Program Synthesis for Geometric Modeling

Romain Pascual[1](✉), Pascale Le Gall[1], Hakim Belhaouari[2], and Agnès Arnould[2]

[1] MICS, CentraleSupélec, Université Paris-Saclay, France
`{romain.pascual,pascale.legall}@centralesupelec.fr`
[2] Université de Poitiers, Univ. Limoges, CNRS, XLIM, Poitiers, France
`{hakim.belhaouari,agnes.arnould}@univ-poitiers.fr`

Abstract. Implementing geometric modeling operations in a programming language can be inherently challenging despite their seemingly simple input-to-output descriptions. We propose a program synthesis method to generate executable code from representative examples. We focus on geometric computations in topology-based modeling, where nD objects are decomposed into cells with added geometric information. This domain uses combinatorial structures represented as graphs, with operations formalized as graph transformation rules and geometric modifications given by code-like annotations on the rule's graph nodes.

Keywords: Programming-by-example · Domain-specific language · Topology-based geometric modeling · Constraint-solving.

1 Introduction

Program synthesis, often described as "the holy grail of Computer Science" [23], aims to automatically generate executable programs from high-level specifications, alleviating the burden of manual coding. Inductive synthesis, in particular, infers programs from input-to-output examples [4, 50], such as the FlashFill feature in Microsoft Excel [21]. In this work, we investigate program synthesis in a specific application domain, geometric modeling, which provides methods and algorithms to represent and manipulate geometric shapes. Geometric modeling operations are typically implemented in a low-level programming language like C or C++, with advanced performance optimizations, making them difficult to adapt. While APIs exist to define custom operations [9, 20, 49], they often exceed domain experts' skills. Therefore, a means to create custom operations remains a longstanding aspiration, one that a program synthesis approach could fulfill. Rather than synthesizing low-level code, we propose to synthesize modeling operations in a Domain-Specific Language (DSL) based on graph transformations [14, 25] at the core ot the Jerboa platform [3] to benefit from the high-level representation of operations and associated formal verification mechanisms [2, 42].

Related Work Prior work has explored program synthesis for generating geometric objects with desired properties, e.g., from patterns [7], program traces [8, 26], ruler-compass constructions [22], geometric constraints [37], or an underlying L-sytems [24, 45, 52]. While not advertised as program synthesis, inverse procedural modeling aims to retrieve the procedure that generated an object to build similar-looking objects from variations of the procedure [15, 27, 51]. Other works have applied program synthesis techniques to CAD using Constructive Solid Geometry (CSG) [30, 47]. In particular, [13] proposes to synthesize CSG trees from surface meshes using the Sketch system [48]. These methods are similar in spirit to ours but distinct in goal: they aim to generate objects, whereas we are interested in synthesizing transformations of geometric objects. Closer to our goal, in [34] the authors generate geometric expressions from examples with a neural network learning the parameters of a subdivision scheme. While interesting, this approach diverges from our goal as we aim to compute the target geometry while supporting arbitrary topological changes.

Contributions We propose an automatic generation of geometric expressions over arbitrary topological modifications from an input-to-output example, tailoring established synthesis methods to geometric modeling. Building on Jerboa's DSL and prior work on inferring topological transformations [41], we complete the synthesis of modeling operations by retrieving the missing geometric expressions. Inspired by component- and sketch-based synthesis [22, 28, 48], we treat the synthesis task as completing a sketch skeleton describing an affine combination of components, called *values of interest*, grounded in topological abstractions. The resulting constraints, derived from the input-to-output example provided by the user, are solved via a solver. We present JerboaStudio, a tool integrated into the Jerboa platform, to empirically validate our method.

Paper Organization Sec. 2 clarifies our goal using a running example, the flat extrusion, while Sec. 3 revisits generalized maps for object representation and Jerboa's DSL. We outline our synthesis method in Sec. 4, with practical implementation details discussed in Sec. 5. Sec. 6 evaluates our framework with a benchmark on subdivision schemes. Sec. 7 provides some concluding remarks.

2 Motivation and Running Example

To illustrate our motivation, consider a 2D flat extrusion. The user provides two objects (Fig. 1a): an initial pentagonal face and a modified version with a shrunk copy, sharing the same barycenter and connected vertex-to-vertex with the original. Our goal is to synthesize a program that computes the new vertex positions. Using the annotations in Fig. 1b, we aim to compute the *output* positions $o_1, \ldots, o_5$ (in orange) from the *input* positions $i_1, \ldots, i_5$ (in blue). Since the added face is a homothety, each o_k for$k \in 1..5$ can be expressed as a linear combination $o_k = p \cdot i_k + (1 - p) \cdot C$ where i_k is the corresponding input vertex, $C = \frac{1}{5} \sum_{k=1}^{5} i_k$ is the barycenter of the original face, and p is the homothety ratio. Since all o_k share the same expression, the synthesis task reduces to inferring the correct value of p and producing a program that computes this expression.

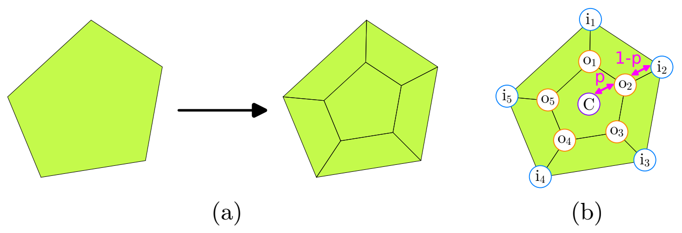


Fig. 1. Synthesis of a geometric computation: (a) a geometric modeling operation, (b) explanation about the geometric computations.

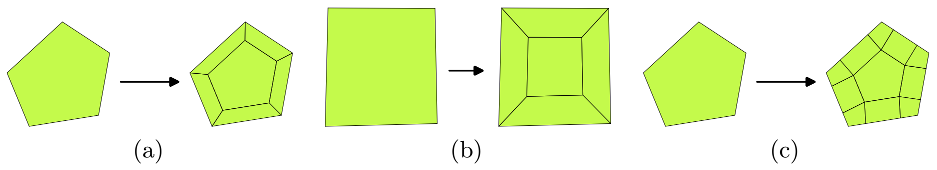


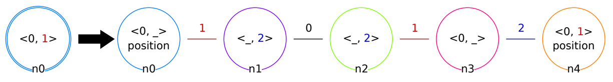
Fig. 2. Variations on the modeling operation of Fig. 1a: (a) different geometry, (b) different initial topology, (c) different topological modification.

Importantly, the expression must generalize beyond pentagons to support reuse across contexts. For example, a quad as inputs (see Fig. 2b) yields the same equation for each o_k with now $k \in 1..4$. We abstract the computation into the symbolic expression $O = p \cdot I + (1 - p) \cdot C$, where I and O are meta-variables for the input and output families. From this symbolic form, we can recover the concrete expressions with the input-to-output example of Fig. 1a.

