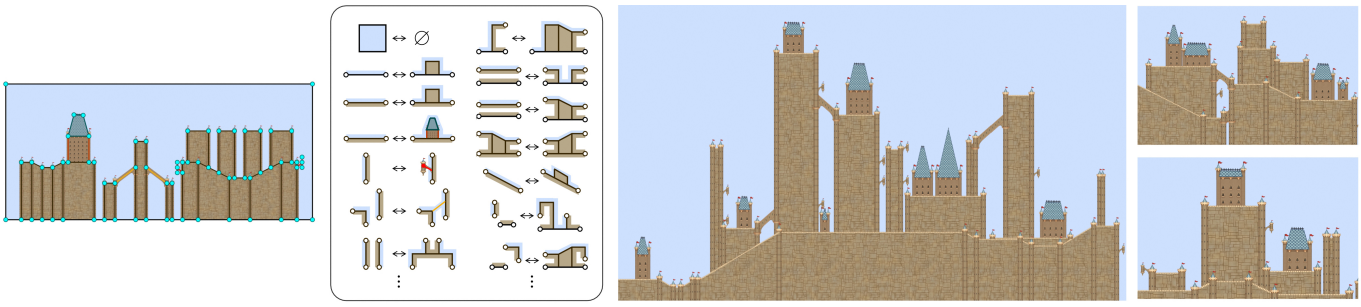
# Example-Based Procedural Modeling Using Graph Grammars

PAUL MERRELL, -, USA



(a) Example Shape (b) Graph Grammar (c) Generated Shapes

Fig. 1. From an example shape (a), our method automatically generates a graph grammar (b) that produces shapes (c) that are locally similar to the example. The graph grammar consists of rules that transform graphs. The graphs are converted into a planar graph drawing to produce the final shape.

We present a method for automatically generating polygonal shapes from an example using a graph grammar. Most procedural modeling techniques use grammars with manually created rules, but our method can create them automatically from an example. Our graph grammars generate graphs that are locally similar to a given example. We disassemble the input into small pieces called primitives and then reassemble the primitives into new graphs. We organize all possible locally similar graphs into a hierarchy and find matching graphs within the hierarchy. These matches are used to create a graph grammar that can construct every locally similar graph. Our method generates graphs using the grammar and then converts them into a planar graph drawing to produce the final shape.

CCS Concepts: • **Computing methodologies** → **Mesh geometry models**.

Additional Key Words and Phrases: inverse procedural modeling, graph grammar, local similarity

**ACM Reference Format:**
Paul Merrell. 2023. Example-Based Procedural Modeling Using Graph Grammars. *ACM Trans. Graph.* 42, 4, Article 1 (August 2023), 16 pages. https://doi.org/10.1145/3592119

## 1 INTRODUCTION

Large detailed geometric shapes are needed in many different games, animated movies, virtual worlds, and other applications. Creating these complex shapes is a challenging labor-intensive task. This remains one of the most important challenges in computer graphics.

Such shapes can be generated automatically using various grammars. Grammars are very effective in generating complex variations. But the grammar itself can be difficult to create. A grammar is made of production rules. It is often unclear what rules are needed to produce a given set of shapes. Finding the rules is a difficult process often requiring some trial and error. This process is technically challenging and resembles computer programming more than artistic design. A simpler approach would be to create grammars automatically from a set of desired shapes.

Our method begins with a set of polygonal example shapes and automatically constructs a grammar that generates similar shapes. The examples act as a guide for the new grammar, but not a strict guide. The grammar should not merely reproduce the examples. It should generalize the examples, developing novel variations from them. Our approach is to require that the output be *locally similar*. On a small local scale, each part of the output must match part of the example. But at the same time, its large-scale structure can be very different. Local similarity is used in many texture synthesis and procedural modeling techniques. In these techniques, we expect the examples to be self-similar and contain repeated parts. Otherwise, the output will be the same as the input. Our method is focused on local constraints. Large-scale constraints are also important for controlling the output, but are not the focus of this work.

In our approach, we generate a shape by first determining its connectivity and later determine its geometry. The shape's connectivity is represented by a graph with labeled edges and vertices. We first synthesize a graph and later determine its precise geometry i.e. the positions of its edges and vertices. The input shape is converted into a graph and the graph is cut into small pieces called *primitives*. Intuitively, our method synthesizes new graphs by gluing these pieces together until the pieces are fully connected. Completing the graph is relatively easy if the input graph is a tree with leaf nodes. But if the graph does not have leaves, every path must be part of a closed cycle. Closing every path can be exceptionally difficult. Existing techniques do not handle these cycles properly. They either avoid them by focusing on tree-like structures or they directly copy the input with only minor changes. Handling cycles correctly is a major contribution of our algorithm and allows it to handle arbitrary polygonal input shapes.

Author's address: Paul Merrell, paul@merrells.org, - , Redwood City, CA, USA.


0730-0301/2023/8-ART1 $15.00
https://doi.org/10.1145/3592119

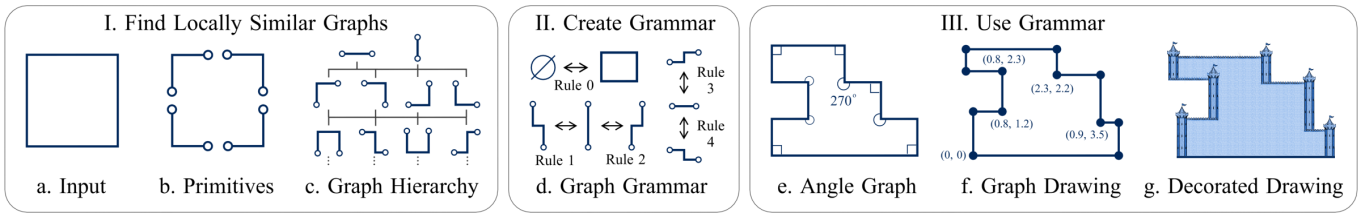



Fig. 2. Overview. The input shape (a) is cut into primitives (b). Primitives are glued into new graphs which are organized into a hierarchy (c). From this, a graph grammar (d) is constructed that generates angle graphs (e). Vertex positions are added to create a graph drawing (f) and decorations can be added (g, optional).

In approaching this problem, we first explain a way of reasoning over the abstract space of locally similar graphs. We are the first to fully describe this space. We use this reasoning to automatically construct a graph grammar that can generate all locally similar graphs. The grammar consists of rules that can modify a graph. The rules can add or remove closed loops. Each rule produces a new graph that is valid meaning that it has no paths left to be closed. This graph grammar is then integrated into a method for generating geometric shapes. The rules are applied incrementally and positions are assigned to the edges and vertices.

Our approach has several advantages over existing inverse procedural modeling techniques. It is not limited to specific shapes like trees or buildings. It can handle arbitrary polygonal shapes. Our outputs are not minor variations on the input. Our graph grammars can synthesize any locally similar shape, producing a wide range of possible large-scale structures.

## 2 RELATED WORK

A common approach to procedural modeling is to use some kind of shape grammar [Müller et al. 2006; Smelik et al. 2014; Wonka et al. 2003]. In their original formulation [Stiny 1975], shape grammars were complicated and were often manually applied with a human deciding which rules to use. Shape grammars can be simplified to set grammars [Stiny 1982; Wonka et al. 2003]. This simplification is used in most recent work on shape grammars. Another approach is to use an L-system [Lindenmayer 1968; Parish and Müller 2001; Prusinkiewicz 1986]. L-systems are a string replacement grammar. The string is turned into a model by interpreting it with a Logo-style turtle. The rules of a shape grammar or L-system or growth engine [Wong et al. 1998] are usually hand crafted by experts. Our rules are generated automatically.

A few methods do generate rules automatically from an example. This is known as *inverse procedural modeling*. Some techniques learn a split shape grammar to model building facades [Aliaga et al. 2007; Demir et al. 2016; Martinovic and Van Gool 2013; Wu et al. 2014]. Talton et al. [2012] learn a grammar from 3D models with scene graphs or web pages. Other methods learn L-systems from trees or vector art [Guo et al. 2020; Stava et al. 2010, 2014]. These techniques are only targeted to particular types of models and tree-like structures are relatively easy to handle.

In terms of goals, our work is most similar to the work of Bokeloh et al. [2010] and Liu et al. [2015]. Their methods generate a shape grammar from an input shape. They search for partial symmetries and use them to create a grammar that produces locally similar shapes. These methods work well for some, but not all, input shapes. They have trouble with cycles and cannot generate every locally similar shape for even simple shapes like rectangles (Sec. 9.1).

Our method uses a different type of grammar called a graph grammar. Graph grammars are a powerful framework introduced decades ago [Ehrig et al. 1973; Pfaltz and Rosenfeld 1969]. They are used in many applications [Rozenberg 1997] ranging from compiler design, pattern recognition, concurrent systems, database design, mesh subdivision [Smith et al. 2004; Velho 2003], and robot design [Zhao et al. 2020]. But they are only rarely used for procedural modeling [Christiansen and Bærentzen 2012; Pogrzebacz and Ilčík 2019]. Again these methods use hand crafted rules that are not generated automatically. Fiser et al. [2016] learn a graph grammar from a road network. They use this for data compression with the output graph matching the input unless the grammar is edited.

Several methods take an existing shape grammar and direct it towards a particular goal. Given a grammar, Talton et al. [2011] use an MCMC method to optimize for a set of desired properties. Dang et al. [2015] invited users to rate which generated shapes they prefer and then their method directs a grammar towards these preferences. Lipp et al. [2008] provide a framework for interactively editing shape grammars.

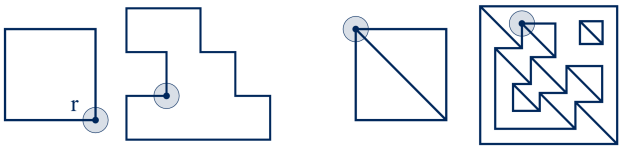
There is a long tradition of requiring local similarity when doing texture synthesis [Barnes et al. 2009; Cross and Jain 1983; Efros and Leung 1999]. Similar approaches are used to generate locally similar element arrangements [Ijiri et al. 2008], element textures [Ma et al. 2013, 2011], solid textures [Kopf et al. 2007], and curved shapes [Hertzmann et al. 2002; Merrell and Manocha 2010; Tu et al. 2020]. Kalojanov et al. [2012] provide insights in the space of locally similar shapes. These methods are very effective in their particular domain, but geometric shapes used in procedural modeling are structured quite differently. Wu and Zheng [2022] generate 3D shapes from a single example using a GAN. Their method can robustly handle noisy input data, but it tends to overfit to the training data.

Grid-based techniques can generate locally similar 3D shapes from an example [Gumin 2016; Merrell 2007; Merrell and Manocha 2008; Yeh et al. 2013]. These techniques have been used in video games like *Bad North* and *Townscaper*. Early work [Merrell 2007] used constraint solving on a set of regular tiles. This was expanded upon to operate on a set of parallel planes [Merrell and Manocha 2008, 2011], to use factor graphs [Yeh et al. 2013] and overlapping tiles [Gumin 2016]. However these methods rely on a grid as an integral part of their technique. The grid limits the shapes they can produce. Overcoming this has been a long-standing problem.

