexity and occurrence frequency. Finally, we use the orientation of each template and transfer it to the turtle states associated with each right hand side (RHS) symbol for each rule.

Construction of the n-ary tree: The detected instances (Sec. 4.1) are organized into an n-ary tree that corresponds to the derivation of a grammar with the starting symbols as root, terminal nodes as leaves, and non-terminals as internal nodes. Contrary to grammar expansion, we also have to deal with geometric information. Using the edges of the tree structure we represent each instance in the local coordinate frame of its parent node, which makes the later extraction of the turtle states $S(\mathbf{p}, \theta)$ (Sec. 3.1) more convenient.

To obtain this n-ary tree, we first assign instances of different templates to different labels. Next, we construct tree edges by linking close or adjacent instances, where each node denotes one instance. However, there are overlaps between the oriented bounding boxes even though their instances might not have a contact (see Fig. 5). The overlaps will lead an undirected graph with cycles, which would hamper the interference of the grammar. To avoid the generation of cycles, we define a new pairwise distance to predict the probability of connecting two bounding boxes in the graph. Since the goal is to generate the structure by using a procedural model, the adjacent instances are assumed to be connected by the turtle movement in a sequential way. Suppose we have two boxes with centers $\mathbf{c}_x$ and $\mathbf{c}_y$, heading directions θ_x and θ_y, and height h_x and h_y (see inset). We compute the projection distance of vector $\overrightarrow{\mathbf{c}_x\mathbf{c}_y}$ to the heading directions θ_x and θ_y, which are denoted as $d_{\mathbf{c}_x B}$ and $d_{\mathbf{c}_y A}$. The normalized pairwise distance between two bounding boxes is:



$$D_s(x,y) = \frac{d_{\mathbf{c}_x B} + d_{\mathbf{c}_y A} - (h_x + h_y)}{0.5(h_x + h_y)}. \quad (4)$$

This equation considers both the position and orientation (which indicates the turtle moving direction) of each instance. Usually, the smaller the value of $D_s(x,y)$, the more likely the two bounding boxes are connected. To avoid manually tuning the threshold of $D_s(x,y)$, we use a relatively large value ($D_s(x,y) = 1.2$) to filter out most of the unreasonable edges. Then we extract a minimum-weight spanning tree from the remaining edges and this way creates our final n-ary tree.

Finally, we convert the transformation of each node, which has been estimated in the de-instancing step, to the local frame of its parent. The transformation is decomposed into the three components: scaling, rotation, and translation that are stored as the rule parameters.

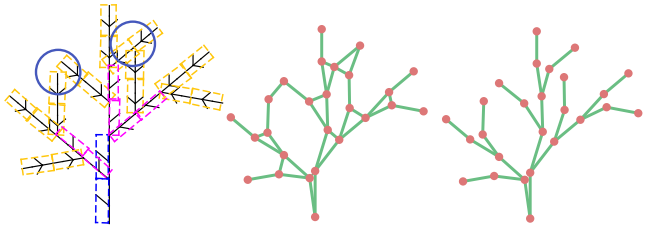


Fig. 5. An example of n-ary tree construction. The intersecting bounding boxes (left) would cause cycles (middle), while our method avoids by discarding close edge connections (right).

Grammar inference: Next we infer a compact grammar from the tree. At this step we do not consider parameters of the rules because they only change tree geometry, *e.g.,* branching angles. The expanded string corresponds to the same structure topology.

We introduce a user-controlled weighting parameter w_l into the optimization, which controls if rules of small length but with many repetitions should be preferred or large rules with a small number of repetitions. Our objective is to find a grammar $\mathcal{L}^+$ minimizing:

$$C_i(\mathcal{L}^+) = \sum_{A(P)\rightarrow M^* \in \mathcal{L}^+} w_l \cdot |M^*| + (1 - w_l) \cdot N(A(P) \rightarrow M^*), \quad (5)$$

where $|\cdot|$ computes the number of symbols in a rule, $N(\cdot)$ counts the repetitions of rule applications in the current example, w_l is the weighting parameter $w_l \in [0.0, 1.0]$, where $w_l = 1.0$ encodes the entire string as one rule, $w_l = 0.0$ finds all strings of length one, and $w_l = 0.5$ detects large substrings that are frequently repeated.

ALGORITHM 1: Grammar inference

Input: An n-ary tree T
Output: A compact grammar $\mathcal{L}^+$

$\mathcal{L}^+ = L_s$, L_s is the expanded string directly derived from T;
$\mathcal{L} = \emptyset$;
Set the weighting parameter w_l ;
Find the maximal sub-tree structure T' and its repetition n in T ;
while $n > 1$ **do**
  Replace all occurrence of T' with a same symbol;
  Extract rule set R for tree T;
  $\mathcal{L} = \mathcal{L} + R$;
  if $C_i(\mathcal{L}) >= C_i(\mathcal{L}^+)$ **then**
    break;
  end
  $T \leftarrow T'$, $\mathcal{L}^+ \leftarrow \mathcal{L}$;
  Find the maximal sub-tree structure T' and its repetition n in T ;
end

We solve this problem by performing a greedy search for maximal sub-trees within our n-ary tree instead of sub-strings of the example (Algorithm 1, see Section 1 in Supplemental Material for more details). This avoids the grouping of meaningless sub-strings, *e.g.,* $F[+$. Note we do not consider the n-ary tree itself as a maximal sub-tree. Our algorithm stops when the cost in Eqn. (5) does not decrease any more. In this work we prefer to create larger rules by setting $w_l = 0.5$ for all experiments (exceptions are mentioned).

Rule parameter estimation: As mentioned above, the compact grammar inferred in the previous step does not consider rule parameters. Each RHS symbol of a rule may have different turtle states, and even the same symbols in different rules or at different positions within the same rule may have different states. Thus we cluster instances of the same symbol due to its corresponding rule and position within the rule. The turtle state is then estimated by averaging the different instances of each cluster to eliminate noise caused by inaccurate parameter prediction in the de-instancing step.