The objects of Fig. 1a are meshes, described by an internal arrangement of cells (vertices, edges, faces) – the *topology* – enriched with data attached to the cells (positions to vertices, curvatures to edges, and colors to faces) – the *geometry*. The operation in Fig. 2a uses the same topological modification as the flat extrusion of Fig. 1a but with different positions: the geometric computation only differs in the value of p. Fig. 2b results from the same operation but applied to a quad. If we synthesize the operation from the pentagonal example in Fig. 1a and apply it to this quad, we should recover the correct output. Put differently, both input-to-output examples should synthesize the same operation. Finally, Fig. 2c shows a different modeling operation, the ternary subdivision [32], which performs a different topological modification.

3 A DSL for Topology-based Geometric Modeling

Geometric modeling requires an internal representation of shapes. For our synthesis framework, the representation must be precise enough to express fine-grained edits but regular enough to support automated reasoning. We adopt *generalized maps* (Gmaps) [11,33], which offer a combinatorial structure to describe how cells (vertices, edges, faces, etc.) are connected. Gmaps are well-suited for our synthesis framework because (1) they support arbitrary-dimensional meshes (e.g., surfaces,



(a)

```
// Position for n0
return n0.position;
```

(b)

```
// Position for n4
return Point3::midpt(Point3::middle(<0,1>_position(n0)),n0.position);
```

(c)

Fig. 3. Flat extrusion: (a) rule scheme, (b) expression for $n0$, (c) expression for $n4$.

volumes); (2) they can be interpreted as graphs with regular patterns, enabling rule-based rewriting; (3) they separate the *topology* (connectivity) from the *geometry* (e.g., positions, colors). With the encoding of Gmaps as graphs, modeling operations can be described as graph rewriting rules [14, 25]. These consist of a left-hand side (LHS) graph identifying a pattern to match and a right-hand side (RHS) graph for the modified pattern. In Jerboa, rules are generalized to rule schemes as illustrated in Fig. 3a, where the rule structure defines topological edits and node annotations (e.g., `position`) provides the computational expressions to update the geometry. These expressions, given in Figs. 3b and 3c, are the target of our synthesis framework detailed in Sec. 4. The remainder of this section introduces the formal background required to interpret such rules.

3.1 Embedded Generalized Maps

Generalized maps can be formalized as a specific subclass of graphs with constraints that ensure the topological correctness of the structure.

Definition 1 (Generalized map (see [42])). *Given $n \in \mathbb{N}$, a* generalized map *of dimension n, n-Gmap or simply* Gmap*, is an undirected graph $G = (N, A)$ with arcs labeled by a dimension d in $0..n$ (then called a d-arc) satisfying:*

Incidence constraint: *each node has exactly one incident arc per dimension,*
Cycle constraint: *every path labelled $ijij$ (with $i + 2 \leq j$) forms a cycle.*

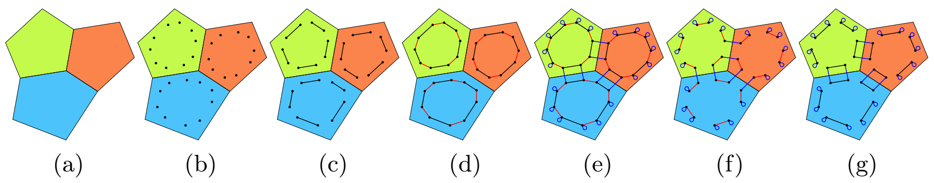


Fig. 4. Gmap construction: (a) 2D object, (b) darts (●) for each valid vertex-edge-face triplet, (c) 0-arcs (●━●) link darts in the same face and edge but not the same vertex, (d) 1-arcs (●━●) link edges, (e) 2-arcs (●━●) link faces. Cells: (f) $\langle 1, 2\rangle$-orbit (vertices), (g) $\langle 0, 2\rangle$-orbit (edges), (d) $\langle 0, 1\rangle$-orbit (faces).

The incidence constraint enables unambiguous path descriptions as sequences of dimensions. To distinguish between nodes in Gmaps and in our DSL rules, we refer to the nodes in N as *darts*, following [11]. The semantics of a dart is given by a tuple of incident cells, and a d-arc connects darts differing only at dimension d. In a 2-Gmap, darts represent vertex-edge-face triplets, and 0-arcs connect darts in different vertices of the same edge and face. Fig. 4 illustrates the progressive construction of a Gmap: darts are created (Fig. 4b), then connected via arcs (Figs. 4c to 4e), forming the object's topology. Darts on the boundary of a cell are self-linked along the corresponding dimension (see 2-loops in Fig. 4e). Cells, such as vertices, edges, or faces, correspond to subgraphs called orbits.

Definition 2 (Orbit (see [42])). *Let G be an n-Gmap, v a dart of G and $o \subseteq 0..n$ a subset of dimensions. The* orbit *$G\langle o\rangle(v)$ is the maximal subgraph induced by the dimensions in o and containing v.*

The orbit $G\langle o\rangle(v)$ is of type $\langle o\rangle$ or is an $\langle o\rangle$-orbit.

The $\langle o\rangle$-orbits are the connected components after ignoring arcs labeled outside $\langle o\rangle$. Since 0-arcs split vertices but preserve edges and faces, $\langle 1, 2\rangle$-orbits contain all darts belonging to a vertex and encode the vertices (Fig. 4f). Similarly, faces correspond to $\langle 0, 1\rangle$-orbits (Fig. 4d) and edges to $\langle 0, 2\rangle$-orbits (Fig. 4g).

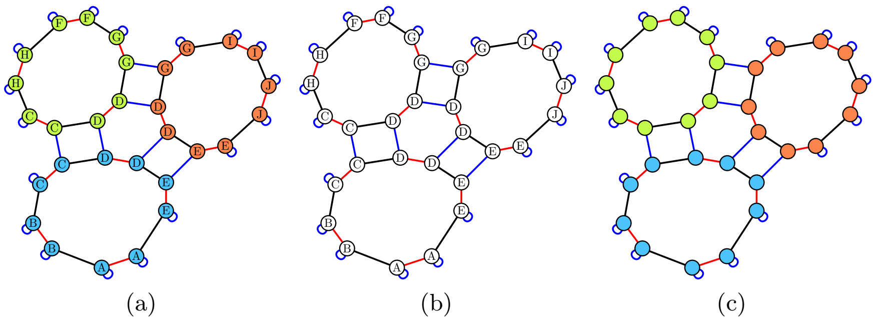


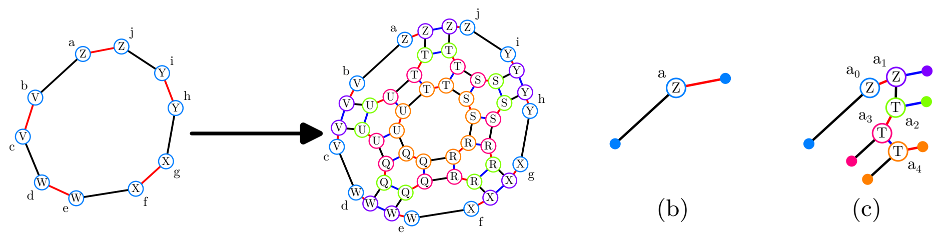
Fig. 5. Embeddings: (a) embedded Gmap, (b) `position` : $\langle 1, 2\rangle \rightarrow$ `Point3`, (c) `color` : $\langle 0, 1\rangle \rightarrow$ `Color3`.