# 3 METHOD

## 3.1 Goal: Local Similarity

Our goal is to generate an output shape that resembles an input shape. The input and output must be *locally similar*. This means that every small region within the output must match a small region in the input. This idea can be described more formally using *r-similarity* [Bokeloh et al. 2010]. Two shapes are *r-similar* if for every neighborhood of radius $r$ in one shape, a translated copy of the neighborhood appears in the other shape:




We take r-similarity to its logical extreme and make each neighborhood as small as possible. The radius $r$ can be infinitesimally small, so that each neighborhood contains an edge, vertex, or face. A shape remains r-similar if its edges are shortened or extended to any length, but the edge angles must remain the same.

## 3.2 Overview

Figure 2 gives an overview of our method, which consists of three main parts. First, our method finds the set of locally similar graphs and organizes them into a hierarchy (Fig 2a-c). Second, it constructs a graph grammar from the hierarchy (Fig 2d). And third, it uses the graph grammar to generate locally similar shapes (Fig 2e-g).

(I) Section 4. Our method begins with a polygonal input shape (Fig. 2a) that is represented by a graph. This graph is disassembled and cut into small pieces called *primitives* (Fig. 2b). The primitives can be reassembled and glued into new graphs and the graphs are organized into a *graph hierarchy* (Fig. 2c).

(II) Section 5. The graph hierarchy is used as a tool to construct a *graph grammar* (Fig. 2d). Our method incrementally constructs the hierarchy and finds matching graphs within it. Each match provides a new production rule for our graph grammar and allows us to remove parts of the graph hierarchy. Our method continues finding matches ideally until the graph grammar has enough rules to produce every locally similar graph.

(III) Section 6. The first two parts focus on generating graphs. We hold off on determining the shape's geometry until the last part. Our graph grammars generate labeled graphs with straight edges at known angles (Fig. 2e). Graphs with this property are called *angle graphs* [Garg 1998]. Our angle graphs are missing vertex positions and edge lengths. The next step is to fill in this information through rejection sampling. An angle graph with vertex positions is called a *graph drawing* (Fig. 2f). Finally, our method can add decorations to the geometry in an optional post-processing step (Fig. 2g).

## 3.3 Notation & Representation

The input and output shape consists of vertices, edges, and faces.

*Face Labels.* The faces are labeled in the input and output. Different labels are shown in different colors. Figure 3a shows two face labels.

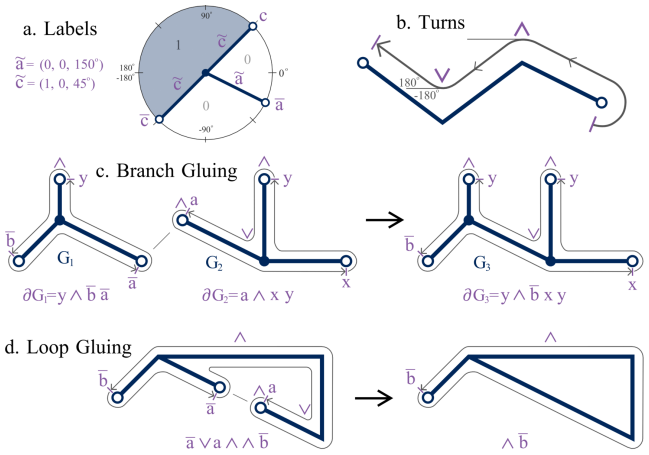



Fig. 3. (a) Face and edge labels. (b) Positive ∧ and negative ∨ turns for paths. (c,d) Graphs are assembled by gluing half-edges together. The opposite of gluing is cutting. Each graph $G$ has a boundary string $\partial G$.

*Edge Labels.* The edge labels depend on the face labels. An edge has a face on its left side and on its right. If these two faces are labeled $l$ and $r$, then the edge is labeled $\tilde{a} = (l, r, \theta)$ where $\theta$ is its tangent angle. According to this definition, edges with the same label are locally similar.

*The Cut Operation.* A cut splits an edge $\tilde{a}$ into two half-edges labeled $a$ and $\overline{a}$. Figure 3c,d shows a cut in reverse. Half-edges are illustrated as lines that end in an empty circle. While full edges are undirected, half-edges are directed. The half-edge $\overline{a}$ points in a negative direction $\overline{\theta} \in [-180°, 0°)$. While the opposite half-edge $a$ points in the opposite direction $\theta = \overline{\theta} + 180° \in [0°, 180°)$.

*The Glue Operation.* Two half-edges $a$ and $\overline{a}$ can be glued together to form one full edge. Figure 3c,d shows two examples. Cutting and gluing are exact opposites.

*Planarity.* A drawing of a graph is planar if its edges do not intersect. Planarity depends on how the tangent angle changes as we follow a path around the graph. We can determine the tangent angle of each half-edge from its label $a$ or $\overline{a}$. But the angles wrap, so that the labels alone do not tell us if the path has turned $\theta$ or $\theta + 360°$ or $\theta + 720°$ or more. To keep track of the angles wrapping, we define positive and negative turns:

*Positive Turn* ∧. If a path is turning counter-clockwise, its tangent angle $\theta$ is increasing until it reaches 180°. At that point it wraps to −180°. We call this wrapping a positive turn ∧. We use the symbol ∧ since the path makes a ∧ shape. (See Fig. 3b).

*Negative Turn* ∨. If a path is turning clockwise, its tangent angle $\theta$ wraps in the opposite direction from −180° to 180°. We call this a negative turn ∨. The path makes a ∨ shape. We sometimes use exponents to describe repeated turns: $\wedge^2 = \wedge\wedge$ and $\wedge^{-2} = \vee\vee$.

*The Graph Boundary String.* We introduce a new compact notation that fully describes the possible gluing operations assuming planarity. Each graph $G$ has a boundary string $\partial G$. The string $\partial G$ contains every half-edge and every turn in $G$. For example, $\partial G_1 = y \wedge \overline{ba}$ means the graph $G_1$ has three half-edges: $y\overline{ba}$ and one turn: ∧ and they appear in the order $y \wedge \overline{ba}$ as we follow a path

counter-clockwise around $G_1$ (Fig. 3cd). This path forms a circular loop around the graph. It is unclear where the path should start. The loop is the same no matter where it started. We treat different starting points as being equivalent by defining shifted strings to be equal: $y\wedge\overline{ba} = \wedge\overline{ba}y = \overline{ba}y\wedge$. Consecutive positive and negative turns cancel: $a\wedge x\wedge\vee = a\wedge x$. Let $P_G$ and $N_G$ be the number of positive and negatives turns for any graph $G$. Then $P_G - N_G = 1$ since the path loops once counter-clockwise.

## 4 FINDING LOCALLY SIMILAR GRAPHS

### 4.1 Disassembly: Cutting Into Primitives

Our method begins by cutting the input shape into small pieces. The input shape consists of vertices, edges, and faces (Fig. 2a). We can convert the input shape $S$ into a graph $G_S$ without losing any essential information because it is contained in the edge labels. Our method then disassembles the input graph $G_S$ by cutting it into as many pieces as possible (Fig. 2b & 4). Every edge is cut into two half-edges. Afterwards, the vertices of $G_S$ are each disconnected from the other vertices and are surrounded by half-edges. Each disconnected piece is a new graph. We call these graphs *primitives* since they are the most basic building blocks of our method. Any graph $H$ that is assembled from these primitives will be locally similar to $G_S$.

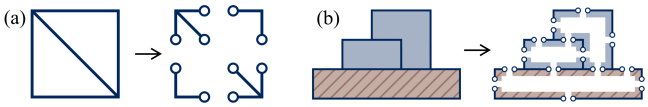



Fig. 4. (a) A diagonal box and (b) a platform shape are cut into primitives.

### 4.2 Assembly: Gluing Primitives Together

Primitives are glued together at their half-edges. Half-edges can be glued in two different ways which we call *branch gluing* and *loop gluing*. Branch gluing means the two half-edges are on disconnected graphs with no path between them (Fig. 3c). Loop gluing means the half-edges are on a connected graph (Fig. 3d). If such half-edges are glued together, they form a loop. Loop gluing and branch gluing change the boundary string according to a simple string replacement. This is explained more below, but can be summarized as:

| | |
|---|---|
| Loop Glue: $a\overline{a} \rightarrow \epsilon$ | $\overline{a}\vee a\wedge \rightarrow \epsilon$ |
| Branch Glue $\overline{a}B$ to $a$: $a \rightarrow B\vee$ | $aB$ to $\overline{a}$: $\overline{a} \rightarrow \vee B$ |

where $\epsilon$ is the empty string and uppercase letters like $B$ represent arbitrary strings. The gluing operations define a context-sensitive grammar that acts on the boundary string. The loop gluing rules are context-sensitive, while the branch gluing rules are context-free.

Loop gluing creates closed loops. In a planar graph, closed loops must turn 360°. So the half-edges $\overline{a}$ and $a$ can only be loop glued if the path between them turns ±360°. When the path turns +360°, the boundary string contains the substring $a\overline{a}$ for some label $a$. When the path turns −360°, it contains the substring $\overline{a}\vee a\wedge$. Gluing the half-edges removes these substrings: $a\overline{a} \rightarrow \epsilon$ and $\overline{a}\vee a\wedge \rightarrow \epsilon$.

In branch gluing, two graphs $G_1$ and $G_2$ are glued together. If we glue the graphs at the half-edges $\overline{a}$ and $a$, then the two graph boundaries $B_1\overline{a}$ and $aB_2$ are combined into $B_1\vee B_2$. For example, in Figure 3c, the boundaries $\partial G_1 = y\wedge\overline{ba}$ and $\partial G_2 = a\wedge xy$ are glued together to form $\partial G_3 = y\wedge\overline{b}xy$. This can be described as a string replacement in two different ways that are equivalent. If we glue the graph $\overline{a}B$ to the half-edge $a$, this replaces $a \rightarrow B\vee$. If we glue the graph $aB$ to the half-edge $\overline{a}$, this replaces $\overline{a} \rightarrow \vee B$.

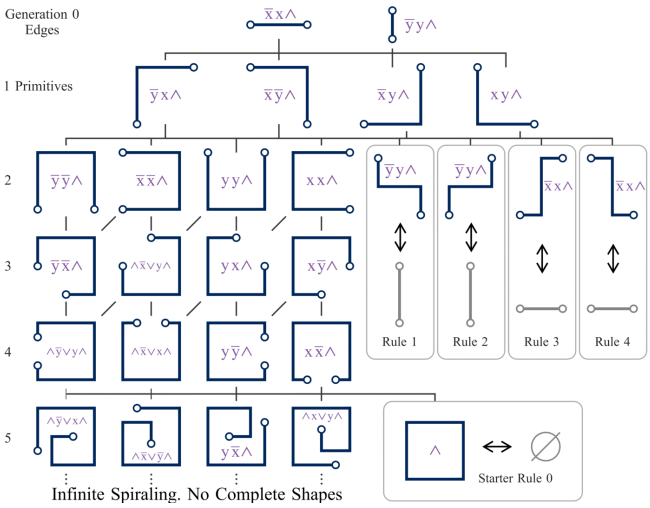
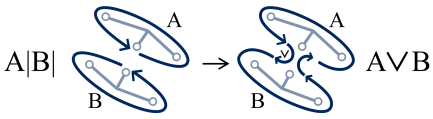



Fig. 5. A graph hierarchy starts simple and grows more complex as primitives and loops are glued. From the hierarchy, a graph grammar with five rules is found by matching boundary strings.

