Interactive Optimization of Scaffolded Procedural Patterns

DAVIDE SFORZA*, Sapienza University of Rome, Italy
MARZIA RISO*, Sapienza University of Rome, Italy and INRIA, Université Côte d'Azur, France
FILIPPO MUZZINI, University of Modena and Reggio Emilia, Italy
NICOLA CAPODIECI, University of Modena and Reggio Emilia, Italy
FABIO PELLACINI, University of Modena and Reggio Emilia, Italy

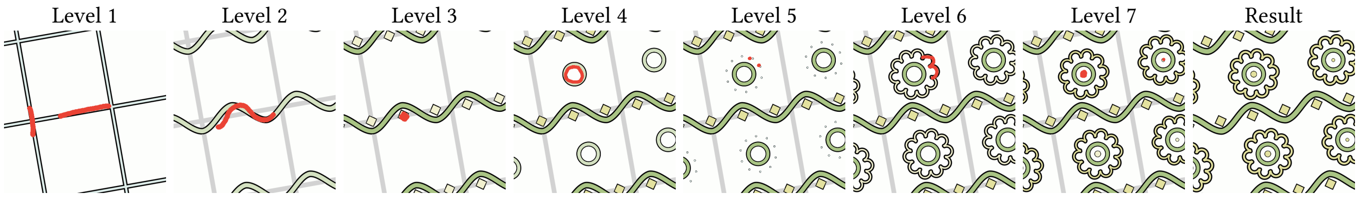

Fig. 1. We interactively edit procedural patterns by optimizing their parameters towards the best match with respect to hand-drawn sketches. We base our work on the use of *scaffolded procedural patterns*, that are sets of coarse-to-fine procedural functions, called *levels*, each of which is parametrized by a superset of the previous function's procedural parameters. Scaffolded patterns are edited level-by-level, allowing users to sketch in a coarse-to-fine manner rather than requiring them to sketch all details simultaneously. Also, since the optimization is split into multiple levels, the overall method is fast and stable, making it suitable for real-time sketching. Here, we show a pattern drawn in seven steps, using red to highlight user's strokes.

A procedural program is the representation of a family of assets that share the same structural or semantic properties, whose final appearance is determined by different parameter assignments. Identifying the parameter values that define a desired asset is usually a time-consuming operation, since it requires manually tuning parameters separately and in a non-intuitive manner. In the domain of procedural patterns, recent works focused on estimating parameter values to match a target render or sketch, using parameter optimization or inference via neural networks. However, these approaches are neither fast enough for interactive design nor precise enough to give direct control. In this work, we propose an interactive method for procedural parameter estimation based on the idea of scaffolded procedural patterns. A scaffolded procedural pattern is a sequence of procedural programs that model a pattern in a coarse-to-fine manner, in which the desired pattern appearance is reached step-by-step by inheriting previously optimized parameters. Through scaffolding, patterns are more straightforward to sketch for users and easier to optimize for most algorithms. In our implementation, patterns are represented as procedural signed distance functions whose parameters are estimated with a gradient-free optimization method that runs in real-time on the GPU. We show that scaffolded patterns can be created with a node-based interface familiar to artists. We validate our approach by creating and interactively editing several scaffolded patterns. We show the effectiveness of scaffolding through a user study, where scaffolding enhances both the output quality and the editing experience with respect to approaches that optimize the procedural parameters all at once. We also perform a comparison with previous strategies and provide several recordings of real-time editing sessions in the accompanying materials.

CCS Concepts: • **Computing methodologies → Texturing**; **Graphics systems and interfaces**.

*Joint first authors.

Authors' addresses: Davide Sforza, Sapienza University of Rome, Rome, Italy, sforza@di.uniroma1.it; Marzia Riso, Sapienza University of Rome, Rome, Italy and INRIA, Université Côte d'Azur, Sophia Antipolis, France, riso@di.uniroma1.it; Filippo Muzzini, University of Modena and Reggio Emilia, Modena, Italy, filippo.muzzini@unimore.it; Nicola Capodieci, University of Modena and Reggio Emilia, Modena, Italy, nicola.capodieci@unimore.it; Fabio Pellacini, University of Modena and Reggio Emilia, Modena, Italy, fabio.pellacini@unimore.it.

Additional Key Words and Phrases: Procedural modeling, Scaffolding.