Geometry and attributes, such as vertex positions or face colors, are attached via embedding functions associating orbits to data types [2], similar to graph attribution [14, 25]. For example, vertex positions are encoded by `position` : $\langle 1, 2\rangle \rightarrow$ `Point3` and face colors by `color` : $\langle 0, 1\rangle \rightarrow$ `Color3`.

Definition 3 (Embedded generalized map (see [2])). *Let $\pi : \langle o_\pi\rangle \rightarrow \tau_\pi$ be an embedding symbol π together with an orbit type $\langle o_\pi\rangle$ and a data type τ_π. A Gmap embedded on π is a pair (G, π_G) where $G = (N, A)$ is a Gmap and π_G is a function $N \rightarrow \tau_\pi$ satisfying:*

Embedding constraint *all darts of an $\langle o_\pi\rangle$-orbit share the same value of type τ_π.*

Embedded Gmaps can be extended to support multiple embedding symbols. Fig. 5 shows the position and color embeddings of the Gmap from Fig. 4.



(a)

Fig. 6. Instantiation of the flat extrusion based on the rule scheme of Fig. 3a: (a) instantiated rule, (b) zoom on the LHS and (c) RHS instantiations, a_0 in (c) coincides with a in (b) sharing the same position Z.

3.2 Modeling Operations as Graph Transformations

In Jerboa [3], rules are extended to rule schemes parameterized by orbit types to abstract topological changes [42]: nodes become placeholders for orbits, labeled by their orbit type. Replacing a node with an orbit is called *instantiation*. A distinguished LHS node, the *hook*, anchors the matching. Other nodes encode transformations via relabeling functions derived from the hook's orbit type and theirs. Fig. 3a shows the rule scheme for the flat extrusion from Fig. 1. The hook $n0$ (double line) of type $\langle 0, 1\rangle$ is modified to $\langle 0, \_\rangle$, encoding the function $\langle 0, 1\rangle \mapsto \langle 0, \_\rangle$, i.e., $\{0 \mapsto 0, 1 \mapsto \_\}$. The symbol _ denotes the removal of the initial dimension. Node $n4$ with type $\langle 0, 1\rangle$ encodes the identity function, creating the homothetic copy of the initial face.

Instantiating this rule on a pentagon (Fig. 6a) replaces the hook $n0$ with the $\langle 0, 1\rangle$-orbit containing the darts $a, \ldots, j$. Five copies are created for each RHS node (color-coded to match the nodes in Fig. 3a), relabeled accordingly. Finally, the copies are connected following the arcs of the rule scheme: an arc between two nodes results in an arc between each pair of twin darts in the two copies. Figs. 6b and 6c zoom on dart a. Subscripts indicate the node index from Fig. 3a. The transformations builds a 1012-path between a_0 and a_4, while arcs connecting same-color darts originate from the node orbit types. This high-level explanation elucidates the abstraction used within our synthesis framework. For further details on the instantiation, we refer the reader to the set-based explanation in [39].

In Fig. 3a, RHS nodes $n0$ and $n4$ carry geometric expressions as indicated by the annotation `position`. For $n0$, the expression (Fig. 3b) preserves the initial positions. For $n4$, the expression (Fig. 3c) computes:

$$\texttt{midpt}(\texttt{middle}(\langle 0, 1\rangle\_\,\texttt{position}(n0)), \texttt{position}(n0)). \quad (1)$$

where `position` gives the position, `midpt` the midpoint between two positions, `middle` the barycenter of a collection of positions, and $\langle 0, 1\rangle\_$ `position` a collection of positions in an orbit $\langle 0, 1\rangle$. This collection is retrieved by a dedicated operator in Jerboa's DSL, leveraging the high regularity of Gmaps to gather embedding values within an orbit. During instantiation, $n0$ is substituted by the darts associated with $n0$ before modification. For dart a, we derive $\texttt{midpt}(\texttt{middle}(\langle 0, 1\rangle\_\texttt{position}(a)), \texttt{position}(a))$. Substituting $n0$ by a makes the expression computable as it solely relies on dart identifiers. First, $\texttt{middle}(\langle 0, 1\rangle\_\texttt{position}(a))$ computes the barycenter $C = \frac{1}{5}\sum_{k=1}^{5} i_k$ from Fig. 1b. All darts $a, \ldots, j$ compute the same value. Then, $\texttt{midpt}(\ldots, \texttt{position}(a))$ computes the midpoint between the barycenter and the position of a, as in the expression $p.I + (1-p).C$ of Sec. 2 with $p = \frac{1}{2}$.

Geometric expressions for rules over embedded Gmaps can be formalized with algebraic data types [2]. An embedded Gmap enriches a Gmap with an algebra over a signature combining node variables (from LHS nodes), standard operations (e.g., `middle` for the data type `Point3`), and topological operators such as collect operators $\pi_{\langle o \rangle}$, (e.g., $\langle 0, 1\rangle\_\texttt{position}$ in Equation (1)), retrieving τ_π-values within an orbit $\langle o \rangle$. These last operators allow concise expressions for geometric computations, justifying the expression $O = p \cdot I + (1-p) \cdot C$ in Sec. 2. Jerboa allows the user to declare and implement new functions (in Java) to be used in an imperative language for geometric expressions, part of Jerboa's DSL. The expressions are replaced with the implemented functions when the rule is translated into an efficient program via code generation. In this work, we will use a fragment of the native expressions available in Jerboa (see Sec. 4.2).

4 Synthesizing Geometric Expressions

Sec. 3 explained that Jerboa's DSL edits the topology via graph rewriting rules and the geometry via node-based expressions, enabling the separate synthesis of each kind of change. In [41], we introduced the *topological folding algorithm* extracting topological changes by folding an input-to-output example into a rule scheme, leaving only the geometric expressions to be synthesized. Embeddings, parameterized by orbit types, anchor geometric expressions in the topological structure, providing convenient access to *values of interest*. We propose to synthesize linear combinations of such values of interest.

Program synthesis faces two core challenges: the intractability of the program space and the interpretation of the user intent [23]. Designing a program synthesis method involves choosing how to represent the user intent, define the program space, and analyze it to find a candidate program. These choices are interdependent; for example, a grammar-encoded search space favors enumeration-based analysis [1]. We now elucidate our choices and rationale, while focusing on the synthesis of positions.

4.1 User Intent

We adopt a programming-by-example approach to synthesize geometric expressions of modeling operations, using the same user-provided input-to-output example and leveraging intermediate results from the topological folding algorithm [41]. The user provides (1) an input instance, (2) an output instance, (3) a mapping of preserved parts. This process is comparable to the substring analysis in FlashFill [21] but becomes exponentially harder with graphs (see [38, Sec. 7.2] for details). The algorithm also records the set of darts in the input-to-output example associated with each node of the rule scheme. We aim to automatically insert geometric expressions to ensure that instantiating the rule scheme yields the input-to-output example.