Branch gluing could alternatively be described as looping gluing combined with a splice operation. We use the symbol | for splicing. The boundary strings $A$ and $B$ represent closed paths. $A|B|$ means that the end of path $A$ is attached to the start of $B$ and the end of $B$ is attached to the start of $A$. The result of a splice is $A|B| \rightarrow A\vee B$. The strings are concatenated with a turn $\vee$ added in between:




The extra turn $\vee$ is necessary for the boundary to turn once counter-clockwise: $P_{A\vee B} - N_{A\vee B} = P_A - N_A + (-1) + P_B - N_B = 1$. Branch gluing $B\overline{a}$ and $aC$ is equivalent to a splice $B\overline{a}|aC| \rightarrow B\overline{a}\vee aC$ followed by loop gluing $B\overline{a}\vee a\wedge\vee C \rightarrow B\vee C$.

### 4.3 The Graph Hierarchy

We now explain how to enumerate every possible way of gluing primitives into graphs. Properly defining this space is an important contribution of our work. Each possible graph is placed inside of a hierarchy (Fig. 5). The hierarchy is divided into generations. Generation $i$ consists of every graph that can be constructed using $i$ gluing operations. If the hierarchy were continued forever to infinitely many generations, it would contain every possible way of gluing the primitives together. It would contain every locally similar graph. Such a hierarchy cannot be implemented since it would be infinitely

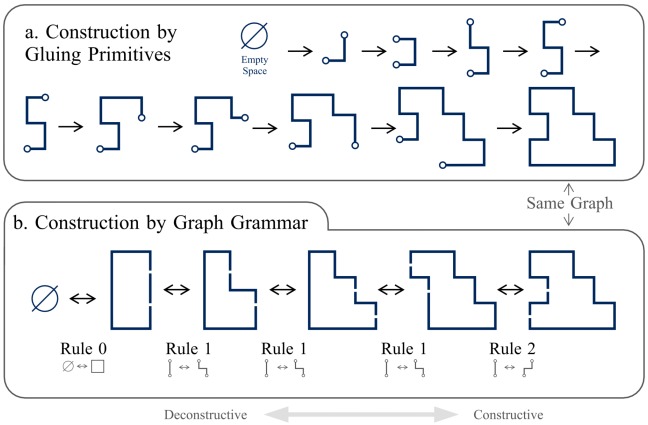
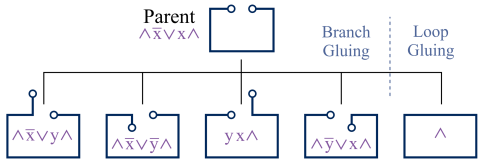



Fig. 6. One way of constructing graphs (a) is to randomly glue primitives together until the graph is complete. A better way is to use a graph grammar (b). Fig. 5 shows how this graph grammar is automatically generated.

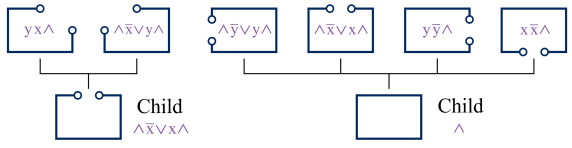
large. Thus this hierarchy should not be thought of as a concrete data structure, but rather as an abstract model. We call this the *abstract* graph hierarchy. In Section 5, we will implement a *concrete* version of this hierarchy as a data structure. But we will limit that concrete hierarchy to be a finite subset of the full abstract hierarchy.

Figure 5 shows a graph hierarchy. It will be used as a guiding example through this section. The hierarchy is similar to a tree. Each graph has parents, children, and descendants. But the hierarchy is actually not a tree since two siblings can share a child.

The simplest graphs are at the top of the hierarchy. As we descend, the graphs grow more complex. Generation 0 contains the edge graphs. Generation 1 contains the primitives. Each graph has a set of children. To find a graph's children, perform all possible loop gluing operations and branch gluing operations with the primitives:




A child can have multiple parents. Each of its parents is a copy of the child with one primitive cut out or one loop cut:




## 4.4 Complete Graphs

A graph is *complete* if it has no remaining half-edges left to glue. A complete graph $G$ always has the boundary string $\partial G = \wedge$. Otherwise, the graph is incomplete.

Our goal is to generate complete graphs. One possible approach is to do a random walk through the graph hierarchy. Basically, we would randomly glue primitives together until reaching a complete graph by chance. Figure 6a illustrates the idea. This simple approach works fine for some input graphs including graphs without cycles. But for many other inputs, this is an error-prone and inefficient way to construct graphs. We might start constructing a graph and then be unable to complete it. Consider the two incomplete graphs below. The left graph is assembled from the diagonal primitives (Fig. 4a). The right graph is assembled from the platform primitives (Fig. 4b):

The left graph is difficult to complete. The right graph is impossible to complete. We consider a different strategy in Section 5 that can generate every complete graph, but without the wrong turns and dead ends that might be encountered along a random walk.

# 5 GRAPH GRAMMARS

In this section, we explain how to use the graph hierarchy to construct a graph grammar that can produce all complete, locally similar graphs. First, we give some background on graph grammars. Graph grammars generalize the concept of a formal grammar based on strings into graphs [Rozenberg 1997]. There are many different approaches to graph grammars. We use the gluing approach, also known as the algebraic approach [Ehrig et al. 1973]. Within that approach, we use *double-pushout* graph grammars (DPO). The term pushout is borrowed from category theory. Graph grammars are often described using category theory. But we prefer to center our discussion around graph gluing, following König et al. [2018].

*Morphism*: A graph homomorphism (or morphism for short) maps between two graphs $G \rightarrow H$. It consists of two mappings. One from $G$'s vertices to $H$'s vertices. And one from $G$'s edges to $H$'s edges. The mappings should respect edge and vertex labels.

Our grammars consist of a set of *DPO production rules*. A DPO rule contains a left graph $L$ and a right graph $R$. The right graph replaces the left. In addition to specifying the graphs $L$ and $R$, we must define the relationship between them. This is done through an interface graph $I$ and two morphisms: $\varphi_L : I \rightarrow L$ and $\varphi_R : I \rightarrow R$ (Fig. 7).

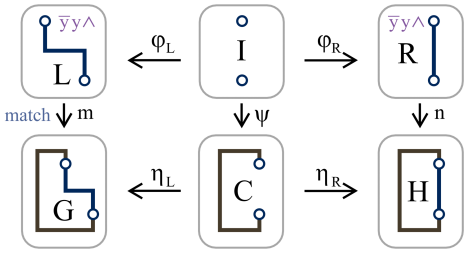



Fig. 7. A double-pushout production rule. The left side $L$ is matched with $G$. $L$ is cut from $G$ and $R$ is glued in its place to make $H$.

In our method, graphs $L$ and $R$ always have the same graph boundary string ($\partial L = \partial R = \bar{y}y\wedge$ in Fig. 7). This is necessary so

**Algorithm 1** Find Graph Grammar From Primitives

1: Starting from primitives, glue graphs together to build a concrete hierarchy ideally until every complete shape is reducible.
2: **for** each graph $G$ **do**
3: If possible, use $G$ in a rule on the left or right (Sec. 5.3-5.5).
4: **if** $G$'s has no complete descendants **then**
5: Remove $G$ and all its descendants (Sec. 5.6).

that $R$ can replace $L$ without breaking planarity. Graphs $L$ and $R$ have the same half-edges. Each half-edge ends in a vertex shown as an empty circle. The interface graph $I$ consists of those vertices and the morphisms $\varphi_L$ and $\varphi_R$ map between them based on their matching half-edges. In most figures, we omit the interface $I$ since in our method $I$ is fully determined from the half-edges of $L$ and $R$.

To apply a DPO rule to graph $G$, we find a subgraph of $G$ that matches $L$. This is described by a morphism $m$ called the *match* where $m : L \to G$. The subgraph of $G$ that is matched to $L$ is cut out, and $R$ is glued in its place. Cutting $L$ from $G$ produces the context graph $C$. Gluing $R$ to $C$ produces the final graph $H$. To summarize, we match a part of $G$ to $L$ and replace it with $R$ to produce $H$.

Another perspective is that Figure 7 consists of two graph gluing operations. On the left side, $L$ is glued to $C$ to make $G$. On the right side $R$ is glued to $C$ to make $H$. According to category theory, these two gluing operations are pushouts and this is a double pushout.

## 5.1 Outline

Algorithm 1 summarizes our approach. The input to the algorithm is a set of primitives. The output is a set of rules that make up a graph grammar. We begin by building a graph hierarchy incrementally starting from the primitives, using the same branch gluing and loop gluing operations described in Section 4.3. But this hierarchy is smaller. Before each graph $G$ is added to the hierarchy, we check if it is possible to create a rule that will simplify $G$. (We explain how to create the rules in Sec. 5.3). If a graph $G$ can be simplified by a rule, we say that $G$ is *reducible* and we remove it from the hierarchy. Each time we add a graph $G$, we check if $G$ can be simplified or if $G$ can be used to simplify another graph.

*Graph Complexity.* We define a way of ordering the graphs from simple to complex. Graphs with fewer half-edges are simpler than those with more. If two graphs have an equal number of half-edges, then the graphs are ordered according to the hierarchy. Graphs added earlier in the hierarchy are simpler than those added later.

We use this to order the graphs in each rule. The right side $R$ of a rule is always simpler than the left $L$. A graph is reducible if it is on the left side of a rule. A rule can split a graph $L$ into multiple simpler graphs on the right $R$.

If a graph grammar contains enough rules to reduce every complete graph in the hierarchy, then it can fully construct every locally similar graph. We explain the theory behind this and how we can make such a claim in Section 5.2. We explain how to find the production rules in Section 5.3. We then discuss a complication that can happen if a graph is reduced by one of its descendants in Sections 5.4 - 5.5. It is possible that a graph has an infinite number of irreducible descendants that are all incomplete. Our method detects these graphs and removes them and their descendants (see Section 5.6). Ideally, we continue adding rules to the hierarchy until we are certain that every complete graph is reducible. But in some cases, the number of graphs in each generation grows faster than they can be removed. Then the algorithm exits without a guarantee that all graphs are reducible (see Section 5.7).

## 5.2 How this Algorithm Works

An important property of DPO graph grammars is that all production rules are invertible [Ehrig 1979]. Each rule can transform a left graph into a right graph: $L \to R$, or it can be reversed to transform a right graph into a left graph: $R \to L$.

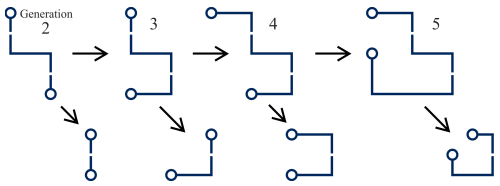
By convention, we put the simpler graph on the right $R$, so that applying a rule $L \to R$ will simplify any graph that it is applied to $G \to H$. We say that applying a rule $L \to R$ is *destructive* because it deconstructs the graph into simpler parts. Applying it $R \to L$ is *constructive* because it constructs a more complex graph.