1 INTRODUCTION

Procedural content creation is ubiquitous in computer graphics, given its ability to produce high-quality assets that can be easily edited to create countless variations. In procedural modeling workflows, users typically generate different assets by either developing new procedural programs or modifying the parameters of existing ones. The latter approach is by far the most common, given the widespread availability of extensive program libraries and the complexity of writing new procedural programs from scratch. However, as the number of parameters increases, the task of finding target values to achieve the desired look becomes increasingly cumbersome and time-consuming.

To mitigate this issue, recent works have focused on inverse procedural modeling techniques, aiming to automatically identify the optimal parameter set that matches a procedural model to a target asset. These methods differ in the type of user interaction they offer. Example-based techniques [Guo et al. 2020; Hu et al. 2022b; Riso et al. 2022] determine procedural parameters by optimizing their fit to a given exemplar, providing users with initial settings for further refinement. In contrast, direct manipulation methods [Gaillard et al. 2022; Michel and Boubekeur 2021; Pellacini 2010; Riso and Pellacini 2023] allow users to directly interact with the asset while an optimization process identifies the optimal procedural parameters to match the desired outcome. Additionally, parameter estimation via neural networks has been investigated [Hu et al. 2019; Trunz et al. 2024], leveraging machine learning algorithms to infer procedural parameters from input data. To this day, while methods based on neural networks are general, they are not precise in the parameter estimates. On the other hand, optimization-based methods are more specific to each procedural domain. Hence, they represent the state of the art in terms of parameter estimation accuracy.

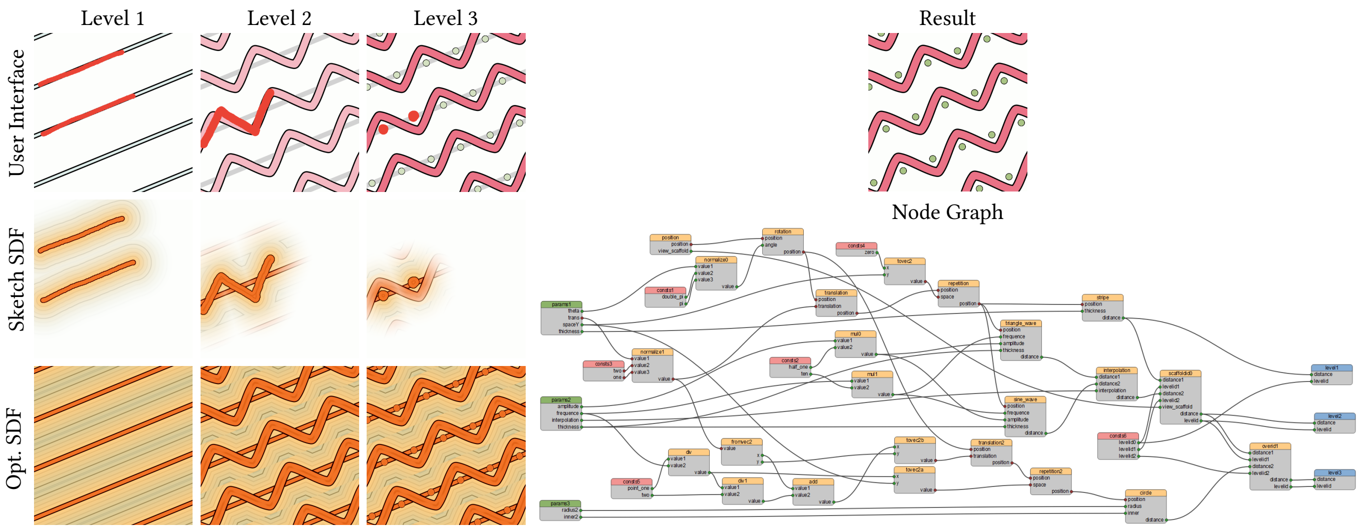

Fig. 2. *Scaffolded procedural patterns* are sequences of procedural functions, called *levels*, where the parameters of each function are a superset of the preceding ones. Left: We sketch scaffolded patterns by optimizing their procedural parameters, level-by-level, to match the user's sketches. We measure the difference between patterns and sketches using a loss function defined as a weighted distance between the respective Signed Distance Functions, since SDFs provide better fits than image-based metrics. At each level, the parameters of the preceding functions are fixed during optimization, making sketching easier and optimization faster and more robust. Right: We create scaffolded patterns using familiar node-graph interfaces, where, in most cases, existing internal nodes of a pattern work well as intermediate levels. In the graph, we show the levels' outputs as blue nodes and the levels' parameters as green nodes.