4.3 Grammar Generalization

So far we have extracted a compact grammar of branching patterns from images. However, the inferred grammar has limited expressiveness because it can only reproduce the input image. We further improve the grammar by adding non-deterministic rules and detecting recursive structures (see Sec. 3.1).

We extend the extracted grammar with variations by merging similar rules. Each merging operation takes two rules, collapses their left hand side (LHS) to a single symbol, assigns these two rules an

equivalent probability, and replaces all occurrences of the original LHSs. For example, the rules from Eqns. (2) and (3) are merged to:

$$AB(P_A) \xrightarrow{1,cond} M_A*, \quad (6)$$

$$AB(P_B) \xrightarrow{2,cond} M_B*, \quad (7)$$

where $A(P)$ and $B(P)$ are LHSs and the RHSs are replaced by one new common module $AB(P)$.

This problem poses two main challenges: first, each merging operation generalizes the grammar, but it may also initiate possible invalid variations. Therefore a proper metric is required to guide the merging process. Second, a grammar may produce a large number or even infinite examples with large structural or geometric variations, so it is both expensive and difficult to determine the feasibility of a merging operation based on their derived examples. Previous works [Martinovic and Van Gool 2013; Talton et al. 2012] examine the merging fitness by using a Bayesian model from a small number of examples, but this is still quite costly. Instead, we examine the merging operations based on the rules themselves. We optimize a general grammar $\mathcal{L}^*$ starting from an initial smallest grammar $\mathcal{L}^+$ (Sec. 4.2) by leveraging the grammar length $L(\mathcal{L}^*)$ and the grammar edit distance resulting from a distance $D_g(\mathcal{L}^+, \mathcal{L}^*)$ between two rules. We attempt to minimize:

$$C_g(\mathcal{L}^*, \mathcal{L}^+) = w_0 \cdot (L(\mathcal{L}^*) - L(\mathcal{L}^+)) + (1 - w_0) \cdot D_g(\mathcal{L}^+, \mathcal{L}^*), \quad (8)$$

where w_0 is a weight balancing the two metrics. By merging a rule, the grammar length will become shorter, but the rule edit distance will get larger. During optimization, the grammar length measures the number of symbols that represent it:

$$L(\mathcal{L}) = |M| + \sum_{A(P) \to M^* \in \mathcal{L}} |M^*|, \quad (9)$$

where $|\cdot|$ is the size of a set of symbols.

The grammar edit distance $D_g(\mathcal{L}^+, \mathcal{L}^*)$ determines the overall cost to convert a grammar $\mathcal{L}^+$ to $\mathcal{L}^*$ by a set of merging operations $M(\mathcal{L}^+ \to \mathcal{L}^*)$ as follows:

$$D_g(\mathcal{L}^+, \mathcal{L}^*) = \sum_{(A(P) \to M_A^*, B(P) \to M_B^*) \in M(\mathcal{L}^+ \to \mathcal{L}^*)} D_s(M_A^*, M_B^*), \quad (10)$$

where $D_s(M_A^*, M_B^*)$ estimates the edit distance [Navarro 2001]. We use three edit operations: string replacement, insertion, and deletion, with empirical weights 1, 1.5, 1.5. The edit distance can be calculated by various algorithms [Hirschberg 1975; Schulz and Mihov 2002], we use dynamic programming.

There is no closed-form solution for Eqn. (8) because of its discrete form and usage of edit distance, and the solution space is of combinational complexity. We employ the greedy algorithm to seek an approximate solution (Algorithm 2). We first generate all possible merging rules as candidates. Then, we evaluate the cost of each merging operation by Eqn. (8), the two rules with minimal cost will be merged. We iterate through this procedure until the cost does not decrease any more.

ALGORITHM 2: Grammar generalization

Input: A compact grammar $\mathcal{L}^+$
Output: A generalized grammar $\mathcal{L}^*$

```
Initialize the merging pair p* = ∅;
L* = L+;
C_g^old = C_g(L* + {p*}, L*);
do
    Generate all possible merging rule pairs P in L*;
    Find a pair p* with the minimal C_g(L* + {p_i}, L*), ∀p_i ∈ P;
    if C_g(L* + {p*}, L*) >= 0 then
        break;
    end
    c* = C_g(L* + {p*}, L*) − C_g^old;
    C_g^old = C_g(L* + {p*}, L*);
    L* = L* + {p*};
while c* <= 0;
```

5 RESULTS AND EVALUATION

This is a first paper dealing with the generation of L-systems from images. In this section, we show the evaluation of our algorithm on structures generated by L-systems selected from [Prusinkiewicz and Lindenmayer 1990], on synthetic images, real-world images, hand drawings, and images created manually by artists.

All experiments were conducted on a desktop computer equipped with an Intel i7 Xeon processor @2.1 GHz, 32 GB of RAM, and an NVIDIA GeForce TITAN X graphics card. We implemented our detection algorithm in TensorFlow and Python. The grammar inference was implemented in C++. The detection network started training with a learning rate of 0.0003, and was decreased by a factor of 10 every time when the validation loss started to oscillate. The training process took about 18 hours for 12 thousands epochs, while the inference process needs 0.12 seconds per image.

5.1 Robustness of Atomic Structure Detection

We evaluated the robustness of our algorithm on three tasks: 1) varying branching parameters, 2) blur, and 3) instance detection.

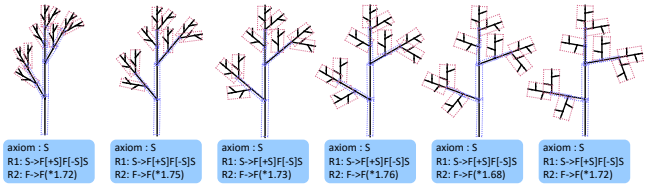


Fig. 6. Detection with varying branching angles (left to right: 30°, 40°, 50°, 60°, 70°, 80°).

Robustness w.r.t. different branching parameters: we tested our method by varying the branching angle and scale. Fig. 6 shows that the same instances can be detected when branching angle varies within a large range (from 30° to 80°). Moreover, (Fig. 7) we randomly changed branching angles and scaling at the same time (angle: $[-20°, 20°]$, scaling: $[0.7, 1.7]$). All instances were successfully detected and we achieved an average precision of 87% in all tests (including 40 test images). Here "precision" means the instance is assumed to be successfully detected if its detected angle and scaling parameters are below the angle and scaling thresholds 10° and 0.7, respectively.