Figure 6b shows rules being applied constructively i.e. a complex graph is constructed from a simple one. The simplest graph is the empty graph $\emptyset$. If a complex graph can be deconstructed from a simple one, the reverse is also true. By reversing each step in Figure 6b, a complex graph can be deconstructed into the empty graph.

A graph can always be reduced to a set of irreducible graphs. If a graph is reducible, it can be reduced to another graph. If that graph is reducible, it can be reduced to an even simpler graph. This is a proof by infinite descent. The graphs can continue to be reduced ad infinitum until an irreducible graph is reached.

Our algorithm continues adding production rules ideally until every complete graph is reducible. At that point, the only irreducible graph left is the empty graph $\emptyset$. Every complete graph can be reduced to $\emptyset$. And since every destructive action can be reversed, the reverse is also true. Every complete graph can be constructed from the empty graph $\emptyset$. At that point, our graph grammar can construct every complete, locally similar graph and Algorithm 1 terminates.

*5.2.1 Reducing All Descendants.* If a rule can be applied to $L$, it can be applied to all of $L$'s descendants because its descendants contain $L$ as a subgraph. Remember that $L$'s descendants are found by gluing primitives to $L$. Ordinarily, if $L$ can be reduced by some rule, then all of $L$'s descendants can be reduced by the same rule. (There is an important exception to this statement discussed in Section 5.4). Below we show an example. A graph $L$ can be reduced by Rule 1 in Figure 5 and so can all of its descendants.




## 5.3 Finding a Rule to Reduce a Graph

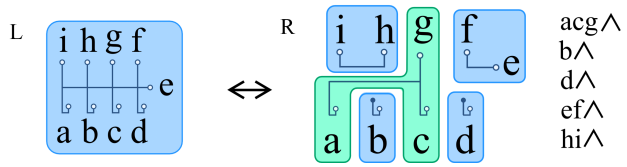
*Boundary Strings.* Our method finds production rules by matching boundary strings. A graph $R$ can replace a graph $L$ if both graphs have the same boundary string $\partial R = \partial L$. In Figure 7, $\partial R = \partial L = \overline{y}y\wedge$.

Similarly, the five rules in Figure 5, have the same boundary strings on the left and right sides.

When two graphs $L$ and $R$ have the same boundary strings, they can seamlessly replace one another. Their half-edges are the same, so each of $L$'s half-edges can be cut out and replaced by one of $R$'s half-edges. They also have the same turns $\wedge$ and $\vee$ so the paths between the half-edges turn the same. They have the same total curvature. This is necessary to preserve planarity.

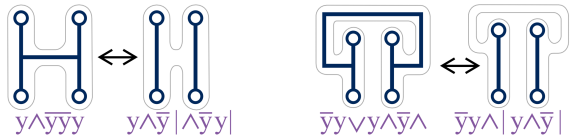
We find rules by string matching. For a given boundary string $\partial L$, we can match $L$ to a graph $R$ with the same boundary $\partial R = \partial L$. But we could also match $L$ to a set of graphs $\{R_1, R_2, \ldots\}$. This set matches $L$ if the strings $\{\partial R_1, \partial R_2, \ldots\}$ can be combined to equal $\partial L$.

Our approach is outlined in pseudocode in Algorithm 2. This is a recursive divide-and-conquer algorithm. For example, suppose that graph $L$ has the boundary $\partial L = abcdefghi\wedge$:




Our algorithm tries to match $L$ using every graph $R$ in the hierarchy. It tries to match each $\partial R$ to some part of $\partial L$. Suppose that $\partial R = acg\wedge$. After matching $\partial R$ to $\partial L$, we are left with three unmatched substrings: $b$, $def$, and $hi$. The algorithm runs recursively on each substring. One solution might be to match $\partial L = abcdefghi\wedge$ to $acg\wedge$, $b\wedge$, $d\wedge$, $ef\wedge$, and $hi\wedge$ assuming those graphs are in the hierarchy.

The way we combine the boundary strings $\{\partial R_1, \partial R_2, \ldots\}$ requires some additional explanation. The strings are combined by splicing them together. When we splice two boundary strings an extra turn $\vee$ is added between them: $\partial R_1|\partial R_2| \rightarrow \partial R_1 \vee \partial R_2$. In fact, there are multiple ways to splice two strings together. Below are two examples where the same right graphs $R_1$ and $R_2$ are spliced together in different ways to produce different left graphs: $y\wedge\overline{y}\overline{y}y$ and $\overline{y}y\vee y\wedge\overline{y}\wedge$




When we splice together two strings, we are free to add extra turns between them. The splice operation can be written more generally as $A|B| \rightarrow A\wedge^n B\vee^{n+1}$ for some $n \in \mathbb{Z}$.

Suppose we have matched $\partial R$ to part of $\partial L$. And let $\partial L_i$ be the unmatched part so that $\partial L = \partial R \partial L_i$. We wish to splice some string to $\partial R$ to get $\partial L$. This formula will give us the desired result: $\partial R|\vee^n \partial L_i \wedge^{n+1}| \rightarrow \partial R\partial L_i = \partial L$. Consequently on line 2 of Algorithm 2, we match $\partial R$ with $\vee^n \partial L \wedge^n$ and on line 5 we match with $\partial L_i\wedge$. Notice that in the example above the unmatched substrings were: $b$, $def$, and $hi$ and they were matched to $b\wedge$, $def\wedge$, and $hi\wedge$. The extra $\wedge$ cancels with the $\vee$ that is added during splicing.

**Algorithm 2** findMatchingGroups($\partial L$): $\partial L$ is a boundary string

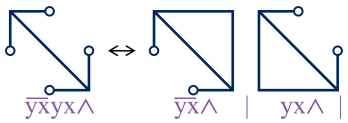
```
1: for each graph R with boundary ∂R in the hierarchy do
2:   for each way of matching ∂R with ∨^n ∂L ∧^n for some n do
3:     matches = [R]
4:     for each unmatched substring ∂L_i do
5:       matches.push(findMatchingGroups(∂L_i ∧));
6:     if all matches successful return matches;
7: return null
```

*5.3.1 Special Case: Complete Graphs.* A complete graph $G$ has no half-edges and has the boundary string $\partial G = \wedge$. For every complete graph, we can define a rule that deletes the graph like Rule 0 in Figure 5. We call these *starter rules*.


Like other grammars, graph grammars have an axiom or a start graph that they begin with and then rules are applied to derive new graphs. In our method, the start graph is an empty graph $\emptyset$. Initially, the starter rules are the only rules that can be applied.

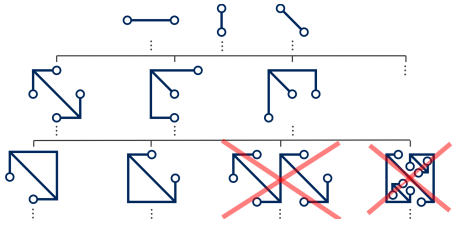
## 5.4 Reducing a Graph with Its Descendants

Suppose that a rule simplifies the graph $L$ into the graphs $R_1$ and $R_2$. Ordinarily, that would mean that $L$ and all its descendants are reducible (Sec. 5.2.1). But what if $R_1$ is a descendant of $L$? This rule does not reduce every descendant of $L$ since applying the rule to $R_1$ replaces $R_1$ with itself. This does not simplify $R_1$. For example, consider this rule from the Diagonal example (Fig. 4a):




This rule reduces the left graph $\partial L = \overline{yx}yx\wedge$ into two simpler right graphs $\partial R_1 = \overline{yx}\wedge$ and $\partial R_2 = yx\wedge$. $R_1$ and $R_2$ are simpler than $L$ since they have two half-edges and $L$ has four. $R_1$ and $R_2$ contain $L$ as a subgraph, so they are descendants of $L$.

This rule is still useful. It can reduce all descendants of $L$ besides those that are descendants of $R_1$ and $R_2$. Our method then restructures the graph hierarchy. It removes $L$ and its descendants from the hierarchy and replaces them with $R_1$ and $R_2$ and their descendants. Every descendant of $L$ can be turned into a descendant of $R_1$ and $R_2$ using the above rule:



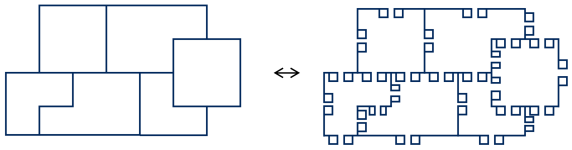
This scenario only happens occasionally. In most of our results, our algorithm can finish without using any rules that contain descendants of $L$ on the right side of the rule. Often this technique reduces the number of rules, but the algorithm can finish without it.

### 5.5 One-half-edge Graphs, Stubs

If a graph has one half-edge, its boundary string is $a\wedge$ or $\overline{a}\wedge$ for some label $a$. We call these graphs *stubs*. If the stub $a\wedge$ is glued to half-edge $\overline{a}$, then $\overline{a} \rightarrow \wedge\vee = \epsilon$ according to Section 4.2. This property is very useful. Stubs often exist. If the input graph has no cycles, then a pair of stubs $a\wedge$ and $\overline{a}\wedge$ exists for every label $a$. And even when the input graph has only cycles, stubs often exist. Our algorithm finds any stubs that can be created since our graph hierarchy checks every locally similar graph. Our algorithm finds stubs that are not in the primitives and are not part of the input graph.

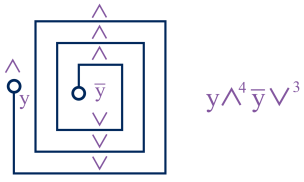
Stubs are very useful for deconstructing graphs. If the stubs $a\wedge$ and $\overline{a}\wedge$ exist, every primitive and every edge that has the label $a$ can be deconstructed. For example, below left is an input graph. The input contains no stubs. But when we build the graph hierarchy we find stubs that can deconstruct all the primitives.


The approach outlined in Section 5.4 applies here since the stub $x\wedge$ is a descendant of the primitive $\overline{y}x\wedge$ and it can also deconstruct $\overline{y}x\wedge$. Every primitive and every graph can be deconstructed into the four stubs: $x\wedge$, $\overline{x}\wedge$, $y\wedge$, and $\overline{y}\wedge$. Below a shape is deconstructed by applying the rules $\overline{x}x\wedge \rightarrow \overline{x}\wedge|x\wedge|$ and $\overline{y}y\wedge \rightarrow \overline{y}\wedge|y\wedge|$. The remaining graphs can be deconstructed by 8 starter rules. In this case, the stubs alone solve the problem.



### 5.6 No Complete Irreducible Descendants

The graph hierarchy may contain graphs that cannot be reduced. For example, if Figure 5 were continued several more generations, we would see the graph $y\wedge^4\overline{y}\vee^3$ inside the hierarchy:




Not only is $y\wedge^4\overline{y}\vee^3$ irreducible, it is part of an infinite set of irreducible graphs of the form $y\wedge^{n+1}\overline{y}\vee^n$ for some $n$. The hierarchy of Figure 5 contains no stubs. Each graph can only be completed by forming a path that turns 360° and then loop gluing it. The path from $y$ to $\overline{y}$ has 4 counter-clockwise turns ($\wedge^4$). The only way to form it into a loop is for it to turn clockwise. But if anything is glued to the path to turn it clockwise, the resulting graph can be reduced by one of Figure 5's rules. Therefore every descendant of $y\wedge^4\overline{y}\vee^3$ is reducible or incomplete.