In this work, we focus on real-time parameter optimization for discrete element patterns, namely collections of shapes arranged in a desired layout determined by a procedural generator. We base our method on the idea of *scaffolded procedural patterns*. In computer graphics, scaffolding is used to guide the creation of digital assets, particularly in 2D and 3D sketches [Igarashi et al. 1999; Olsen et al. 2009; Schmidt et al. 2009]. This approach provides users with sets of guidelines that aid in the accurate drawing, modeling, or manipulation of complex shapes or scenes. Scaffolding techniques often involve the incorporation of auxiliary elements, such as construction lines or geometric primitives [Hähnlein et al. 2022; Hennessey et al. 2017; Yu et al. 2021]. They serve as guides for the user during the design process, offering tangible visual cues and spatial references.

We define a *scaffolded pattern* as a sequence of procedural functions, that we call *levels*, where the parameter set of each function is a superset of that of the previous function. While the idea of scaffolding is general, in this paper we focus on procedural patterns defined as Signed Distance Functions. In our optimization method, users edit the pattern level-by-level by sketching on a canvas. For each optimization step, a single procedural function is optimized, thus allowing users to specify coarse structural characteristics before moving into finer details. Scaffolded patterns can be created with the same node-based editors artists are familiar with, for example, by defining levels as the output of internal nodes, as shown in Fig. 2. This naturally generates a sequence of procedural programs, growing in complexity and inheriting parameters fine-tuned from simpler ones.

We estimate the parameters of the selected procedural program while the user is sketching the pattern, providing a real-time "autocompletion" framework that allows users to achieve the desired look in an interactive manner. The optimization, which is performed every time the user draws a new stroke, is based on a loss function that measures the distance between the Signed Distance Functions of the sketch and the target pattern. To achieve real-time optimization, we map the computation on the GPU and use a gradient-free method that converges faster and with lower overhead than gradient-based formulations for our problem domain.

Although our system delivers responsive interaction and accurate results across diverse procedural patterns, it also inherits some of the limitations typical of procedural modeling approaches. In particular, the expressiveness of the system is bounded by the procedural program and scaffolding sequence, both of which are fixed during optimization. As a result, users may sometimes sketch patterns that fall outside the representational capacity of the underlying program, leading to suboptimal fits. Additionally, authoring or modifying scaffolds requires programming, which may limit accessibility for non-technical users.

We evaluate our system by sketching different patterns, that are showcased in Figs. 1, 9, 10, and 11, and in the recordings of interactive sketching sessions provided as supplementary videos. All these patterns are defined as node graphs, that we include together with the corresponding generated code as supplemental material. We compare our optimization strategy to existing techniques that we found to be too computationally expensive for interactive feedback and run an ablation study to confirm our algorithm design. Finally, we discuss a user study where users found our approach to produce better patterns faster than optimizing the parameters all at once.

2 RELATED WORK

2.1 Inverse Procedural Modeling

With the term "inverse procedural modeling", we usually refer to the problem of finding a procedural representation of given target assets, such as textures, materials, and shapes. Due to the high diversity in assets, the literature in the field is quite copious. Depending on the application, some inverse procedural modeling approaches estimate the procedural program from the provided example, while others find the procedural parameters given known programs. In terms of program estimation, works have been proposed for inverse procedural architecture and urban models [Müller et al. 2007; Nishida et al. 2018, 2015, 2016; Stava et al. 2014], trees [Stava et al. 2010, 2014], knitwear [Trunz et al. 2019], motion data [Park et al. 2011] and materials [Guerrero et al. 2022]. In the reminder, we focus only on the works mostly related to our own, while we refer the reader to [Aliaga et al. 2016; Gieseke et al. 2021; Smelik et al. 2014] for comprehensive overviews.