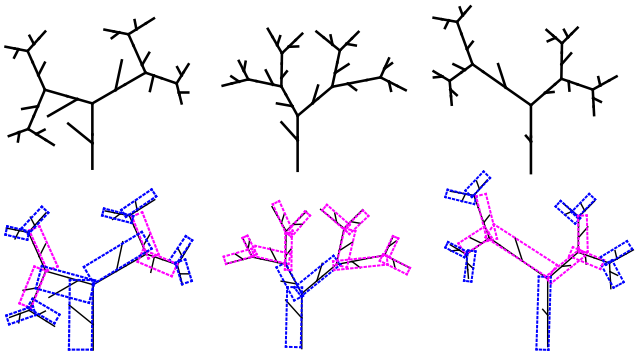


Fig. 7. Detection (bottom) of structures from input images (top) with random branching angles and scales.

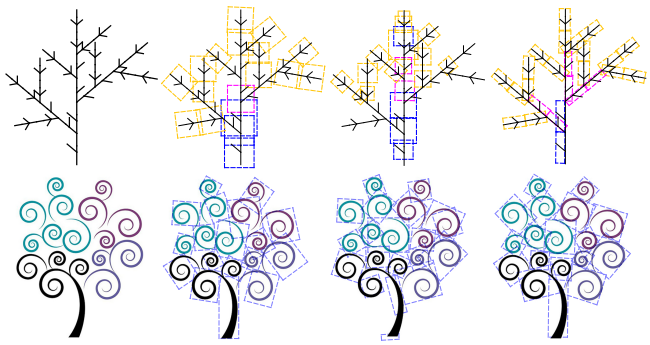


Fig. 8. Comparison of instance detection methods (left-right): input image, detection results for template matching [Brunelli 2009], generalized Hough transformation [Mata et al. 1999], and our deep learning-based method.

Robustness to blur: We generated test images by applying average blur, Gaussian blur, and downsampling (with scaling factors 0.8 and 0.5). We have 400 test images for each configuration. We utilized *precision*, *recall*, and detection *mean average precision* (mAP) [Girshick et al. 2014] to measure the quality of the detection algorithm. Table 1 shows results of this experiment. Our detection model is robust to different filtering approaches: for downsampled images we detected the instances even if the images were reduced by 50%.

Table 1. The detection performance on blurred and downsampled images.

Blurring Level	Recall (%)	Precision (%)	mAP (%)
Averaging blur	80.7	75.4	70.2
Gaussian blur	78.9	73.5	69.3
Downsampling ×0.8	81.5	76.3	71.8
Downsampling ×0.5	63.9	61.1	56.6

Instance detection alternatives: We used a state-of-the-art deep learning-based method for de-instancing. Traditional methods, *e.g.*, template matching [Brunelli 2009] and generalized Hough transformation [Ballard 1981] could be also used, but they rely on a predefined set of hand-tuned parameters. Fig. 8 shows a comparison of the deep learning-based method to the two alternatives implemented in OpenCV [Bradski 2000]. The traditional methods do not provide comparable results for rotated or scaled objects and the presence of noise in the image affects their performance.

5.2 Grammar Inference on Synthetic Images

Firstly, we evaluate the grammar inference by using synthesized images with ground-truth rules. We show grammar length $L(\mathcal{L})$, branching angle, and scaling factor of the compact and the generalized grammar for each example. Moreover, we also show the string length L_s of the original example, which is the expanded string only representing the input image itself. Finally, we report the Hausdorff distances of the shape between the input and the output structures.

We reproduced branching structures using L-systems from the book of [Prusinkiewicz and Lindenmayer 1990] and we compared the detected L-systems to the originals. Also, we evaluated whether our algorithm was able to infer the same L-system when we changed parameters of the input L-system. The varying branching angles mostly lead to the same L-system, as expected (Fig. 6). Also, our algorithm was able to compress the original examples encoded by 41 characters to a grammar with a length of 18 and 2 rules.

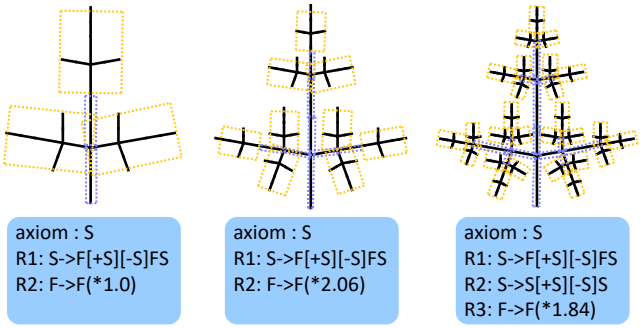


Fig. 9. Detection and grammar inference results on images generated with different number of iterations (3-5).

A varying number of produced iterations from a single grammar should, in theory, lead to the discovery of the same grammar. Three different iterations of the same grammar were generated and afterwards our detection algorithm was applied (Fig. 9). The result shows that we infer the correct ground truth L-system from all three images. Note that we obtained an additional rule in the last iteration, because the increasing number of iterations makes the line segments or branching structures smaller that makes them more difficult to detect.

In Fig. 10, we show a more complex example, where the ground-truth grammar contains three production rules and 36 symbols. The input is produced at the level of four iterations. Our algorithm output the correct L-system and reproduced its structure.

In addition, we tested our algorithm on a large synthesized dataset consisting of 150 images. Fig. 11 illustrates some examples of the dataset as well as our inferred L-system. We reported all results of the test examples in the supplemental material. It demonstrates that we are able to use only a few rules for representing a collection of images. Four examples of detection results and reproduced images

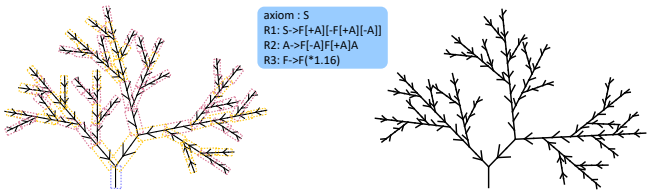


Fig. 10. Our algorithm infers correct grammar from a complex example (left) and reproduces its structure (right).