We need to detect and remove graphs with only reducible or incomplete descendants. Otherwise, we cannot be certain that our grammar can reduce every complete graph. We work through a more complex example in Appendix C. We consider all the options for a graph. We show that the half-edges always end up in a replacement cycle and that the graph cannot be completed.

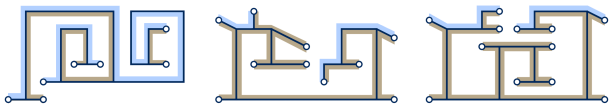
For every boundary string, there exists a complementary boundary string that could be glued to it to make it complete. For example, $abc\wedge$ can be glued to $\overline{cba}\wedge$. A graph can be completed if and only if a complementary string can be derived using the context-sensitive grammar defined in Section 4.2 and splicing.

We can guarantee that all of a graph's descendants are reducible or incomplete if: (1) The graph contains a half-edge that has turned more than 360° from the previous or next half-edge. (2) The total curvature stays above 360° through every path that descends from the graph. Considering that every sufficiently long path must end either in a stub or be part of a replacement cycle, that means (3) none of the paths ends in a stub and (4) every cycle has positive or zero curvature. If these conditions are met the graph cannot be completed and it has no complete descendants. The graph and all its descendants are removed from the hierarchy.

### 5.7 The Growth of the Graph Hierarchy

For some graph hierarchies, it can be very difficult to track down and conclusively show that every complete graph is reducible. There are simply too many ways that the graphs can spiral in on themselves.

A simple solution is to limit the number of generations or half-edges in the hierarchy or to ignore graphs that have turned more than 360°. The downside is that we can no longer be certain that our grammar can produce every locally similar graph. The remaining irreducible graphs are very complex spiraled shapes. We suspect most of them have no complete irreducible descendants, but we cannot be certain of this. Here are some examples of graphs that remain in the hierarchy for the Castle example in Figure 1:



This is more of an issue when we extend the algorithm to 3D shapes in Section 7. For 2D shapes, we only need to limit the size of the hierarchy in a few cases. And also we can see that these grammars still produce a rich variety of shapes and so as a practical matter they fulfill the desired goals.

## 6 USING THE GRAPH GRAMMAR

Now that we have a graph grammar (Fig. 2d), this section explains how to use it to generate a graph drawing (Fig. 2f). Our approach is described in pseudocode in Algorithm 3. We use the graph grammar to go on a random walk through the space of complete locally similar shapes. Algorithm 3 begins with an empty graph $\emptyset$ and changes it incrementally. At each iteration, our method proposes a change to the graph using a rule (Line 3, Sec. 6.1). Each rule specifies what

**Algorithm 3** Generate Graph Drawing From Grammar

1: Start with an empty graph $\emptyset$.
2: **for** $n$ = 1 to $N$ **do**
3: Propose a change to the graph using a rule.
4: **for** $i$ = 1 to $I$ and not accepted **do**
5: Find the space of consistent graph drawings: $\hat{\mathbf{x}} + \mathbf{K}_A\mathbf{\Lambda}$
6: **for** $j$ = 1 to $J$ and not accepted **do**
7: Sample from the space $\hat{\mathbf{x}} + \mathbf{K}_A\mathbf{\Lambda}$
8: **if** sample is acceptable **then** accept new graph drawing.
9: Free a vertex. Allow its position to change.

edges and vertices to add or remove, but it does not specify the vertex positions. By setting vertex positions, an angle graph (Fig. 2e) becomes a graph drawing (Fig. 2f). Our method proposes a set of vertex positions (Line 7, Sec. 6.2) which are then accepted or rejected (Line 8, Sec. 6.3). While this is a random process, it can be guided by being selective in which proposals are accepted.

## 6.1 Propose an Angle Graph

At each iteration, our method proposes a change to the graph using a production rule. Each DPO rule is bidirectional and can be applied constructively $R \rightarrow L$ or destructively $L \rightarrow R$. Our method applies the rules in both directions with no preference on the direction.

To apply a rule, our method matches part of the existing graph to the left $L$ or right side $R$ of a rule. The match $m$ is a morphism depicted in Figure 7 that maps from the left side $L$ to the existing graph $G$. Determining if a morphism exists between two arbitrary graphs is an NP-hard problem [Cook 1971]. But our graphs are all planar and planar graphs can be matched in linear time [Eppstein 1999]. When a match is found the left or right side is cut out and replaced by the opposite side. There is one special case. A starter rule has an empty graph on its right. Starter rules can create graphs from nothing or delete graphs to nothing.

## 6.2 Propose a Graph Drawing

The next step is to propose a planar graph drawing. Our graph grammars guarantee that all closed loops turn $\pm 360^\circ$. This is a necessary but not a sufficient condition for a planar graph drawing to exist. For a given angle graph, a planar graph drawing may not exist and deciding if it exists is NP-hard [Garg 1998]. But there are many exceptions. The problem can be solved in linear time if the graph is a tree, a simple cycle, or a series-parallel graph [Garg 1998].

Our approach is to make small incremental changes to an existing graph drawing. Ideally, we would only set the positions of vertices added by the production rule and not move other vertices, but sometimes it is necessary to move them. Our method repeatedly attempts to create a planar graph drawing (Lines 4 - 9). If these attempts fail, it tries a new rule at a new location. Because this problem is NP-hard, we cannot rule out the possibility that it may be exceptionally difficult to produce a planar drawing for some input shapes. But we have not seen this behavior in any of our experiments. Our method makes small incremental changes and exits early if any particular angle graph is difficult to solve.

Our eventual goal is to produce a graph drawing that satisfies all requirements listed below in Section 6.3. But first we need to find the space of possible solutions. Following Bokeloh et al. [2012], we will find the nullspace of a linear system.

The angle of each edge must agree with its label. Each edge label $\tilde{a} = (l, r, \theta)$ specifies an angle $\theta$. Let $\mathbf{u} = [\cos\theta, \sin\theta]$ be the direction of an edge that goes between two vertices located at $\mathbf{v}_0$ and $\mathbf{v}_1$, then $\mathbf{v}_1 = \mathbf{v}_0 + s\mathbf{u}$ for some edge length $s$. This is the equation for one edge. The equations for all the edges can be combined into one matrix equation $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{x}$ is a column vector of the vertex positions and edge lengths.

The equation $\mathbf{Ax} = \mathbf{b}$ may have many solutions or it may have none. If matrix $\mathbf{A}$ has a nullspace, let $\mathbf{K}_A$ be a basis for this nullspace. If $\hat{\mathbf{x}}$ is a solution, then $\hat{\mathbf{x}} + \mathbf{K}_A\mathbf{\Lambda}$ is also a solution for any $\mathbf{\Lambda}$. We propose solutions by sampling from the solution space $\hat{\mathbf{x}} + \mathbf{K}_A\mathbf{\Lambda}$.

On the other hand, $\mathbf{Ax} = \mathbf{b}$ may not have a solution. The vertex positions may be overconstrained. This is the potential problem with our initial strategy of moving as few vertices as possible. In this case, it is necessary to move more vertices. We use an iterative approach. At each iteration $i$, there are a set of free vertices whose position can change. While the vertices are overconstrained, randomly pick a non-free vertex that is adjacent to a free vertex and turn it into a free vertex (Line 9). Its position is now a free variable in our equations and so are the lengths of any adjacent edges. We update $\mathbf{A}$ and $\mathbf{b}$ accordingly. This gives us additional degrees of freedom which can turn an overconstrained problem into a solvable problem.

## 6.3 Accept or Reject Proposal

The graph drawing must satisfy four requirements: (1) the edges must have the correct angle, (2) the edge lengths must be positive and fit within a specified range, (3) the drawing must be planar i.e. the edges must not intersect, and (4) the drawing must not be rejected by the Metropolis algorithm. (4) is optional.

For requirement (2), the user can specify length requirements. Alternatively, we could infer this from training data. The user can specify minimum and maximum edge lengths. The user could strictly set the edge length to one particular value. Or they can require it to be an integer multiple of some length. This is useful for tiled patterns like windows or stairs.

Requirement (1) is satisfied by any solution of the form $\hat{\mathbf{x}} + \mathbf{K}_A\mathbf{\Lambda}$. Together requirements (1) and (2) have the same form as a linear programming problem consisting of a set of linear constraints and inequalities. But unlike linear programming, this is a sampling problem, not an optimization. We are sampling from the space of locally similar shapes, not optimizing for any particular edge lengths. We use rejection sampling. We sample solutions of the form $\hat{\mathbf{x}} + \mathbf{K}_A\mathbf{\Lambda}$ and reject samples that do not satisfy the requirements. If $J$ samples are rejected, we move onto a new proposal.

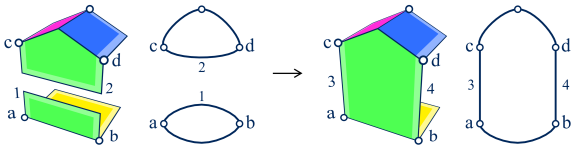
Requirement (4) is an optional way of optimizing the graph drawing using a cost function. Our method can accept or reject the new drawing based on a probability determined by the Metropolis-Hastings Algorithm [Metropolis et al. 1953; Talton et al. 2011]. The cost function is arbitrary. We can optimize for any property. We can specify the desired density of the geometry or a particular edge label we wish to see more or less frequently.

## 7 EXTENSION TO 3D SHAPES

Up to this point our discussion has focused on 2D shapes. But our method can be extended to generate 3D shapes with a few changes. In 3D, the input and output shapes consist of 3D vertices, edges, faces, and volumes. This can be represented as a graph with the edges labeled based on the adjacent faces and volumes. In the 2D case, the boundary $\partial G$ was a 1D path around a 2D graph $G$. But in the 3D case, the boundary $\partial G$ is a 2D surface. In the 2D case, the boundary $\partial G$ could be represented by a 1D string because it was 1D. But in the 3D case, the boundary $\partial G$ is a 2D graph. When we go from the graph $G$ to the boundary $\partial G$, we essentially lose one dimension. A 3D graph $G$ has a 2D boundary graph $\partial G$. The 2D faces of $G$ intersect $\partial G$ along 1D edges. The 1D edges of $G$ intersect $\partial G$ at 0D points. These are the points we have been illustrating as empty circles at the end of each half-edge.

The algorithm is very similar in 2D and 3D. The input shape is disassembled into primitives. The primitives are glued together to form a graph hierarchy. Loop gluing is possible when two half-edges sharing a common face turn $\pm 360°$. We define a coordinate system for each face from which we can compute tangent angles $\theta$ and positive $\wedge$ and negative $\vee$ turns. In 2D, we find graph grammars by matching boundary strings. In 3D, the boundary is a graph, so we find graph isomorphisms instead.