The first step is to identify the RHS nodes requiring a geometric expression. Jerboa enforces the embedding constraint by propagating the computed embedding values on partially matched orbits [2, 3]. For instance, if part of a vertex is translated, propagation ensures that all darts within the $\langle 1, 2\rangle$-orbit share the same position after modification. Only one position expression is needed (and thus has to be synthesized) per $\langle 1, 2\rangle$-orbit in the rule scheme. For the rule scheme of Fig. 3a, the RHS contains two $\langle 1, 2\rangle$-orbits, one with $n0$ and $n1$, and the other with $n2$, $n3$, and $n4$. The first orbit contains a preserved node ($n0$) associated with the darts $(a, b, \ldots, j)$ that have the same positions $(A, B, \ldots, E)$ in both input and output. We could add the expression `return n0.position`. No expression is needed in practice since the initial values can be propagated. For the second orbit, we must choose one of the nodes $n2$, $n3$, or $n4$ as a candidate for a geometric expression. In the sequel, we will consider the synthesis of a position expression for $n4$. Having identified the programs to be synthesized from the user-provided example, we now delimit the set of candidate expressions before validating them against all darts associated with the node.

4.2 Search Space

In the programming-by-example paradygm, program specifications are partial mappings corresponding to the input-to-output example(s). To mitigate overfitting when synthesizing programs, we constrain the search space to a subset of Jerboa's geometric expressions used as components in a sketching approach. Specifically, we focus on orbit-dependent built-in functions.

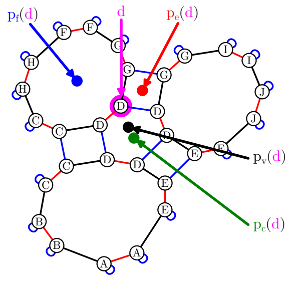
Component-based Synthesis Component-based synthesis generates programs from domain-specific primitives called *components* [17, 22, 28]. Because of the abstraction layer induced by rule schemes, we seek functions that do not depend on a specific instantiation, which we call *points of interest* (POI). Such functions should be independent of the (size of the) input orbit and remain invariant to changes in the dart chosen for computation. Orbit-based barycenters of positions satisfy these conditions, with a generic expression given by:

$$\texttt{middle}(\langle o\rangle\_\ \texttt{position}(x)), \tag{2}$$

generalizing $\texttt{middle}(\langle 0, 1\rangle\_\ \texttt{position}(x))$ (Sec. 3.2). Orbit types $\langle o\rangle$ induce an equivalence relation $\equiv_{\langle o\rangle} \subseteq N \times N$ between darts in the same $\langle o\rangle$-orbit, which extends to embeddings $\pi : \langle o_\pi\rangle \to \tau_\pi$ such that $\equiv_{\langle o_\pi\rangle}$-equivalent darts share the same τ_π-value [2, Def. 7]. For instance, $\texttt{position} : \langle 1, 2\rangle \to \texttt{Point3}$ partitions a 2-Gmap into vertices where all darts have the same `Point3`-value. This reduces the orbit-based barycenters of positions to four: vertices, edges, faces, and connected components, written $\text{poi}_{\langle o\rangle}$. The expressions are given in Table 7a and illustrated

Point of interest	Name	Expression
vertex	$p_v = \mathrm{poi}_{\langle 1,2\rangle}$	`middle(`$\langle 1,2\rangle$`_position(`d`))`
edge	$p_e = \mathrm{poi}_{\langle 0,2\rangle}$	`middle(`$\langle 0,2\rangle$`_position(`d`))`
face	$p_f = \mathrm{poi}_{\langle 0,1\rangle}$	`middle(`$\langle 0,1\rangle$`_position(`d`))`
component	$p_c = \mathrm{poi}_{\langle 0,1,2\rangle}$	`middle(`$\langle 0,1,2\rangle$`_position(`d`))`

(a)



(b)

Fig. 7. Points of interest: (a) geometric expressions, (b) examples of computations.

in Fig. 7b. Geometric expressions on RHS nodes refer to values of LHS nodes, similar to contract programming practices. Thus, the components are the POI $\mathrm{poi}_{\langle o\rangle}$ from one LHS node per $\langle o\rangle$-orbit ($n0$ for the flat extrusion). The choice of orbit-based barycenters as components meets the standard way in geometric modeling of using barycentric [10] and generalized barycentric coordinates [18] to interpolate or deform geometric objects from known positions [29]. Limiting POIs to orbit-based barycenters aligns well with the program-by-example setting and offers a minimal yet expressive set of components compatible with rule schemes: they abstract local geometry while remaining invariant under dart permutations.

Sketching POIs provide elementary built-in functions for synthesis. We complement them with a control structure akin to a sketch with holes for the numerical values to be retrieved, delimiting a search space of valid candidates. Here, we seek to retrieve affine combinations of POIs, similar to the approach of [48]. For a node n_R, the synthesized code defines a symbolic equation of the form:

$$\texttt{position}(n_R) = t + \sum_{\langle o\rangle \in 0..2/\sim_{\langle 1,2\rangle}} \sum_{n_L \in N_L/\equiv_{\langle o\rangle}} w_{\mathrm{poi}_{\langle o\rangle}(n_L)} \cdot \mathrm{poi}_{\langle o\rangle}(n_L) \quad (3)$$

where $0..2/\sim_{\langle 1,2\rangle}$ is the quotient of dimensions by the equivalent relation induced by $\langle 1,2\rangle$ (the rows of Table 7a), and $N_L/\equiv_{\langle o\rangle}$ the quotient of LHS nodes by the $\langle o\rangle$-induced relation. The weights $w_{\mathrm{poi}_{\langle o\rangle}(n_L)}$ and the translation t are the values to be synthesized, encoded by the keyword `??` in the sketch skeleton of Fig. 8. The sketch is the same for all RHS nodes. For $n4$ in the flat extrusion, it exactly matches Fig. 8 with `#node#` replaced by `n0`, specializing Equation (3) into:

$$\texttt{position}(n4) = t + \underbrace{w_v \cdot p_v(n0)}_{\text{vertex}} + \underbrace{w_e \cdot p_e(n0)}_{\text{edge}} + \underbrace{w_f \cdot p_f(n0)}_{\text{face}} + \underbrace{w_c \cdot p_c(n0)}_{\text{component}}. \quad (4)$$