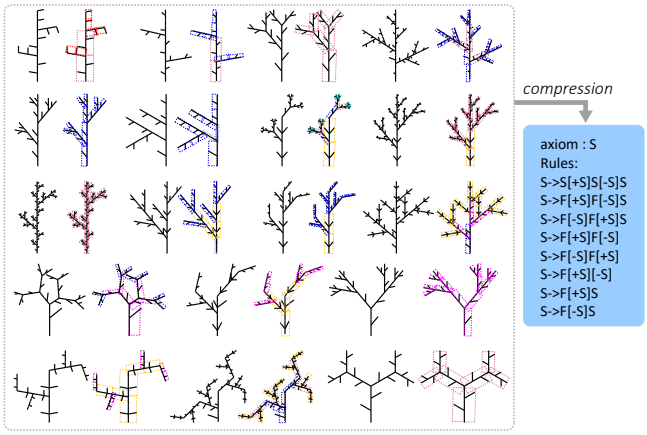


Fig. 11. Subset of 150 synthesized examples and inferred L-system.

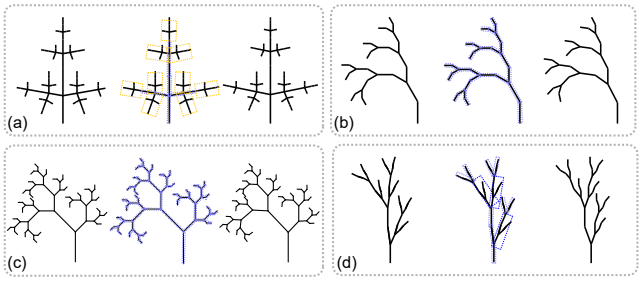


Fig. 12. Examples of grammar inference (left-right): input, detection result, and our reproduced results using the inferred L-system.

from our inferred L-systems are shown in Fig. 12. Our algorithm correctly reproduced the examples in Fig. 12(a-c). In Fig. 12(d), the reproduced result differs a bit from the original input because of an inaccurate rotation estimation in the first branch and complex turning commands (see below). Statistics shown in Tab. 2 show that our algorithm achieved the same compression rate in three out of four examples and is able to estimate branching angles with an error within 3° and the scaling factor of the branches with an error of 0.17. Over the whole synthesized dataset, our mean angular error is 2.56°, our mean relative scale error is 0.14, and our mean Hausdorff distance is 8.06. The original L-system from Fig. 12(d) has multiple turning commands in a single rule ($S \rightarrow F[+F][---S]+F-F[+++S]-S$), which is represented as one turning command in our inferred grammar, not precisely reproducing the input.

Table 2. Grammar statistics: string length of original input L_s, grammar length of compact grammar $L(\mathcal{L}^+)$, generalized grammar $L(\mathcal{L}^*)$, and ground truth grammar $L(\mathcal{L})$. Branching angles θ and scaling factors s of the ground truth and inferred grammar. The Hausdorff distance d_H (normalized by dividing its value by the diagonal length of the input bounding box) between the input and output structures. "SA" and "GA" represent simulated annealing and genetic algorithm, respectively.

		compact	generalized				ground truth		
Fig.	L_s	$L(\mathcal{L}^+)$	$L(\mathcal{L}^*)$	$\theta(\circ)$	s	$d_H(\%)$	$L(\mathcal{L})$	$\theta(\circ)$	s
12(a)	41	24	12	77.8	2.06	2.98	12	80	1.95
12(a)-SA	41	24	12	77.8	2.06	2.98	12	80	1.95
12(a)-GA	41	24	12	77.8	2.06	2.98	12	80	1.95
12(b)	87	48	14	27.4	1.34	3.28	14	25	1.40
12(b)-SA	87	52	14	29.1	1.28	3.74	14	25	1.40
12(b)-GA	87	48	14	27.4	1.34	3.28	14	25	1.40
12(c)	297	60	18	43.1	1.17	13.7	18	40	1.20
12(c)-SA	297	61	29	41.9	1.28	15.2	18	40	1.20
12(c)-GA	297	60	18	43.1	1.17	13.7	18	40	1.20
12(d)	45	32	23	20.0	1.71	5.78	25	8	1.70
12(d)-SA	45	40	28	21.4	1.65	7.29	25	8	1.70
12(d)-GA	45	37	25	18.8	1.59	6.34	25	8	1.70

Grammar inference alternatives: We implemented two other heuristic algorithms for grammar inference: simulated annealing (SA) and genetic algorithm (GA). The input to SA and GA is the expanded string generated from the geometries of our n-ary tree. Then we find all repeated sub-strings and perform the algorithms to optimize Eqn. (5). Note the meaningless sub-strings (*e.g.,* $+$, $+F][-F$) are not considered that greatly reduces the search space. Inference results on examples of Fig. 12 are listed in Tab. 2. Although SA and GA are also able to find good approximate solutions, they usually need a relatively large number of iterations. They also tend to produce more compact rules than our greedy approach. Considering the complex input in Fig. 10, our method outputs three rules using 0.135 seconds. In comparison, SA takes 1.652 seconds to generate seven rules, while GA takes 3.749 seconds to generate four rules.

We further compare to a greedy grammar inference algorithm, called Sequitur algorithm[1] [Nevill-Manning and Witten 1997], which infers a hierarchical structure from a sequence of discrete symbols by replacing repeated phrases. Since this method does not consider the meaningless sub-strings, it generates more production rules for our problem (see Fig. 12(b) for example, the number of generated compact rules by Sequitur is 12 while ours is four).

5.3 Grammar Inference on Non-synthetic Images

We further evaluated grammar inference by challenging examples: 1) user sketches, 2) real-world images, 3) images created by artists.