We generalize the splice operation $A|B|$ for the 3D case. Splicing is now a graph operation where we swap the connections between two edges. Below is an example of this operation. It merges two separate graphs into one:




The two graphs are joined along a common face. Faces are edges in the boundary. Edge 1 goes from vertex $a$ to $b$. Edge 2 from $c$ to $d$. The splice operation replaces these edges. Edge 3 goes from $a$ to $c$. Edge 4 from $b$ to $d$. This is the same splice operation we have been using. It only requires a shift in perspective thinking of the boundary $\partial G$ as a graph rather than a string.

## 8 RESULTS

Figures 1, 8, and 9 show shapes that are automatically generated from an example. The inputs are an example shape, information about the expected edge lengths (Sec. 6.3) and optionally instructions for decorating the vertices, edges, and faces (Appendix B). From the example shape, our algorithm automatically generates primitives, then a graph grammar, and then a decorated graph drawing. Algorithm 3 contains a few parameters. We set $I = 5$ and $J = 10$ for all the results. We vary the number of iterations $N$ as needed based on the size of the output and the amount of geometric detail desired. Table 1 shows various statistics for the results in Figure 9. Some of the results require some extra handling to include a ground plane or to be fully connected (see Appendix A).

The graph grammars generated for Figure 8 are relatively simple and easier to understand. The graph grammars generated for Figure 9 are much more complicated. Figure 8 demonstrates a number of the techniques we describe including starter rules, stubs, and grammars with no loops.

Table 1. Statistics for the Large-Scale Generated Shapes in Figure 1 and 9. Shows total computation time and time for creating grammars. The remaining time is for rejection sampling and finding the match $m$ in Fig. 7. Shows number of iterations $N$, number of input primitives, output primitives, graphs in the hierarchy, and rules in the grammar.

| Name | Total Time | Gram. Time | $N$ | In Prim. | Out Prim. | Hier. Graph | # of Rules |
|---|---|---|---|---|---|---|---|
| 1. Castle | 9.5s | 0.7s | 4k | 24 | 181 | 1,030 | 88 |
| a. Cave | 18.1s | 0.2s | 8k | 27 | 655 | 425 | 87 |
| b. Station | 13.3s | 0.3s | 4k | 12 | 185 | 240 | 25 |
| c. Factory | 43.7s | 0.3s | 20k | 27 | 434 | 308 | 154 |
| d. Islands | 15.1s | 1.3s | 4k | 45 | 848 | 2,901 | 367 |
| e. Neigh. | 14.1s | 2.4s | 4k | 11 | 385 | 1,679 | 216 |
| f. Flowers | 10.5s | 0.04s | 2k | 9 | 364 | 20 | 8 |
| g. Village | 11.8s | 0.5s | 4k | 17 | 195 | 656 | 55 |
| h. Trees | 19.5s | 0.3s | 4k | 13 | 342 | 449 | 33 |
| i. Docks | 93.7s | 0.2s | 20k | 23 | 3,598 | 69 | 34 |
| j. Skyline | 44.6s | 1.2s | 4k | 36 | 7,066 | 170 | 44 |
| k. Houses | 57.0s | 0.3s | 20k | 11 | 1,716 | 45 | 13 |
| l. Sci-Fi | 78.8s | 0.8s | 10k | 21 | 4,659 | 351 | 23 |

Figures 1 and 9 shows several examples of how an artist can use our method to generate large complex shapes like castles, caves, space stations, processing plants, islands, neighborhood streets, flowers, villages, trees, houses, and city skylines. We apply bending in Figures 9a,d-g. Figures 8h,i,q,r and 9i-l show some preliminary results for 3D shapes.

Only a few of the 2D results require us to limit the size of the hierarchy as discussed in Section 5.7, namely Figures 1, 9e,g,h. And Figures 9e,h only have this problem because we add restrictions according to Appendix A. (Fig. 9e is fully connected and Fig. 9h has no loops.) 3D shapes provide more ways for the primitives to be glued together. Most of the 3D shapes require us to limit the size of the hierarchy which can impact the expressiveness of the grammar.

## 9 DISCUSSION

### 9.1 Comparison

Most procedural modeling techniques use hand-crafted rules designed by experts [Christiansen and Bærentzen 2012; Lindenmayer 1968; Müller et al. 2006; Parish and Müller 2001; Pogrzebacz and Ilčík 2019; Prusinkiewicz 1986; Smelik et al. 2014; Wong et al. 1998; Wonka et al. 2003]. There are many of these techniques and they can produce beautiful intricate shapes. But creating these rules is difficult especially for non-experts.

There are several inverse procedural modeling techniques, but they are usually restricted to particular types of shapes like building facades [Aliaga et al. 2007; Demir et al. 2016; Martinovic and Van Gool 2013; Wu et al. 2014] or trees [Guo et al. 2020; Stava et al. 2010, 2014] and cannot handle arbitrary shapes. These methods also do not explore new variations very well. They can create a grammar from an input shape and then a user could edit the grammar to

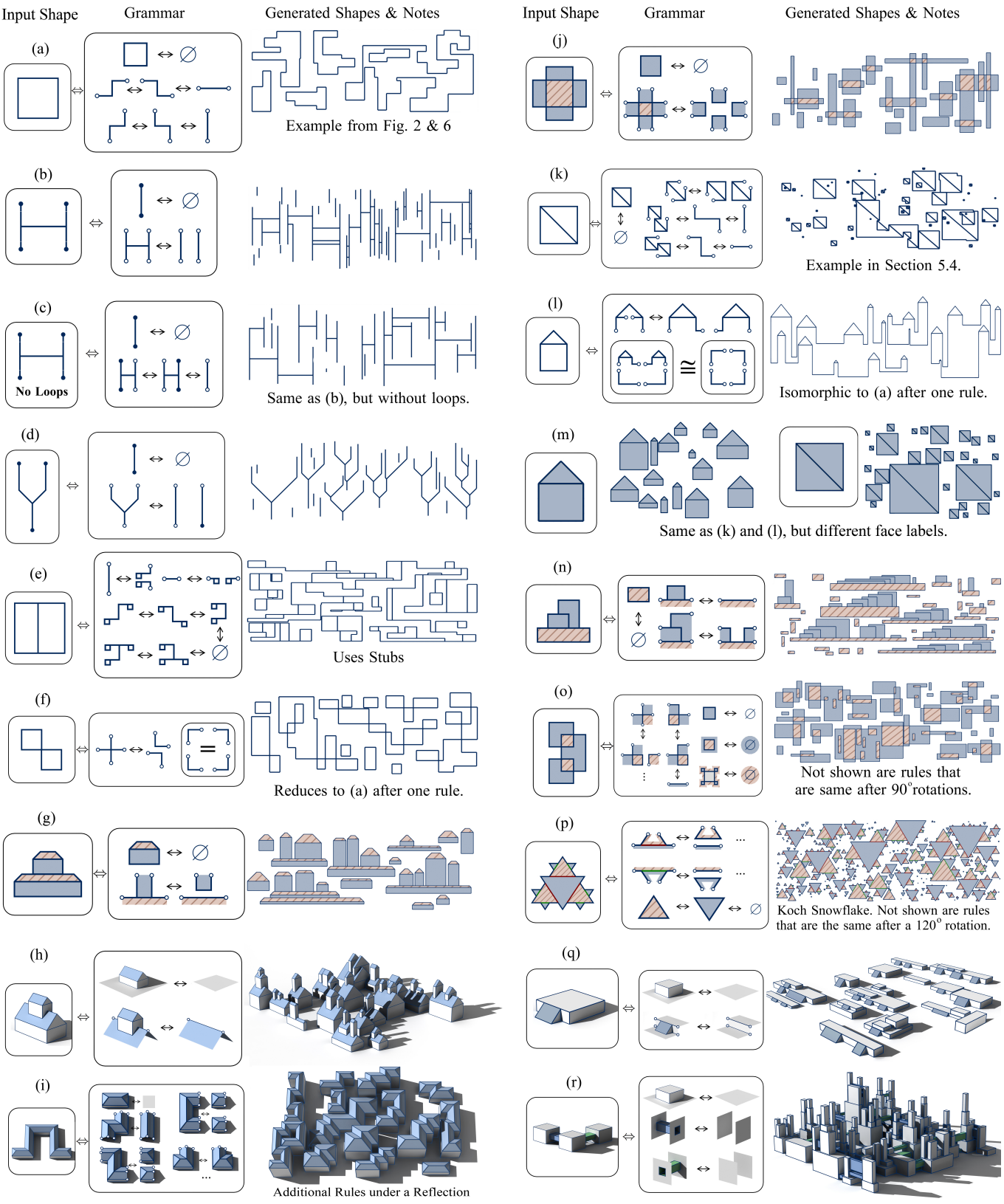



Fig. 8. From each set of primitives, a graph grammar is automatically generated and then it is used to generate shapes.

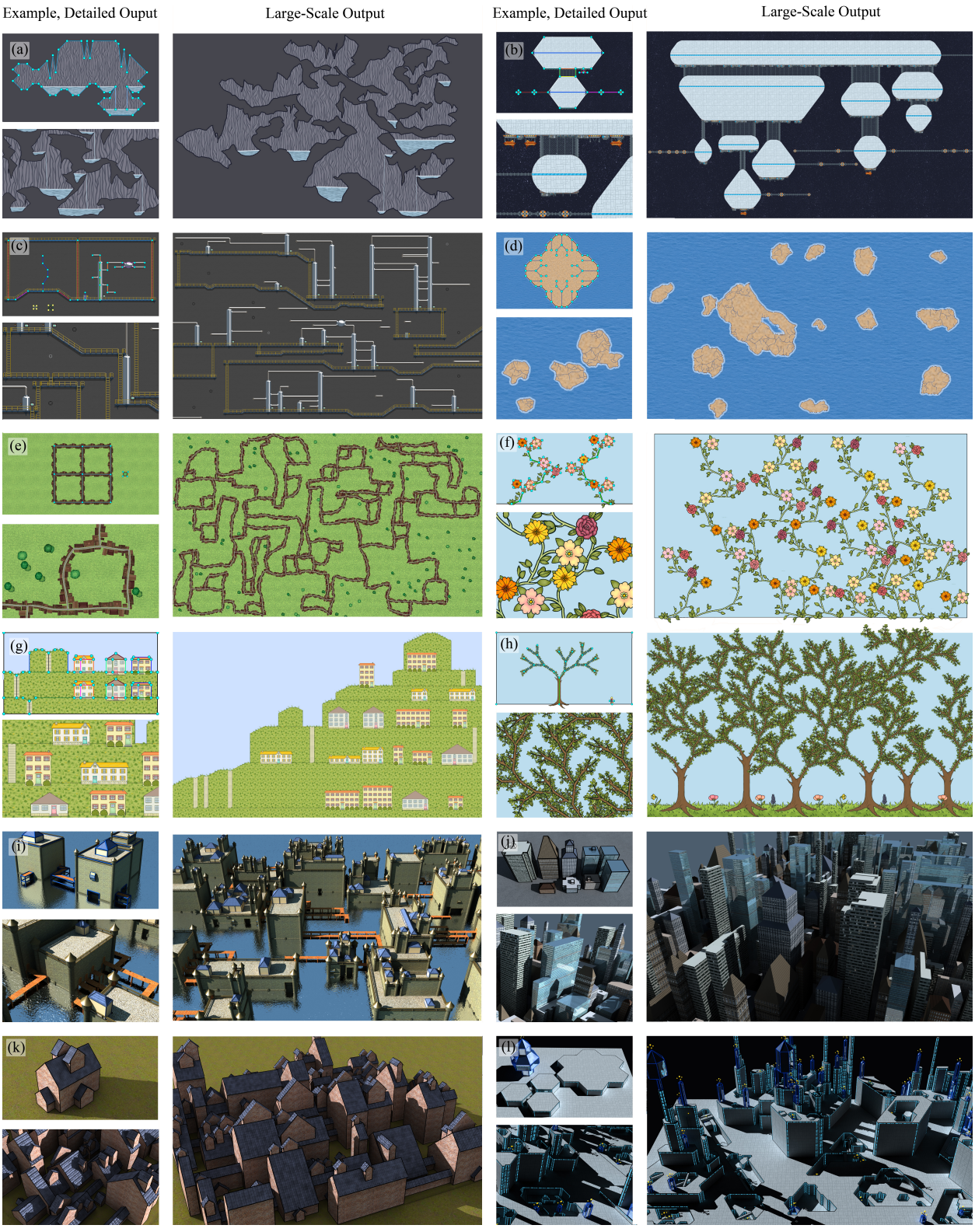



Fig. 9. From each example shape, a graph grammar is automatically generated. We show two output shapes produced from each grammar. One is large-scale and one is detailed.