4.3 Resolution via a Solver

During instantiation, geometric expressions are evaluated by substituting the node with each associated dart. Mimicking this process, we treat the sketch

```
 1  // translation
 2  Point3 res = new Point3( ?? , ?? , ?? );
 3
 4  // per #node# in <1,2>-orbit of the left hand side
 5  Point3 pV_#node# = Point3::middle(<1,2>_position( #node# ));
 6  pV_#node#.scaleVect( ?? );
 7  res.addVect(pV_#node#);
 8
 9  // per #node# in <0,2>-orbit of the left hand side
10  Point3 pE_#node# = Point3::middle(<0,2>_position( #node# ));
11  pE_#node#.scaleVect( ?? );
12  res.addVect(pE_#node#);
13
14  // per #node# in <0,1>-orbit of the left hand side
15  Point3 pF_#node# = Point3::middle(<0,1>_position( #node# ));
16  pF_#node#.scaleVect( ?? );
17  res.addVect(pF_#node#);
18
19  // per #node# in <0,1,2>-orbit of the left hand side
20  Point3 pC_#node# = Point3::middle(<0,1,2>_position( #node# ));
21  pC_#node#.scaleVect( ?? );
22  res.addVect(pC_#node#);
23
24  return res;
```

Fig. 8. Sketch skeleton for a geometric expression, where `??` encodes the values to be synthesized. Line 2 is a placeholder for the translation t. For each POI, the sketch operates in three steps: (1) the expressions `middle(<o>_position(#node#))` (lines 5, 10, 15, and 20) compute the POIs $\mathrm{poi}_{\langle o\rangle}(n_L)$, where `#node#` is replaced by one n_L (per equivalence class in $N_L/{\equiv_{\langle o\rangle}}$); (2) these `Point3` values are multiplied by weights $w_{\mathrm{poi}_{\langle o\rangle}(n_L)}$ via `scaleVect` (lines 6, 11, 16, and 21); (3) each POI's contribution is added to the global computation via `addVect`.

as a parametric equation with unknown weights (keyword `??`) and generate one constraint per associated dart. This transformation is valid because (1) the components (the POIs) only depend on input nodes, and (2) the output value is known from the user-provided example. We instantiate Equation (3) by substituting n_R and n_L with darts from the input-output example. For the flat extrusion, substituting $n0$ with a (Fig. 6b) and $n4$ with a_4 (Fig. 6c) in Equation (4) yields a constraint involving w_v, w_e, w_f, w_c, and t. The 10 darts from the pentagon of Fig. 6a yield 10 vector constraints, which we decompose into 30 scalar constraints (one per coordinate axis) over 4 weights and 3 translation components.

The constrained problem is a system of linear equations over real-valued variables and delegated to a solver. We experimented with Google OR-Tools with GLOP and Z3 with quantifier-free floating-point arithmetic. Z3 is a Satisfiability Modulo Theories (SMT) solver allowing soft constraints and prioritization, while GLOP is a simple-to-integrate linear programming solver. The results of Sec. 6 were obtained with OR-Tools as a backend. To break ties between valid solutions, we bias the solver toward simpler orbits – vertices, edges, faces, and finally, connected components – with cells preferred over the translation to avoid overfitting, similar to ranking or optimization methods [23]. For the flat extrusion, the solution is $w_v = 0.5$, $w_e = 0.0$, $w_f = 0.5$, $w_c = 0.0$, $t_x = 0.0$, $t_y = 0.0$, and $t_z = 0.0$. As postprocessing, we discard POIs with weight below an (adjustable) threshold of 10^{-3}, removing edge and connected component POIs in Fig. 9.

```
// no translation
Point3 res = new Point3(0.0,0.0,0.0);

// vertex
Point3 p0 = Point3::middle(<1,2>_position(n0));
p0.scaleVect(0.5);
res.addVect(p0);

// face
Point3 p2 = Point3::middle(<0,1>_position(n0));
p2.scaleVect(0.5);
res.addVect(p2);

return res;
```

Fig. 9. Synthesized geometric expression for $n4$.

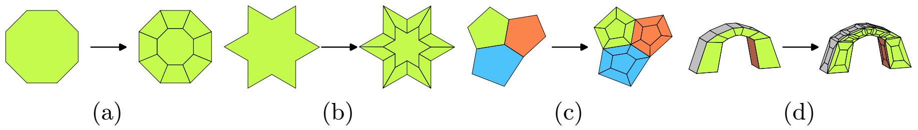


Fig. 10. Applications of the synthesized flat extrusion to other faces: (a) to an octogon, (b) to a snowflake. Applications of the synthesized operation generalized to surfaces: (c) to the geometric object of Fig. 4, (d) to an arche.

For the variations in Fig. 2, we obtain the same expression for Fig. 2a with the modified weights $w_v = 0.67$ and $w_f = 0.33$, while the additional vertices of Fig. 2c have $w_v = 0.5$ and $w_e = 0.5$. Our approach follows [16], splitting the synthesis task into *sketch generation* from components and *sketch completion* using constraint-solving, jointly encoding component structure, syntactic sketch restrictions, and user intent into a unified problem which guarantees syntactic and semantic correctness.

4.4 Application of the Synthesized Operation

As discussed in Sec. 2, our goal is to synthesize geometric expressions for modeling operations that generalize across shapes. Having synthesized the expressions for the flat extrusion (Fig. 1), we can apply it to other faces, such as the quad in Fig. 2b, producing the expected result. As shown in Figs. 10a and 10b, the synthesized operation can edit various shapes. We can also run the topological folding algorithm on the same example with the orbit type $\langle 0, 1, 2 \rangle$ to obtain a rule scheme similar to Fig. 3a with $n0$ carrying $\langle 0, 1, 2 \rangle$, and RHS nodes adjusted accordingly. The synthesized geometric expression remains unchanged, generalizing the flat extrusion to surfaces (Figs. 10c and 10d).

5 Implementation and Practical Details

5.1 Generalization to Vector Spaces and 3D Objects

Our synthesis method is independent of the embedding type and extends beyond positions. We enrich Jerboa's standard signature with a type `VectorialEbd` and

Table 1. Example of vectorial embeddings on a 2-Gmap.

Embedding	Data Type	Orbit Type	Array length	Clamp
`position`	`Point3`	$\langle 1, 2\rangle$	3	No
`color`	`Color3`	$\langle 0, 1\rangle$	3	Yes
`transparency`	`Transparency`	$\langle 0, 1\rangle$	1	Yes
`normal`	`Vector3`	$\langle 0, 1\rangle$	3	No

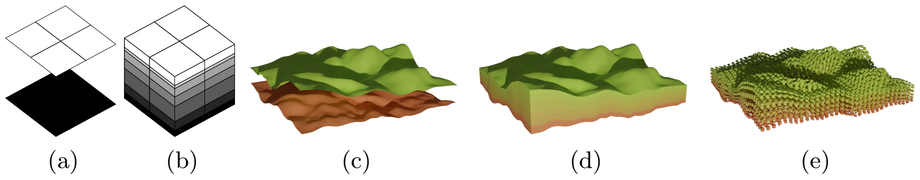


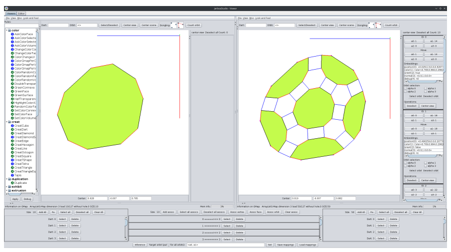
Fig. 11. Layering of subsoil horizons. Input (a) and output (b) data for synthesizing position and color expressions for the layering operation, with an application of the synthesized operation to a scene: (c) initial surfaces, (d) result of the layering operation, and (e) volume explosion hiding the initial surfaces to reveal inner volumes.