User Sketches: We tested our method on sketched inputs with hand-drawn curves and scribbled lines. The steps of our algorithm and potential variations of the inferred L-system are shown in Fig. 13, and Fig. 14 demonstrate our results on two more aggressively hand-drawing examples. Our algorithm detected most instances and the output grammar generates branching structures similar to the input, even though the hand-drawings are very rough and contain

[1] An implementation: https://github.com/craignm/sequitur

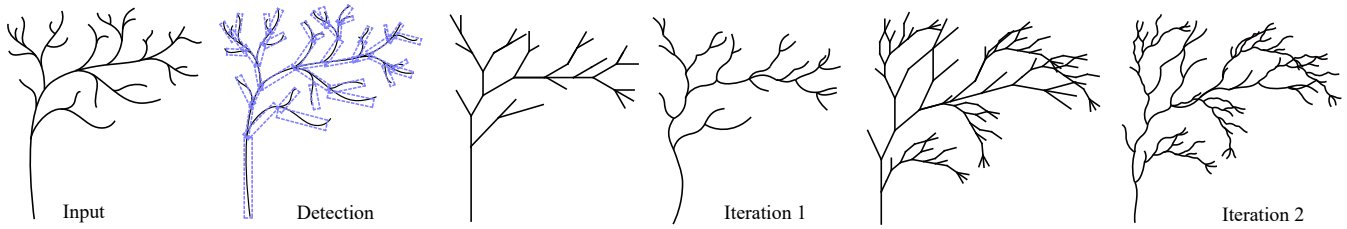


Fig. 13. Grammar inference from user sketches (left-right): input, detection result, and our re-produced results with two different iterations. For each iteration, we automatically select random user-drawn curves to replace straight line segments.

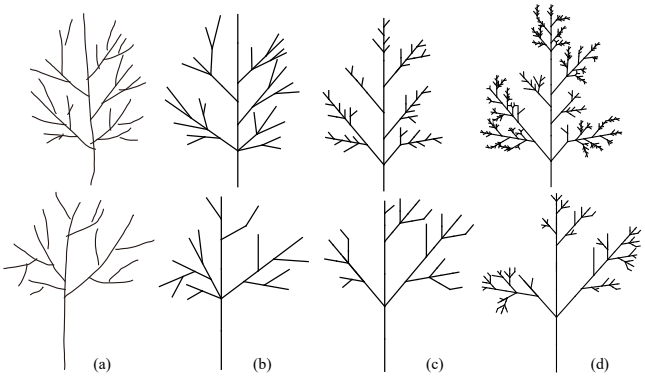


Fig. 14. Other sketch examples. Given an input hand-drawing image (a), we first use the compact grammar to pre-produce the result (b) similar to the input. Then we use our generalized grammar to generate results with two different derivation iterations in (c) and (d) .

disconnected line segments or curves. More results on user sketches are provided in the supplemental materials.

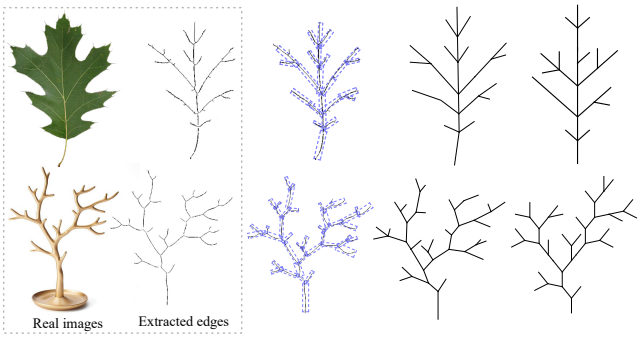


Fig. 15. Real-world images (left to right:) input images, edges, detection results, reconstructed structures, and generated results with pattern variations and parameter regularizations.

Real-world images: Our method has been applied to real-world images (Figs. 15 to 17). In Fig. 15, we automatically extracted almost one pixel-wide edges by using the Canny edge detector and morphological operations. Even when the extracted edges are noisy, our algorithm is able to reconstruct the skeleton and to generate a number of variations.

Furthermore, we evaluated our method on three irregular networks; one generated by the Dendry algorithm [Gaillard et al. 2019] and two by the algorithm of [Zhang and Guilbert 2016] (Fig. 16). The input structures incorporated randomness on the pixel level that is difficult to reproduce by an L-system. Our algorithm mimicked this by adding Perlin noise [Perlin 2002] to the detected segments. We also generated variations with segments differing in width and color based on the Strahler stream order.

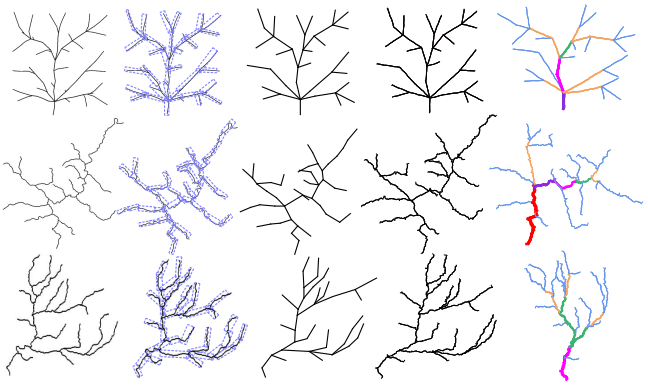


Fig. 16. Three river network images as input: top [Gaillard et al. 2019], middle and bottom [Zhang and Guilbert 2016]. Left to right: input, detected result, reconstructed straight branches, structure with adding Perlin noise, and branching with random parameter variations where the segment width and colors are assigned according to the Strahler stream order.

Another example in Fig. 17 shows a Lichtenberg image and a stream network delineation from a digital elevation model of a real river. These examples were more challenging because the images are blurry and have varying branch width and color. Although we did not have such data available for training, our neural network detected most of the stream segments and our grammar inference captured the branching structure well.

Artistically designed images: By training the detection algorithm with complex patterns, our algorithm can be applied to extract L-systems from artistic designs such as the examples in Fig. 18. These L-systems can be then used to synthesize structural variations of the branching styles.