In our work, we focus on determining the procedural parameters given known procedural programs via optimization. Parameter optimization can be performed either by direct manipulation or by example-based matching. In the former, transformations are directly performed on procedural assets with click-and-drag interactions, with the optimizer recomputing the parameter values that best fit the constraint expressed by the user. [Pellacini et al. 2007, 2002] are examples of the two methods applied to points lights, while [Loi et al. 2017], [Hempel et al. 2019] and [Riso and Pellacini 2023] focus on 2D vector graphics, offering script-directed, output-directed and click-and-drag manipulation tools, respectively. For 3D models, direct control of procedural meshes and SDFs can be obtained by adding differentiability to the procedural graphs [Michel and Boubekeur 2021; Riso et al. 2024], by transferring updates from a proxy shape [Gaillard et al. 2022] or enabling bi-directional editing [Cascaval et al. 2021].

An alternative to direct manipulation is example-based methods, where a user-provided target, usually in the form of an image, is used to compute the procedural parameters that best match the given image. Although the same problem has also been studied for other assets like procedural 3D shapes [Huang et al. 2017; Manfredi et al. 2023], we will focus on procedural materials, as they are closer to our domain. [Guo et al. 2020] estimate parameters with Markov Chain Monte Carlo, [Shi et al. 2020] provide building blocks to generate differentiable node graphs to optimize from photos, [Hu et al. 2019] estimate parameters with CNNs, [Hu et al. 2022b] proceduralize SVBRDFs via differentiable rendering, [Hu et al. 2022a] use differentiable proxies for node graphs, and [Trunz et al. 2024] uses an ensemble of feature-focused NNs for procedural yarn. The closest work to our own is [Riso et al. 2022], which proposes a differentiable signed distance definition for vector patterns, enabling parameter optimization by minimizing an SDF-based distance loss. Compared to these works, we focus on real-time parameter estimation so that procedural patterns can be sketched interactively.

2.2 Scaffolding

The term "sketch-based modeling" refers to methods that exploit freehand 2D curves to model 3D assets [Igarashi et al. 1999; Olsen et al. 2009]. Sketches enable rough visualization of a 3D shape, for which variations can be generated by interpolating 2D drawings [Arora et al. 2017] or hands-on drawing tutorials can be extracted to help users in perspective drawing [Hennessey et al. 2017]. A key concept in this field is scaffolding, which employs visual hints to guide the creation of complex shapes, thus enforcing alignments and proportions [Schmidt et al. 2007], detail refinements [De Paoli and Singh 2015], or inferring them from strokes [Schmidt et al. 2009].

Due to their geometric features, scaffolds provide hints to turn 2D sketches into coherent 3D models. While single-view reconstruction systems [Xu et al. 2014] rely on the most descriptive viewport only, multiview systems combine 2D drawings and 3D navigation tools to visualize and define curves while sketching [Bae et al. 2008; Nealen et al. 2007]. In recent VR drawing applications, scaffold lines define key points for shapes [Yu et al. 2021], which can be turned into unstructured 3D sketches by fitting piecewise smooth surfaces [Yu et al. 2022]. Recently, [Gryaditskaya et al. 2020] focused on lifting freehand drawings to 3D models, while [Li et al. 2022] reconstructed the CAD sequence that produces a shape from its sketch using a transformer-based stroke grouping network. Opposite to that, [Hähnlein et al. 2022] synthesize intermediate construct lines from 3D CAD sequences, exploiting geometrical properties to distinguish scaffold lines from clutter.

We take inspiration from these works and port the idea of scaffolding to the domain of parameter optimization for sketched procedural patterns. In our method, procedural patterns are optimized from coarse to fine, and visible scaffolds are sometimes used for both sketch guidance and layering of details, enabling better control of the overall pattern structure as well as finer pattern details.

3 METHOD

3.1 Sketching Procedural Patterns

The goal of our work is to determine the parameters of a procedural pattern as the user sketches the pattern interactively. This task can be formulated as an optimization problem with a loss function that measures the difference between the user-sketched pattern and the pattern produced by the procedural program whose parameters are being optimized. We focus on patterns composed of a collection of shapes in arbitrary arrangements. These patterns can be represented as Signed Distance Functions (SDFs) of the shapes' arrangements, a representation that was shown to be easier to optimize for, in the non-interactive setting, than using vector shapes directly [Riso et al. 2022].