functions `scaleVect`, `addVect`, and `middle`. `VectorialEbd` serves as a generic interface, enabling generalization from barycentric *points* to *values* of interest. Subtypes of `VectorialEbd` are arrays of fixed length. Table 1 summarizes standard data types and associated embeddings for which our synthesis mechanism works. To deal with `Color3` and `Transparency` having bounded values, the result of the computation is clamped via the call to a function `clampVect` right before the return statement of line 24 in Fig. 8. Our approach generalizes to volumetric models (3-Gmaps). We demonstrate this on a layering operation used in geological modeling [43], where interpolated soil horizons are synthesized between two input surfaces (Fig. 11). The 3D operation involves both position and color embeddings, resp. defined on $\langle 1, 2, 3\rangle$- and $\langle 0, 1\rangle$-orbits. The system synthesized 25 expressions – 6 for positions, 19 for colors – via barycentric interpolation. Complete synthesis took under 55ms, which is negligible compared to the time needed to choose (or build) a representative input-to-output example or implement the operation.

5.2 JerboaStudio

Jerboa includes (1) an editor for designing operations, (2) a kernel for generating efficient code from rules, and (3) a generic Gmap viewer for applying operations. The editor allows specialists to create software that domain experts can use through the viewer. We integrated these components in a tool called JerboaStudio, available online[3] and illustrated in Fig. 12. In Fig. 12a, the user provides input and output shapes and hints for the input-to-output mapping. Fig. 12b displays the rule and a synthesized embedding expression. JerboaStudio offers a solution for the automatic construction of modeling operations combining program synthesis for geometric expressions and the topological folding algorithm from [41].

[3] Last consulted on July 16th, 2025 : https://gitlab.com/jerboateam/jerboa-studio


(a)

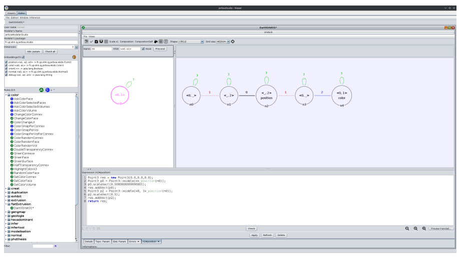

(b)

Fig. 12. JerboaStudio: (a) viewer with a pentagonal face as input (left), the extruded face as output(right), and the input-to-output mapping (bottom); and (b) editor with the retrieved rule scheme and a synthesized geometric expression (bottom).

5.3 Limits

Syntax-guided synthesis [1] may fail when the target expression lies outside the syntactic search space. Our search space consists of linear combinations of orbit barycenters, which can lead to overfitting or no solution being found, as in the example of the von Koch curve (Fig. 13). From a square (Fig. 13a), we construct the first iteration (Fig. 13b) and synthesize an operation. Applying it once provides the expected result, but reapplying it again yields an invalid second iteration (Fig. 13c): one expression (computed from edge midpoints) is correct; the other, involving a derivation angle from the original edge, is not. Our method computes the latter using the face's barycenter and translates the vertex in the wrong direction. A skilled user could edit the synthesized expression to fix it and obtain the desired second iteration (Fig. 13d). Figs. 14a and 14b show the input and output, with preservation links identified by the darts (a to h). The synthesized expressions (Fig. 14c) for $n3$ combine edge ($\mathrm{poi}_{\langle 0\rangle}$) and face ($\mathrm{poi}_{\langle 0,1\rangle}$) barycenters, explaining the deviation in Fig. 13c. This solution is found based on the regularity of the input square but would not be with subsequent iterations for which no expression exists within our syntactic space. The corrected version (Fig. 15) uses Jerboa features outside the original syntactic space.

6 Evaluation

We synthesized geometric expressions for subdivision schemes to validate our approach. These are standard operations used in modeling to refine shapes by applying changes to entire connected components. Table 2 reports results for 6 surface subdivisions, 2 volume subdivisions, and 2 surface-to-volume refinements. Each rule has a single LHS node. We use the same cube (48 darts) as a shared input instance for all operations leading to sketches with 8 unknowns solved on 48 equations. The experiments were conducted with Java 11, and OR-Tools 9.6.2534, on an Intel® Core™ Ultra 7, @4.80 GHz with 32 GB RAM. For each operation, we report the number of expressions to synthesize (# Expr), synthesized (# Synth), semantically correct after manual inspection (# Correct), solver time in ms

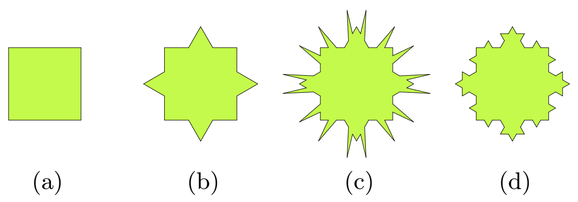
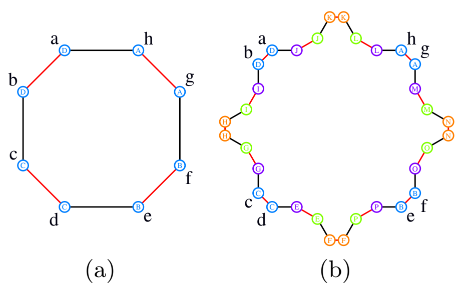


Fig. 13. Synthesis of the von Koch curve: (a) a square, (b) the first iteration of the operation, (c) invalid second iteration obtained by applying the synthesized operation, (d) valid second iteration built after fixing the geometric expression.



```
// n3
Point3 res =
 new Point3(0.0,0.0,0.0);
// edge
Point3 p1 = Point3::
 middle(<0>_position(n0));
p1.scaleVect(1.57735026919);
res.addVect(p1);
// face
Point3 p2 = Point3::
 middle(<0, 1>_position(n0));
p2.scaleVect(−0.57735026919);
res.addVect(p2);
//res
return res;
```

(c)