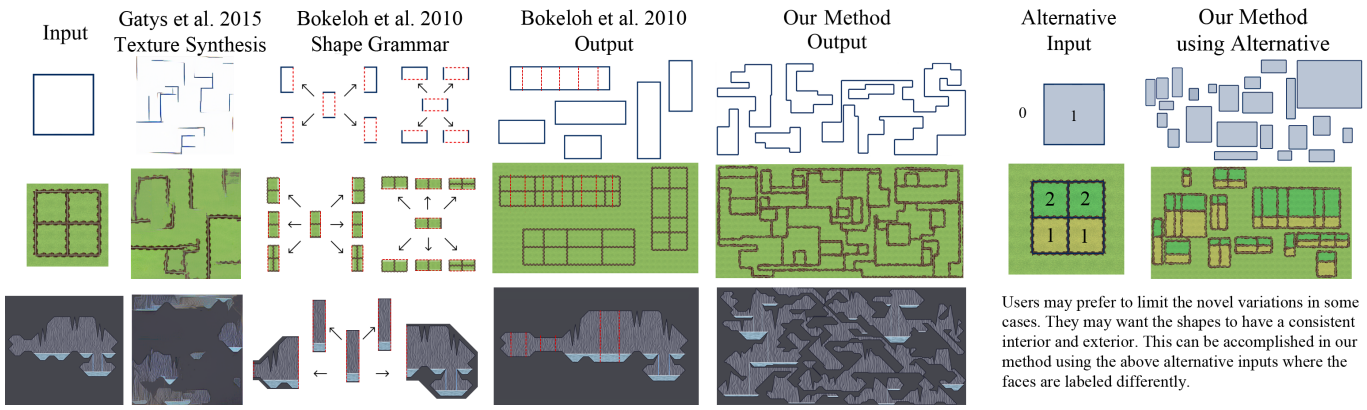



Fig. 10. Prior work has trouble with loops. Existing methods either do not close the loops [Gatys et al. 2015] or they directly copy the input with only minor changes to the aspect ratio [Bokeloh et al. 2010]. The red lines in Bokeloh et al.'s grammar are their docking sites.

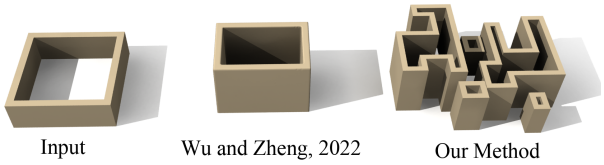



Fig. 11. 3D Comparison. Wu and Zheng, 2022 overfit to the input model.

produce slightly different output shapes. But the output is still quite similar to the input. Our method greatly expands and generalizes the input, exploring a large space of locally similar shapes.

Many techniques have a common failure point. Input shapes like Figure 8d that have a tree-like structure are relatively easy to handle. Several techniques handle them well such as the simple algorithm described in Section 4.4. Completing a path by attaching a dead end is simple. The difficulty comes when that is not an option and the paths must be formed into closed loops. Most of the inputs in Figures 8 & 9 have this property. Our method solves this difficult problem.

Texture synthesis techniques have trouble with this issue. They are designed to create locally similar textures. They work well on normal textures and on some rasterized geometric shapes. They work well on shapes with lots of dead ends like Figure 8d, but not well on shapes with closed loops. They break down in one of two ways. Either the result stops being locally similar or the result is almost identical to the input with no new variations explored. Figure 10 compares our method with a texture synthesis method [Gatys et al. 2015].

Wu and Zheng [2022] generate 3D shapes from an example. Their method can robustly handle noisy volumetric data, while ours assumes exact self-similarity. But their method overfits to the example and does not fully explore the space of locally similar shapes. In the example in Figure 11, their method only changes the shape's aspect ratio.

The most closely related method is the highly influential work of Bokeloh et al. [2010]. Their method takes an input shape and automatically creates a shape grammar that generates locally similar shapes. Their method cuts the input shape into pieces by finding partial symmetries where the input shape matches itself under a rigid transformation. But unlike our method which cuts the input into the smallest possible primitives, their method only allows cuts at *docking sites* defined as sites that divide the shape into two disconnected pieces. So while their method cuts the input into pieces, their pieces can be much larger than our primitives. And so their method cannot produce every locally similar shape. Their method works well for shapes with many dead ends like Figure 8d, but it has trouble with shapes with closed loops. Figure 10 shows that their method only produces minor variations on the input shape while our method produces the full range of variations. Furthermore their method generates similar limited results on most of the examples in Figures 8 and 9.

## 9.2 Practical Utility

A natural application of this technology is in video games and virtual worlds where a rich variety of shapes are needed. Our method is well-suited towards generating the type of randomized levels used in many games. It is also useful for creating decorative designs like floral patterns. Our method has the advantage of only requiring small amount of training data. It can generalize from a limited data set and create novel variations not found in it.

For this type of approach to work properly, the input shape must contain repeated elements. For instance, if the input was a model of a dog, our method would not find repeated elements and would not generate new varieties of dogs from the input. Our method assumes some edges repeat exactly and does not handle noisy input data.

## 9.3 User Control

Our method generates locally similar shapes at random. It enforces local constraints, but not large-scale constraints. Large-scale constraints are needed in many applications to give the user more control over the results. Learning large-scale constraints from examples is challenging and would require a larger training set.

In Section 6.3, we discussed a simple way of combining our method with an MCMC approach to optimize for an arbitrary cost function. Such an approach can apply broadly to many different

goals, but can be slow. And formulating the right cost function is also an important, difficult task that greatly depends on the type of application.

Several other methods focus more on large-scale constraints. Some methods are targeted to specific applications like road networks [Vanegas et al. 2012] or plant ecosystems [Pałubicki et al. 2022]. While others discuss constraints that apply across a wide range of shapes [Dang et al. 2015; Ritchie et al. 2015; Talton et al. 2011]. Our method could be combined with some of these approaches in future work.

### 9.4 Limitations and Future Work

More work is needed to handle the large set of possible gluing combinations for 3D shapes. More sophisticated tools are needed to narrowly focus on parts of the graph hierarchy where production rules can be found rather than such a broad search.

Section 6 describes how create planar graph drawing through rejection sampling. This works fine for most grammar rules, but in some cases it is very difficult to find planar graph drawings. More sophisticated tools for generating planar graph drawings are needed.

We assume the edges can be stretched enough that forming 360° loops without self-intersection is not difficult. But this assumption breaks down when there are tight restrictions on the edge lengths such as when the input is a set of connectable tiles. This scenario is handled well by tile-based techniques [Merrell 2007]. It may be possible to combine these two approaches.

Our method assumes the input contains exact self-similarity and does not handle noisy input data. We can bend and deform the shapes as a post-processing step, but these deformations could be considered when creating the grammar.

## ACKNOWLEDGMENTS

Thank you to Sylvain Lefebvre, Martin Ilčík, Élie Michel, Vladlen Koltun, and Martin Bokeloh for helpful comments and suggestions. Thank you to Morgan McGrath and Abigail Chamberlain for illustrations used to decorate the 2D results. Patent Pending.

## REFERENCES

Daniel G. Aliaga, Paul A. Rosen, and Daniel R. Bekins. 2007. Style Grammars for Interactive Visualization of Architecture. *IEEE Transactions on Visualization and Computer Graphics* 13, 4 (2007), 786–797.

Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. 2009. PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing. *ACM Transactions on Graphics (Proc. SIGGRAPH)* 28, 3 (2009).

Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. 2010. A Connection between Partial Symmetry and Inverse Procedural Modeling. *ACM Trans. Graph.* 29, 4 (2010).

Martin Bokeloh, Michael Wand, Hans-Peter Seidel, and Vladlen Koltun. 2012. An Algebraic Model for Parameterized Shape Editing. *ACM Trans. Graph.* 31, 4 (2012).

Asger Nyman Christiansen and Jakob Andreas Bærentzen. 2012. Generic graph grammar: a simple grammar for generic procedural modelling. In *SCCG*.

Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures *(STOC '71)*. Association for Computing Machinery, New York, NY, USA, 151–158.

George R. Cross and Anil K. Jain. 1983. Markov Random Field Texture Models. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-5, 1 (1983), 25–39.

Minh Dang, Stefan Lienhard, Duygu Ceylan, Boris Neubert, Peter Wonka, and Mark Pauly. 2015. Interactive Design of Probability Density Functions for Shape Grammars. *ACM Trans. Graph.* 34, 6 (2015).

İlke Demir, Daniel G. Aliaga, and Bedrich Benes. 2016. Proceduralization for Editing 3D Architectural Models. In *2016 Fourth International Conference on 3D Vision (3DV)*. 194–202.

Alexei Efros and Thomas Leung. 1999. Texture synthesis by non-parametric sampling. In *ICCV*, Vol. 2. 1033–1038.

Hartmut Ehrig. 1979. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Application to Computer Science and Biology*, Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–69.

H. Ehrig, M. Pfender, and H. J. Schneider. 1973. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. 167–180.

David Eppstein. 1999. Subgraph isomorphism in planar graphs and related problems. *Journal of Graph Algorithms and Applications* 3, 1-3 (1999).

Marek Fiser, Bedrich Benes, Jorge Garcia Galicia, Michel Abdul-Massih, Daniel G. Aliaga, and Vojtech Krs. 2016. Learning Geometric Graph Grammars. In *Proceedings of the 32nd Spring Conference on Computer Graphics (SCCG '16)*. Association for Computing Machinery, New York, NY, USA, 7–15.

Ashim Garg. 1998. New results on drawing angle graphs. *Computational Geometry* 9, 1 (1998), 43–82. Special Issue on Geometric Representations of Graphs.

Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. 2015. Texture Synthesis Using Convolutional Neural Networks *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 262–270.

Maxim Gumin. 2016. *WaveFunctionCollapse*. https://github.com/mxgmn/WaveFunctionCollapse

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 Transactions on Graphics* 39, 5 (2020), 155:1–155:13.