More formally, we can say that a procedural pattern is a function $f_\alpha : \mathbb{R}^2 \rightarrow \mathbb{R}$ that maps a point $\mathbf{p} \in I$ in the image $I \subset \mathbb{R}^2$ to its signed distance to the pattern boundaries. Different patterns are obtained by changing the set of n procedural parameters α, that we normalize for simplicity in the unit domain, i.e. $\alpha \in [0, 1]^n$. The parameters α affect various shapes properties, such as their size and orientation, and their arrangement in the pattern, such as the spacing between rows and columns. They can be obtained by minimizing a loss function $\mathcal{L}$ that measures the difference between the pattern SDF f_α and the SDF s computed from the user's sketch. [Riso et al. 2022] suggest that the L_2 distance between SDFs works well in this problem domain as a loss function. The strokes' SDF is

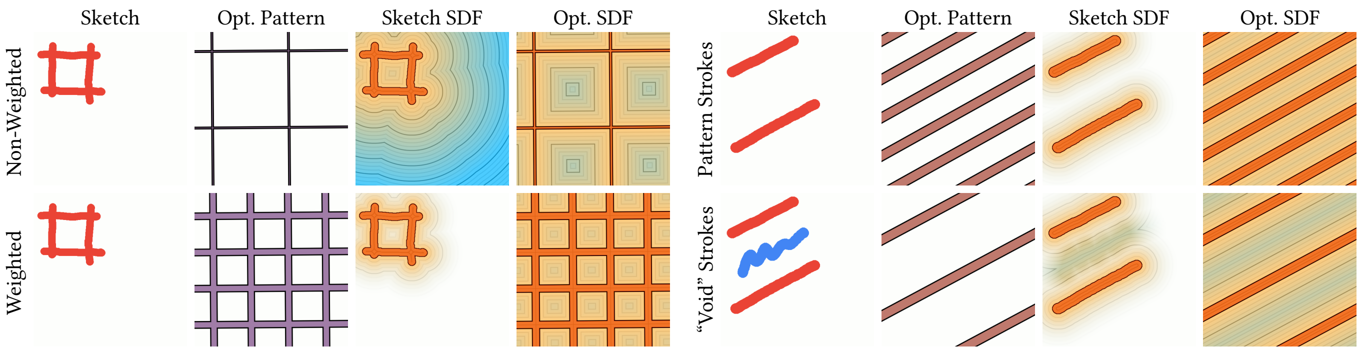

Fig. 3. Since patterns are sketched interactively, users' strokes rarely cover the whole canvas. Left: Optimizing strokes' SDFs directly often leads to failure since the non-drawn areas dominate the optimization. In our method, we weight the loss function near drawn strokes to focus on the relevant part of the canvas. Right: Besides strokes that sketch the pattern, shown in red, users can mark areas of the canvas to indicate empty spaces with "void" strokes, shown in blue.

computed as the Euclidean Distance Transform of the strokes and their inverse to obtain positive and negative distances. In short, we can write the optimization as

$$\alpha = \underset{\alpha}{\operatorname{argmin}}\ \mathcal{L}(f_\alpha, s),\ \text{ with } \mathcal{L}(f_\alpha, s) = \sum_{\mathbf{p}\in I} |f_\alpha(\mathbf{p}) - s(\mathbf{p})|^2 \quad (1)$$

Formalized this way, the optimization has two practical drawbacks. First, it requires users to sketch all pattern details at once since the loss is defined on the final pattern shapes. Second, since the optimization is non-linear, the possibly-high number of procedural parameters makes the optimization non-interactive and prone to get stuck on local minima. We address both issues by introducing the idea of scaffolding in procedural patterns.

3.2 Scaffolded Procedural Patterns

In free-hand drawing, scaffolds are a set of auxiliary lines, drawn in advance, that help the user maintain consistency in the pattern's structure. The main idea of our work is to introduce a similar mechanism when sketching discrete element patterns. We propose a new interface where users sketch a procedural pattern in a coarse-to-fine way by optimizing a sequence of procedural functions, that we call *levels*, of increasing details. The shapes' sketches at each level may or may not be visible in the final pattern, just like scaffolds in free-hand drawing.

More formally, we define a *scaffolded procedural pattern* as a predetermined sequence of K procedural functions $f^k_{\alpha^k}$ that are optimized sequentially, where the final level corresponds to the original function, i.e. $f^K_{\alpha^K} = f_\alpha$. For scaffolding to become effective in procedural modeling, the parameters of a function in the sequence need to be a superset of the parameters of the function that precedes it, i.e. $\alpha^k \supset \alpha^{k-1}$. In this manner, previously determined parameters can be used in subsequent optimizations. In terms of drawn shapes, subsequent levels may include or omit shapes from previous levels, similarly to scaffolds in free-hand drawing.