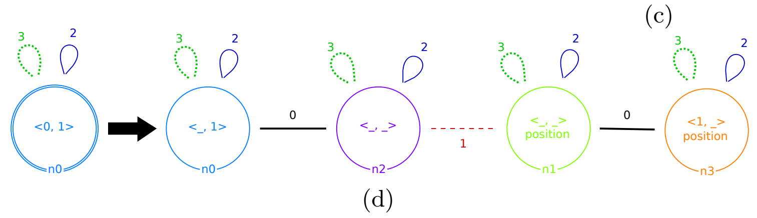


Fig. 14. Synthesizing the von Koch curve: (a) input and (b) output as a part of an embedded Gmap, (d) rule scheme deduced from the topological folding algorithm (node colors encode the retrieved associations), (c) synthesized position expression for $n3$.

```
Point3 src; Point3 tgt; Vector3 fNormal;
if (n0.orient){
  src = n0.position;
  tgt = n0@0.position;
  fNormal = Vector3::newellMethod(n0);
} else{
  src = n0@0.position;
  tgt = n0.position;
  fNormal = Vector3::newellMethod(n0@0);
}
Vector3 eVect = new Vector3(src,tgt);
Vector3 eNormal = eVect.cross(fNormal).normalize();
eNormal.scale(Point3::sqrt3 / 6 * eVect.norm());
eNormal.add(Point3::midpt(n0.position, n0@0.position));
return eNormal;
```

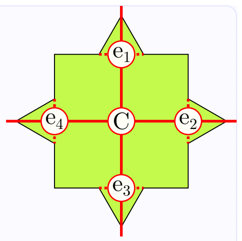


Fig. 15. Explanation for the synthesized expression of Fig.14c: fixing the expression for $n3$, requires expression outside the current search space as the new vertices are combinations of the face and edge barycenters.

Table 2. Benchmark Summary

Operation	# Expr	# Synth (%)	# Correct (%)	SolT (ms)	SynT (ms)
Surface Subdivisions					
Catmull-Clark [6]	3	3 (100%)	1 (33%)	1.2	10
Doo-Sabin [12]	1	1 (100%)	0 (0%)	0.4	11
Powell-Sabin [44]	2	2 (100%)	2 (100%)	0.9	11
Blender's Subdivide [19]	2	2 (100%)	2 (100%)	0.9	9
Sierpinski Carpet [35]	2	2 (100%)	2 (100%)	1.0	13
$\sqrt{3}$ [31]	2	2 (100%)	1 (50%)	0.9	11
Volume Subdivisions					
Menger [36]	3	3 (100%)	3 (100%)	1.4	26
$(2, 2, 2)$-Menger [46]	9	9 (100%)	9 (100%)	2.9	64
Surface to Volume Refinements					
Mesh to Tet	1	1 (100%)	1 (100%)	0.5	12
Mesh to Hex	3	3 (100%)	3 (100%)	1.2	16

(SolT), and complete synthesis time in ms (SynT). The artifacts for reproducing the benchmark are available as supplementary material [40] hosted on Zenodo. The solver handled each sketch in 0.3–0.5 ms. Full integration (from example to editor-ready operation) takes 0.5–2 s, making the pipeline responsive enough for interactive use. We correctly synthesized 24 out of 28 expressions, completing 7 of 10 operations, with failures due to expressions lying outside the search space. The method generalizes well to typical modeling tasks, while more expressive sketches (either in the control structure or components used) would be needed to address the remaining cases.

7 Conclusion

We presented a new method for automatically generating efficient code for geometric modeling operations with arbitrary topological changes. We use a rule-based DSL combined with an algebraically rooted language for geometric computations. Our component-based strategy uses a sketch as a control structure to define a search space of affine combinations over values of interest, completed by a solver from generated constraints. Future work includes interactive synthesis, where users can refine results through (counter)examples, posing challenges for user interaction, especially with 3D objects. Another promising direction is adapting Metasketches [5], which partitions the search space into ordered sets of sketches and guides the search with a gradient function.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis.

In: 2013 Formal Methods in Computer-Aided Design. pp. 1–8 (October 2013). https://doi.org/10.1109/FMCAD.2013.6679385
2. Arnould, A., Belhaouari, H., Bellet, T., Le Gall, P., Pascual, R.: Preserving consistency in geometric modeling with graph transformations. Mathematical Structures in Computer Science **32**(3), 300–347 (March 2022). https://doi.org/10.1017/S0960129522000226
3. Belhaouari, H., Arnould, A., Le Gall, P., Bellet, T.: Jerboa: A Graph Transformation Library for Topology-Based Geometric Modeling. In: Giese, H., König, B. (eds.) Graph Transformation. pp. 269–284 (2014). https://doi.org/10.1007/978-3-319-09108-2_18
4. Biermann, A.W.: The Inference of Regular LISP Programs from Examples. IEEE Transactions on Systems, Man, and Cybernetics **8**(8), 585–600 (1978). https://doi.org/10.1109/TSMC.1978.4310035
5. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 775–788 (January 2016). https://doi.org/10.1145/2837614.2837666
6. Catmull, E., Clark, J.: Recursively generated B-spline surfaces on arbitrary topological meshes. Computer-Aided Design **10**(6), 350–355 (November 1978). https://doi.org/10.1016/0010-4485(78)90110-0
7. Cheema, S., Buchanan, S., Gulwani, S., LaViola, J.J.: A practical framework for constructing structured drawings. In: Proceedings of the 19th international conference on Intelligent User Interfaces. pp. 311–316 (February 2014). https://doi.org/10.1145/2557500.2557522
8. Chugh, R., Hempel, B., Spradlin, M., Albers, J.: Programmatic and direct manipulation, together at last. ACM SIGPLAN Notices **51**(6), 341–354 (June 2016). https://doi.org/10.1145/2980983.2908103
9. Conlan, C.: The Blender Python API: Precision 3D Modeling and Add-on Development. Apress, 1 edn. (June 2017). https://doi.org/10.1007/978-1-4842-2802-9
10. Coxeter, H.S.M.: Introduction to Geometry. John Wiley (1969)
11. Damiand, G., Lienhardt, P.: Combinatorial Maps: Efficient Data Structures for Computer Graphics and Image Processing. CRC Press (September 2014)
12. Doo, D., Sabin, M.A.: Behaviour of recursive division surfaces near extraordinary points. Computer-Aided Design **10**(6), 356–360 (November 1978). https://doi.org/10.1016/0010-4485(78)90111-2
13. Du, T., Inala, J.P., Pu, Y., Spielberg, A., Schulz, A., Rus, D., Solar-Lezama, A., Matusik, W.: InverseCSG: automatic conversion of 3D models to CSG trees. ACM Transactions on Graphics **37**(6), 213:1–213:16 (December 2018). https://doi.org/10.1145/3272127.3275006
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag (2006). https://doi.org/10.1007/3-540-31188-2
15. Emilien, A., Vimont, U., Cani, M.P., Poulin, P., Benes, B.: WorldBrush: interactive example-based synthesis of procedural virtual worlds. ACM Transactions on Graphics **34**(4), 106:1–106:11 (July 2015). https://doi.org/10.1145/2766975
16. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex APIs. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 599–612 (January 2017). https://doi.org/10.1145/3009837.3009851
17. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. ACM SIGPLAN Notices **50**(6), 229–239 (June 2015). https://doi.org/10.1145/2813885.2737977