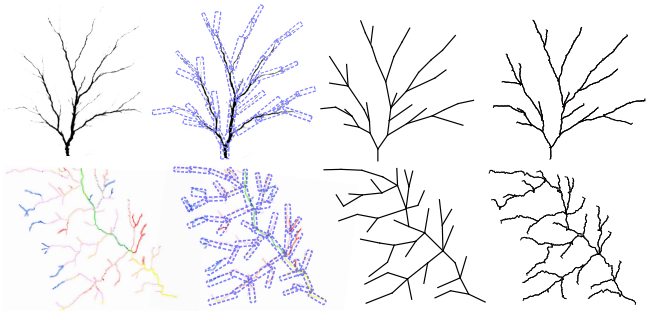


Fig. 17. Lichtenberg image (top) and a stream network (bottom). Left to right: input, detected result, reconstructed straight branching pattern and branches after adding Perlin noise.

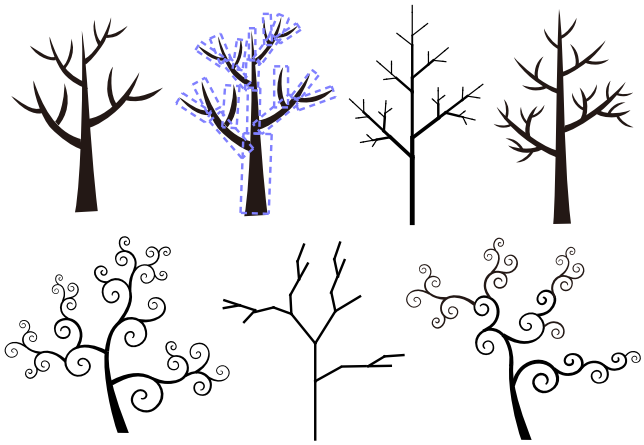


Fig. 18. Left: input artistic images. Middle: derivation results using our inferred grammar. Right: the re-produced designs by manually positioning the patterns according to the inferred branching structures (middle).

5.4 Applications via Rule Editing

Our algorithm can also aid synthesis of various plants and patterns by rule editing. Given an L-system, we synthesized various results by adjusting rule parameters such as branching angles or scaling factors, or modifying a part of a production rule as shown in Fig. 19.

Pattern layout variations: we furthermore synthesized complex decorative patterns using the detected grammars from some input images (Fig. 20). The inputs were analyzed and the rules were determined, then the grammars were manually edited and parameters were changed to generate different layouts based on the inputs.

5.5 Limitations

We use a neural network to detect instances and the generalization of the detector is limited by the richness of the training dataset. In order to detect images with unseen examples, the user needs to provide a library of possible templates, and re-train the detector. Our method may fail when there are significant occlusions between instances or instances become very small after deep recursions.

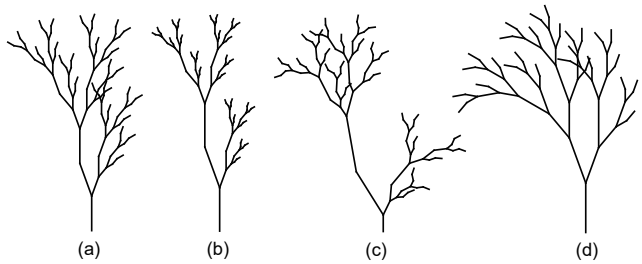


Fig. 19. Editing the rule from Fig. 12 (d): (a) original rule; (b) changed branch scale; (c) randomly changed branching angle and scale; (d) modifying a subpart of the rule creates a quite different branching structure.

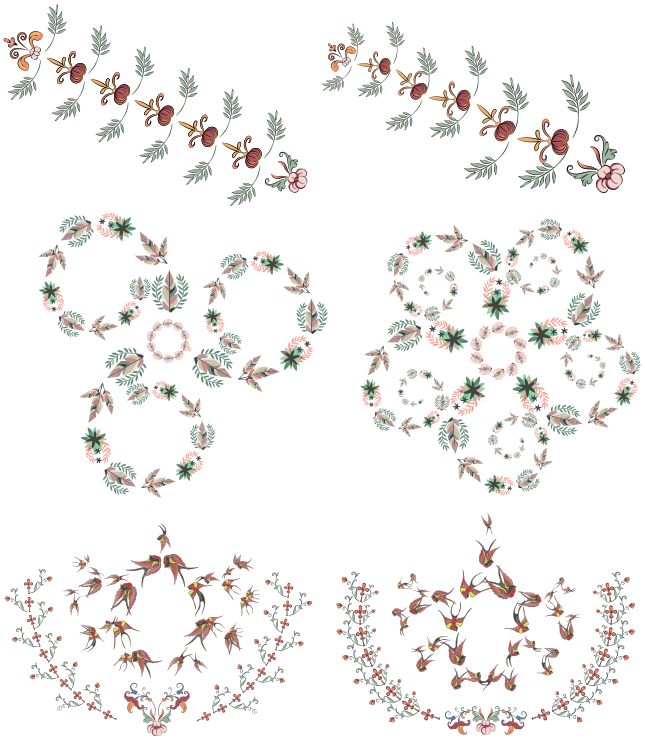


Fig. 20. Synthesis of layout variations: Left: input layouts that are analyzed and coded as L-systems. Right: generated new layouts via rule editing.

For example, in Fig. 21 it is difficult for the detector to correctly detect all instances. In addition, our detection system mostly focuses on curvilinear drawings and is not specifically designed to infer other semantic indications artists might use such as thickness and colorations of strokes used to delineate parts (*e.g.*, trunk vs branches of a tree), or textural cues such as a bumpy boundary used to indicate a leaf canopy.

Our application to real-world photographs was conducted on extracted skeletons of these photographs, because training a detection network on such photographs requires thousands of images with annotations, which is difficult to collect and out of the scope of this

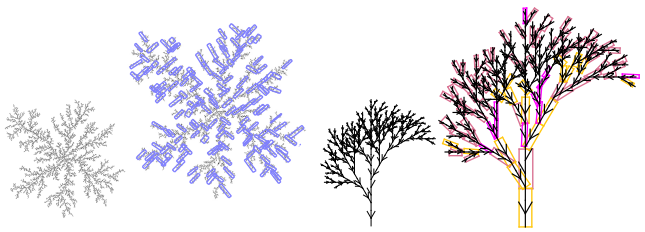


Fig. 21. Failure examples: our method cannot handle very small instances (left) and significant occlusion (right).

paper. Extension to directly handle very complex real-world images is challenging future work.