Aaron Hertzmann, Nuria Oliver, Brian Curless, and Steven M. Seitz. 2002. Curve Analogies. In *Proceedings of the 13th Eurographics Workshop on Rendering* (Pisa, Italy) *(EGRW '02)*. Eurographics Association, 233–246.

Takashi Ijiri, Radomír Mêch, Takeo Igarashi, and Gavin Miller. 2008. An Example-based Procedural System for Element Arrangement. *Computer Graphics Forum* 27, 2 (2008), 429–436.

Javor Kalojanov, Martin Bokeloh, Michael Wand, Leonidas Guibas, Hans-Peter Seidel, and Philipp Slusallek. 2012. Microtiles: Extracting Building Blocks from Correspondences. *Comput. Graph. Forum* 31, 5 (2012), 1597–1606.

Barbara König, Dennis Nolte, Julia Padberg, and Arend Rensink. 2018. *A Tutorial on Graph Transformation*. Springer International Publishing, Cham, 83–104.

Johannes Kopf, Chi-Wing Fu, Daniel Cohen-Or, Oliver Deussen, Dani Lischinski, and Tien-Tsin Wong. 2007. Solid Texture Synthesis from 2D Exemplars. *ACM Trans. Graph.* 26, 3 (2007).

Aristid Lindenmayer. 1968. Mathematical models for cellular interactions in development. II. Simple and branching filaments with two-sided inputs. *J Theor Biol* 18, 3 (1968), 300–315.

Markus Lipp, Peter Wonka, and Michael Wimmer. 2008. Interactive Visual Editing of Grammars for Procedural Architecture. *ACM Transactions on Graphics (Proc. SIGGRAPH)* (2008).

Han Liu, Ulysse Vimont, Michael Wand, Marie-Paule Cani, Stefanie Hahmann, Damien Rohmer, and Niloy Mitra. 2015. Replaceable Substructures for Efficient Part-Based Modeling. *Computer Graphics Forum* 34 (05 2015).

Chongyang Ma, Li-Yi Wei, Sylvain Lefebvre, and Xin Tong. 2013. Dynamic Element Textures. *ACM Trans. Graph.* 32, 4 (2013).

Chongyang Ma, Li-Yi Wei, and Xin Tong. 2011. Discrete Element Textures. *ACM Trans. Graph.* 30, 4 (2011).

Andelo Martinovic and Luc Van Gool. 2013. Bayesian Grammar Learning for Inverse Procedural Modeling. In *CVPR*. 201–208.

Paul Merrell. 2007. Example-Based Model Synthesis. *Symp. Interactive 3D Graphics and Games* (2007), 105–112.

Paul Merrell and Dinesh Manocha. 2008. Continuous Model Synthesis. *ACM Trans. Graph.* 27, 5 (2008).

Paul Merrell and Dinesh Manocha. 2010. Example-based curve synthesis. *Computers & Graphics* 34, 4 (2010), 304–311.

Paul Merrell and Dinesh Manocha. 2011. Model Synthesis: A General Procedural Modeling Algorithm. *IEEE Transactions on Visualization and Computer Graphics* 17, 6 (2011), 715–728.

Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics* 21, 6 (1953), 1087–1092.

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

Wojtek Pałubicki, Miłosz Makowski, Weronika Gajda, Torsten Hädrich, Dominik L. Michels, and Sören Pirk. 2022. Ecoclimates: Climate-Response Modeling of Vegetation. *ACM Trans. Graph.* 41, 4, Article 155 (jul 2022), 19 pages.

Yoav I. H. Parish and Pascal Müller. 2001. Procedural Modeling of Cities *(SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 301–308.

John L. Pfaltz and Azriel Rosenfeld. 1969. Web Grammars. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (Washington, DC) *(IJCAI'69)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 609–619.

Viktor Pogrzebacz and Martin Ilčík. 2019. A Graph Grammar for Modelling of 2D Shapes. In *Proceedings of the 23rd Central European Seminar on Computer Graphics*.

TU Wien.
Przemyslaw Prusinkiewicz. 1986. Graphical applications of l-systems. In *In Proceedings of Graphics Interface '86 — Vision Interface '86*. 247–253.
Daniel Ritchie, Ben Mildenhall, Noah D. Goodman, and Pat Hanrahan. 2015. Controlling Procedural Modeling Programs with Stochastically-Ordered Sequential Monte Carlo. *ACM Trans. Graph.* 34, 4, Article 105 (jul 2015), 11 pages.
Grzegorz Rozenberg. 1997. *Handbook of graph grammars and computing by graph transformation*. Vol. 1. World scientific.
Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. 2014. A Survey on Procedural Modelling for Virtual Worlds. *Computer Graphics Forum* 33, 6 (2014), 31–50.
Colin Smith, Przemyslaw Prusinkiewicz, and Faramarz Samavati. 2004. Local Specification of Surface Subdivision Algorithms. In *Applications of Graph Transformations with Industrial Relevance*. Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.
Ondrej Stava, Bedrich Benes, Radomir Mech, Daniel Aliaga, and Peter Kristof. 2010. Inverse Procedural Modeling by Automatic Generation of L-systems. *Computer Graphics Forum* 29 (05 2010), 1467–8659.
O. Stava, S. Pirk, J. Kratt, B. Chen, R. Mźch, O. Deussen, and B. Benes. 2014. Inverse Procedural Modelling of Trees. 33, 6 (2014), 118–131.
George Stiny. 1975. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, Basel.
George Stiny. 1982. Spatial relations and grammars. *Environment and Planning B* 9 (1982), 313–314.
Jerry Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah Goodman, and Radomír Měch. 2012. *Learning Design Patterns with Bayesian Grammar Induction*. Association for Computing Machinery, 63–74.
Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. 2011. Metropolis Procedural Modeling. *ACM Trans. Graph.* 30, 2 (2011).
Peihan Tu, Li-Yi Wei, Koji Yatani, Takeo Igarashi, and Matthias Zwicker. 2020. Continuous Curve Textures. *ACM Trans. Graph.* 39, 6 (2020).
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 (2012).
Luiz Velho. 2003. Stellar Subdivision Grammars. In *Proceedings of the 2003 Eurographics ACM SIGGRAPH Symposium on Geometry Processing (SGP '03)*. Eurographics Association, 188–199.
Michael T. Wong, Douglas E. Zongker, and David H. Salesin. 1998. Computer-Generated Floral Ornament *(SIGGRAPH '98)*. Association for Computing Machinery, New York, NY, USA, 423–434.
Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. 2003. Instant Architecture. *ACM Trans. Graph.* 22, 3 (2003), 669–677.
Fuzhang Wu, Dong-Ming Yan, Weiming Dong, Xiaopeng Zhang, and Peter Wonka. 2014. Inverse Procedural Modeling of Facade Layouts. *ACM Trans. Graph.* 33, 4 (2014).
Rundi Wu and Changxi Zheng. 2022. Learning to Generate 3D Shapes from a Single Example. *ACM Trans. Graph.* 41, 6, Article 224 (nov 2022), 19 pages.
Yi-Ting Yeh, Katherine Breeden, Lingfeng Yang, Matthew Fisher, and Pat Hanrahan. 2013. Synthesis of Tiled Patterns Using Factor Graphs. *ACM Trans. Graph.* 32, 1 (2013).
Allan Zhao, Jie Xu, Mina Konaković Luković, Josephine Hughes, Andrew Speilberg, Daniela Rus, and Wojciech Matusik. 2020. RoboGrammar: Graph Grammar for Terrain-Optimized Robot Design. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–16.

## A SPECIAL CASES

*Infinitely-Long Lines.* We may want the example and output shapes to contain lines that do not end. For example, a line for the ground extends indefinitely. Our method generates shapes within a finite space. We can treat the border of the space as part of the input shape. Lines that extend infinitely far are modeled as lines that intersect this border.

*Connected Shapes.* We may want the output graph to be fully connected. This can be done by (1) not allowing starter rules to be used more than once, and (2) disallowing rules that use multiple spliced graphs.

*No Loops.* We also may want our output to not contain loops such as for streams and trees. This can be done by (1) disallowing loop gluing and (2) disallowing rules that use multiple spliced graphs.

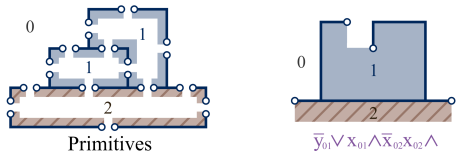
## B DECORATED GRAPH DRAWINGS

Decorations can be added to the graph drawings as an optional post-processing step. Adding decorations like this is a common practice in many procedural modeling techniques [Müller et al. 2006; Wong et al. 1998]. This allows us to use a simplified representation that acts as a proxy for a more detailed final result. The user can specify images or geometry that should be added or tiled.

*Bending.* The user can also allow some edges to bend. To bend the edges, we traverse each edge in the graph, divide it into small segments and bend each segment by a random angle. The result is a new bent graph drawing. But this is just an intermediate solution since the bending can break some of the cycles in the drawing. The final positions are found using linear least squares. We find the optimal positions such that the change in position between two adjacent points matches the bent drawing as closely as possible while preserving all cycles.

## C NO COMPLETE DESCENDANTS EXAMPLE

Following the discussion in Section 5.6, we wish to show that a graph has no complete descendants. Consider the primitives below left and the graph below right. We will show that the graph $\overline{y_{01}} \vee x_{01} \wedge \overline{x_{02}} x_{02} \wedge$ has no complete descendants:




Note that the half-edge $\overline{y_{01}}$ is pointed in the $-y$ direction and is adjacent to the faces labeled 0 and 1. A graph can be completed if there is some sequence of gluing operations that can transform the boundary string to $\wedge$. This is impossible. Consider the half-edge $\overline{y_{01}}$. There is no way to loop glue it. Here is a decision tree showing the result of every possible graph gluing operation starting with $\overline{y_{01}}$:

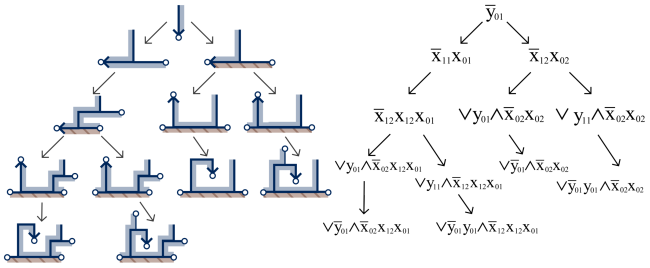



Fig. 12. A decision tree of possible gluing operations starting with $\overline{y_{01}}$.

No matter which choice we take we never get any closer to completing the graph. Gluing has the effect of doing a string replacement. We keep trying to replace the first half-edge, but no matter which decision we make we end up back with $\overline{y_{01}}$ as the first half-edge and a turn added in the wrong direction. Every path down the decision tree cycles: $\overline{y_{01}} \rightarrow \overline{x_{11}} \rightarrow \overline{x_{12}} \rightarrow \vee y_{01} \wedge \rightarrow \vee \overline{y_{01}} \wedge$ without us getting any closer to completing the graph. It is impossible to complete the graph $\overline{y_{01}} \vee x_{01} \wedge \overline{x_{02}} x_{02} \wedge$.