18. Floater, M.S.: Generalized barycentric coordinates and applications. Acta Numerica **24**, 161–214 (May 2015). https://doi.org/10.1017/S0962492914000129
19. Foundation, B.: Blender, https://www.blender.org/
20. Gould, D.: Complete Maya Programming: An Extensive Guide to MEL and C++ API. Elsevier, 1 edn. (2003). https://doi.org/10.1016/B978-1-55860-835-1.X5000-9
21. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM SIGPLAN Notices **46**(1), 317–330 (January 2011). https://doi.org/10.1145/1925844.1926423
22. Gulwani, S., Korthikanti, V.A., Tiwari, A.: Synthesizing Geometry Constructions. ACM SIGPLAN Notices **46**(6), 50–61 (June 2011). https://doi.org/10.1145/1993316.1993505
23. Gulwani, S., Polozov, O., Singh, R.: Program Synthesis. Foundations and Trends® in Programming Languages **4**(1-2), 1–119 (July 2017). https://doi.org/10.1561/2500000010
24. Guo, J., Jiang, H., Benes, B., Deussen, O., Zhang, X., Lischinski, D., Huang, H.: Inverse Procedural Modeling of Branching Structures by Inferring L-Systems. ACM Transactions on Graphics **39**(5), 155:1–155:13 (June 2020). https://doi.org/10.1145/3394105
25. Heckel, R., Taentzer, G.: Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-43916-3
26. Hempel, B., Chugh, R.: Semi-Automated SVG Programming via Direct Manipulation. In: Proceedings of the 29th Annual Symposium on User Interface Software and Technology. pp. 379–390 (October 2016). https://doi.org/10.1145/2984511.2984575
27. Hu, Y., Dorsey, J., Rushmeier, H.: A novel framework for inverse procedural texture modeling. ACM Transactions on Graphics **38**(6), 186:1–186:14 (November 2019). https://doi.org/10.1145/3355089.3356516
28. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. pp. 215–224 (May 2010). https://doi.org/10.1145/1806799.1806833
29. Ju, T., Schaefer, S., Warren, J.: Mean value coordinates for closed triangular meshes. ACM Trans. Graph. **24**(3), 561–566 (July 2005). https://doi.org/10.1145/1073204.1073229
30. Kania, K., Zięba, M., Kajdanowicz, T.: UCSG-NET- Unsupervised Discovering of Constructive Solid Geometry Tree. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. pp. 8776–8786 (December 2020)
31. Kobbelt, L.: sqrt(3)-subdivision. In: Proceedings of the 27th annual conference on Computer graphics and interactive techniques. pp. 103–112 (July 2000). https://doi.org/10.1145/344779.344835
32. Lieng, H., Kosinka, J., Shen, J., Dodgson, N.A.: A Colour Interpolation Scheme for Topologically Unrestricted Gradient Meshes. Computer Graphics Forum **36**(6), 112–121 (2017). https://doi.org/10.1111/cgf.12862
33. Lienhardt, P.: Topological models for boundary representation: a comparison with n-dimensional generalized maps. Computer-Aided Design **23**(11), 59–82 (1991). https://doi.org/10.1016/0010-4485(91)90100-B
34. Liu, H.T.D., Kim, V.G., Chaudhuri, S., Aigerman, N., Jacobson, A.: Neural subdivision. ACM Transactions on Graphics **39**(4), 124:124:1–124:124:16 (July 2020). https://doi.org/10.1145/3386569.3392418

35. Mandelbrot, B.B.: Fractals : form, chance, and dimension. Freeman (1977)
36. Menger, K.: Dimensionstheorie. B.G. Teubner (1928)
37. Merrell, P., Manocha, D.: Constraint-based model synthesis. In: 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling. pp. 101–111 (2009). https://doi.org/10.1145/1629255.1629269
38. Pascual, R.: Inference of graph transformation rules for the design of geometric modeling operations. PhD Thesis, Université Paris-Saclay (Nov 2022), https://www.theses.fr/2022UPAST146
39. Pascual, R.: Instantiation of Jerboa Rule Schemes, a Set-based Explanation (November 2024). https://doi.org/10.48550/arXiv.2411.15986
40. Pascual, R., Belhaouari, H.: Program Synthesis for Geometric Modeling - Benchmark Artifacts (Jul 2025). https://doi.org/10.5281/zenodo.15982906
41. Pascual, R., Belhaouari, H., Arnould, A., Le Gall, P.: Inferring topological operations on generalized maps: Application to subdivision schemes. Graphics and Visual Computing **6**, 200049 (May 2022). https://doi.org/10.1016/j.gvc.2022.200049
42. Pascual, R., Le Gall, P., Arnould, A., Belhaouari, H.: Topological consistency preservation with graph transformation schemes. Science of Computer Programming **214**, 102728 (February 2022). https://doi.org/10.1016/j.scico.2021.102728
43. Perrin, M., Rainaud, J.F.: Shared Earth Modeling: Knowledge Driven Solutions for Building and Managing Subsurface 3D Geological Models. Editions TECHNIP (2013)
44. Powell, M.J.D., Sabin, M.A.: Piecewise Quadratic Approximations on Triangles. ACM Transactions on Mathematical Software **3**(4), 316–325 (December 1977). https://doi.org/10.1145/355759.355761
45. Prusinkiewicz, P., Samavati, F., Smith, C., Karwowski, R.: L-system description of subdivision curves. International Journal of Shape Modeling **09**(01), 41–59 (June 2003). https://doi.org/10.1142/S0218654303000048
46. Richaume, L., Andres, E., Largeteau-Skapin, G., Zrour, R.: Unfolding Level 1 Menger Polycubes of Arbitrary Size With Help of Outer Faces. In: Couprie, M., Cousty, J., Kenmochi, Y., Mustafa, N. (eds.) Discrete Geometry for Computer Imagery. pp. 457–468 (2019). https://doi.org/10.1007/978-3-030-14085-4_36
47. Sharma, G., Goyal, R., Liu, D., Kalogerakis, E., Maji, S.: CSGNet: Neural Shape Parser for Constructive Solid Geometry. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 5515–5523 (2018). https://doi.org/10.48550/arXiv.1712.08290
48. Solar-Lezama, A.: Program sketching. International Journal on Software Tools for Technology Transfer **15**(5), 475–495 (October 2013). https://doi.org/10.1007/s10009-012-0249-7
49. Squillacote, A.H., Ahrens, J., Law, C., Geveci, B., Moreland, K., King, B.: The ParaView guide. Kitware (2007)
50. Summers, P.D.: A Methodology for LISP Program Construction from Examples. Journal of the ACM **24**(1), 161–175 (January 1977). https://doi.org/10.1145/321992.322002
51. Wu, F., Yan, D.M., Dong, W., Zhang, X., Wonka, P.: Inverse procedural modeling of facade layouts. ACM Transactions on Graphics **33**(4), 121:1–121:10 (July 2014). https://doi.org/10.1145/2601097.2601162
52. Št'ava, O., Beneš, B., Měch, R., Aliaga, D.G., Krištof, P.: Inverse Procedural Modeling by Automatic Generation of L-systems. Computer Graphics Forum **29**(2), 665–674 (2010). https://doi.org/10.1111/j.1467-8659.2009.01636.x