Lastly, the way we build the tree representation of an input prefers proximity of the objects and ignores global symmetries and relations. It would be interesting to combine our approach with approaches that consider global properties [Bokeloh et al. 2010; Guerrero et al. 2016; Šťava et al. 2010].

6 CONCLUSION AND FUTURE WORK

We have addressed one of the most important open problems of procedural modeling, *i.e.,* finding a procedural representation of a given input image of trees with branching structures. We tackled this problem by introducing a novel approach that analyses an input image, detects a set of atomic elements by using deep learning, and encodes the content into an L-system representation by using greedy optimization that finds a compact grammar. Combining a detector and optimization algorithm requires proper handling of training data, variations, grouping as well as handling overlaps of element instances. We have demonstrated our approach on a variety of examples, including both synthesized and non-synthesized images. The inferred L-systems can be easily edited and used for distributing other patterns.

In the future we will work on the two main limitations of our approach: extending it to support 3D structures and to allow for describing closed objects, or structures with loops.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments. This work was supported in parts by National Key R&D Program (2018YFB2100602), NSFC (61861130365, 61761146002, 61802406, 61802362), GD Higher Education Key Program (2018KZDXM058), LHTD (20170003), GD Leading Talent Program (00201509), DFG (422037984), NSF (10001387), FAR (602757), and GD Laboratory of Artificial Intelligence and Digital Economy (SZ).

REFERENCES

Sawsan AlHalawani, Yong-Liang Yang, Han Liu, and Niloy J. Mitra. 2013. Interactive Facades Analysis and Synthesis of Semi-Regular Facades. *Comp. Gr. Forum* 32, 2:2 (2013), 215–224.

Daniel G. Aliaga, İlke Demir, Bedrich Benes, and Michael Wand. 2016. Inverse Procedural Modeling of 3D Models for Virtual Worlds. In *ACM SIGGRAPH 2016 Courses (SIGGRAPH '16)*. ACM, New York, NY, USA, Article 16, 316 pages.

Dana H. Ballard. 1981. Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition* 13, 2 (1981), 111–122.

Fan Bao, Michael Schwarz, and Peter Wonka. 2013. Procedural facade variations from a single layout. *ACM Trans. Graph.* 32, 1, Article 8 (Feb. 2013), 13 pages.

Martin Bokeloh, Michael Wand, and Hans-P. Seidel. 2010. A connection between partial symmetry and inverse procedural modeling. In *ACM Trans. on Graph.*, Vol. 29. 104.

Gary Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

Roberto Brunelli. 2009. *Template Matching Techniques in Computer Vision: Theory and Practice.* Wiley Publishing.

Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. 2005. The smallest grammar problem. *IEEE Trans. Information Theory* 51, 7 (2005), 2554–2576.

Colin de la Higuera. 2010. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, New York, NY, USA.

Oliver Deussen and Bernd Lintermann. 2010. *Digital Design of Nature: Computer Generated Plants and Organics.* Springer Publishing Company, Incorporated.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to infer graphics programs from hand-drawn images. In *NIPS*. 6060–6069.

Mathieu Gaillard, Bedrich Benes, Eric Guérin, Eric Galin, Damien Rohmer, and Marie-Paule Cani. 2019. Dendry: A Procedural Model for Dendritic Patterns. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '19)*. 16:1–16:9.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *IEEE CVPR*. 580–587.

Paul Guerrero, Gilbert Bernstein, Wilmot Li, and Niloy J. Mitra. 2016. PATEX: exploring pattern variations. *ACM Trans. Graph.* 35, 4 (2016), 48:1–48:13.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE CVPR*. 770–778.

Daniel S. Hirschberg. 1975. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Commun. ACM* 18, 6 (1975), 341–343.

Haibin Huang, Evangelos Kalogerakis, Ersin Yumer, and Radomir Mech. 2017. Shape synthesis from sketches via procedural models and convolutional networks. *IEEE Trans. on Vis. and Comp. Graphics* 23, 8 (2017), 2003–2013.

Javor Kalojanov, Isaak Lim, Niloy Mitra, and Leif Kobbelt. 2019. String-Based Synthesis of Structured Shapes. *Comp. Gr. Forum* 38, 2 (2019), 027–036.

Yangyan Li, Qian Zheng, Andrei Sharf, Daniel Cohen-Or, Baoquan Chen, and Niloy J. Mitra. 2011. 2D-3D fusion for layer decomposition of urban facades. In *Proceedings of the 2011 International Conference on Computer Vision (ICCV '11)*. 882–889.

Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan, and Serge J. Belongie. 2017. Feature Pyramid Networks for Object Detection. In *IEEE CVPR*. 936–944.

Aristid Lindenmayer. 1968. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology* Parts I and II, 18 (1968), 280–315.

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. In *ECCV*. 21–37.

Yunchao Liu, Jiajun Wu, Zheng Wu, Daniel Ritchie, William T. Freeman, and Joshua B. Tenenbaum. 2019. Learning to Describe Scenes with Programs. In *International Conference on Learning Representations*.

Andelo Martinovic, Markus Mathias, Julien Weissenberg, and Luc J. Van Gool. 2012. A Three-Layered Approach to Facade Parsing. In *ECCV (7) (Lecture Notes in Computer Science)*, Vol. 7578. Springer, 416–429.

Andelo Martinovic and Luc Van Gool. 2013. Bayesian grammar learning for inverse procedural modeling. In *IEEE CVPR*. 201–208.

Nicolás Guil Mata, José María González-Linares, and Emilio L. Zapata. 1999. Bidimensional shape detection using an invariant approach. *Pattern Recognition* 32, 6 (1999), 1025–1038.

Ian McQuillan, Jason Bernard, and Przemyslaw Prusinkiewicz. 2018. Algorithms for Inferring Context-Sensitive L-Systems. In *International Conference on Unconventional Computation and Natural Computation*. Springer, 117–130.

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

Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. 2007. Image-based Procedural Modeling of Facades. *ACM Trans. on Graphics* 26, 3, Article 85 (2007).

Radomír Měch and Przemyslaw Prusinkiewicz. 1996. Visual models of plants interacting with their environment. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, New York, NY, USA, 397–410.

Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM Comput. Surv.* 33, 1 (2001), 31–88.

Craig G Nevill-Manning and Ian H Witten. 1997. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research* 7 (1997), 67–82.

Gen Nishida, Ignacio Garcia-Dorado, Daniel G Aliaga, Bedrich Benes, and Adrien Bousseau. 2016. Interactive sketching of urban procedural models. *ACM Trans. on Graphics (Proc. SIGGRAPH)* 35, 4 (2016), 130.

Yoav I. H. Parish and Pascal Müller. 2001. Procedural Modeling of Cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. 301–308.

Ken Perlin. 2002. Improving noise. In *ACM Trans. on Graphics*, Vol. 21. 681–682.

Przemyslaw Prusinkiewicz. 1986. Graphical Applications of L-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86*. 247–253.

Przemyslaw Prusinkiewicz, Mark S. Hammel, and Eric Mjolsness. 1993. Animation of plant development. In *SIGGRAPH '93: Proc. of the 20th Annual Conf. on Computer graphics and interactive techniques*. ACM Press, 351–360.

Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. 1994. Synthetic topiary. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. ACM Press, New York, NY, USA, 351–358.

Przemyslaw Prusinkiewicz and Aristid Lindenmayer. 1990. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, USA.

Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *IEEE CVPR*. 779–788.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *NIPS*. 91–99.

Daniel Ritchie, Ben Mildenhall, Noah D Goodman, and Pat Hanrahan. 2015. Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo. *ACM Trans. on Graphics* 34, 4 (2015), 105.

G. Rozenberg and A. Salomaa. 1980. *The Mathematical Theory of L Systems*. Academic Press, Inc., New York.

Klaus U. Schulz and Stoyan Mihov. 2002. Fast string correction with Levenshtein automata. *IJDAR* 5, 1 (2002), 67–85.

Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. 2013. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. *CoRR* abs/1312.6229 (2013). arXiv:1312.6229

Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2018. CSGNet: Neural Shape Parser for Constructive Solid Geometry. In *IEEE CVPR*.

Chao-Hui Shen, Shi-Sheng Huang, Hongbo Fu, and Shi-Min Hu. 2011. Adaptive partitioning of urban facades. *ACM Trans. Graph.* 30, 6, Article 184 (Dec. 2011), 10 pages.

Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. 2014. A Survey on Procedural Modelling for Virtual Worlds. *Comp. Gr. Forum* 33, 6 (2014), 31–50.

Jerry Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah Goodman, and Radomír Měch. 2012. Learning design patterns with bayesian grammar induction. In *UIST*. ACM, 63–74.

Jerry O Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. 2011. Metropolis procedural modeling. *ACM Trans. on Graphics* 30, 2 (2011), 11.

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *International Conference on Learning Representations*.

Carlos A. Vanegas, Daniel G. Aliaga, and Bedrich Benes. 2010. Building reconstruction using manhattan-world grammars. *IEEE CVPR* (2010), 358–365.

Carlos A. Vanegas, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Paul Waddell. 2012. Inverse Design of Urban Procedural Models. *ACM Trans. Graph.* 31, 6, Article 168 (Nov. 2012), 11 pages.

Ondřej Šťava, B. Benes, Radomír Měch, D. G. Aliaga, and P. Kristof. 2010. Inverse Procedural Modeling by Automatic Generation of L-systems. *Comp. Gr. Forum (Proc. EUROGRAPHICS)* 29, 2 (2010), 665–674.

Ondřej Šťava, Sören Pirk, Julian Kratt, Baoquan Chen, Radomir Mech, Oliver Deussen, and Bedrich Benes. 2014. Inverse Procedural Modelling of Trees. *Comp. Gr. Forum* 33, 6 (2014), 118–131.

Emily Whiting, John Ochsendorf, and Frédo Durand. 2009. Procedural Modeling of Structurally-Sound Masonry Buildings. *ACM Trans. on Graphics* 28, 5 (2009), 112.

Fuzhang Wu, Dong-Ming Yan, Weiming Dong, Xiaopeng Zhang, and Peter Wonka. 2014. Inverse Procedural Modeling of Facade Layouts. *ACM Trans. Graph.* 33, 4, Article 121 (2014), 10 pages.

Gui-Song Xia, Xiang Bai, Jian Ding, Zhen Zhu, Serge Belongie, Jiebo Luo, Mihai Datcu, Marcello Pelillo, and Liangpei Zhang. 2018. DOTA: A Large-Scale Dataset for Object Detection in Aerial Images. In *IEEE CVPR*.

Jianxiong Xiao, Tian Fang, Ping Tan, Peng Zhao, Eyal Ofek, and Long Quan. 2008. Image-based facade modeling. *ACM Trans. Graph.* 27, 5, Article 161 (Dec. 2008), 10 pages.

Mehmet Ersin Yumer, Paul Asente, Radomir Mech, and Levent Burak Kara. 2015. Procedural modeling using autoencoder networks. In *UIST*. ACM, 109–118.

Hao Zhang, Kai Xu, Wei Jiang, Jinjie Lin, Daniel Cohen-Or, and Baoquan Chen. 2013. Layered analysis of irregular facades via symmetry maximization. *ACM Trans. Graph.* 32, 4, Article 121 (July 2013), 121:1–121:13 pages.

Ling Zhang and Eric Guilbert. 2016. Evaluation of river network generalization methods for preserving the drainage pattern. *ISPRS International Journal of Geo-Information* 5, 12 (2016), 230.

Huilong Zhuo, Shengchuan Zhou, Bedrich Benes, and David Whittinghill. 2015. User-Assisted Inverse Procedural Facade Modeling and Compressed Image Rendering. In *Advances in Visual Computing*. Springer International Publishing, Cham, 126–136.