The process of sketching a scaffolded pattern is illustrated in Fig. 2. Scaffolded patterns are sketched and optimized level-by-level. At each level, a more detailed pattern is optimized, but only the parameters not present in the previous level need to be determined. In short, we can write the optimization of each level as

$$\tilde{\alpha}^k = \underset{\tilde{\alpha}^k}{\operatorname{argmin}}\ \mathcal{L}(f^k_{\alpha^k}, s^k),\ \text{ with } \tilde{\alpha}^k = \alpha^k \setminus \alpha^{k-1} \quad (2)$$

In our prototype user interface, the optimization of each level happens interactively after each stroke. The pattern corresponding to the current level is shown in the canvas below the sketch to help the user visualize the optimized parameter and possibly show additional scaffolds.

During parameter optimization, the levels of a scaffolded pattern are fixed, just like procedural programs are fixed when manipulating their parameters. In our prototype, procedural programs are created and edited as node graphs. Scaffolding is added to a graph by defining additional outputs, one for each level, as shown in Fig. 2. This makes scaffolding controllable for program authors, while still using a familiar user interface. We found that adding scaffolds to a well-parametrized graph is trivial, since procedural patterns are typically built in a coarse-to-fine fashion for development and subsequent use. From the annotated node graphs, we then generate code that computes all levels. In our prototype, this is done by generating a single function that computes all outputs, while relying on compiler optimizations for specialized programs. All graphs and the generated code are available as supplemental material.

In a departure from prior work, we want to support partially drawn canvases to make sketching faster and give user feedback while sketching. To do so, we weight the per-pixel SDF difference to only the sketched regions and their surrounding and include "void" strokes to mark regions that have no sketches but should remain empty, thus letting users remove unwanted shapes from the final pattern. The effect of weighting is illustrated in Fig. 3. If we indicate with w^k the weights for the k-th level, the weighted loss can be written as

$$\mathcal{L}(f^k_\alpha, s^k) = \sum_{\mathbf{p}\in I} w^k(\mathbf{p})|f^k_\alpha(\mathbf{p}) - s^k(\mathbf{p})|^2 \quad (3)$$

The weight w^k localizes the difference to only the sketched pixels, including with "void" strokes, and their surrounding. Since we already have the strokes SDFs, we can use those to define the weights. In particular, for each level k, we consider the SDF $s^k_a = s^k \cup s^k_v$ of all strokes, computed as the union between the pattern strokes SDF

Table 1. Comparison of optimization methods. For each pattern, we report the number of parameters, the number of algorithm iterations, the number of loss function calls, the error upon convergence, the time to converge, and the total time across all different initializations. Input sketches and fitted patterns in Fig. 4. Times in milliseconds.

Pattern		Gradient Descent (CPU)					Nelder-Mead (CPU)					Nelder-Mead (GPU)	
Fig.	Params.	Iterations	Func. Calls	Final Loss	Time	Total Time	Iterations	Func. Calls	Final Loss	Time	Total Time	Time	Total Time
4(a)	4	80	113	25.561	345	177963	67	96	25.563	60	14974	7	1968
4(b)	7	359	567	17.188	1998	549507	378	738	17.062	503	80468	36	6668
4(c)	8	283	504	23.835	2141	390692	286	204	23.716	558	93870	62	11537
4(d)	8	150	576	15.179	707	389294	202	314	15.179	191	54139	40	5470

s^k and the "void" strokes SDF s_v^k. The final weights are computed by modulating the SDF s_a^k with a clamped polynomial as

$$w^k(\mathbf{p}) = \begin{cases} w_s & \text{if } s_a^k(\mathbf{p}) \leq 0 \\ \left(1 - s_a^k(\mathbf{p})/w_d\right)^{w_e} & \text{if } 0 < s_a^k(\mathbf{p}) \leq w_d \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

where w_s is the weight given to the drawn areas, w_d is the distance from the stroke edges at which we ignore the SDFs values, and w_e is the polynomial exponent. We found empirically that the values $w_s = 4$, $w_d = 0.5$, and $w_e = 4$ work well for our application, where the higher weight for negative distances focuses the optimization on the drawn areas.

Our scaffolding formalization addresses both the aforementioned concerns in sketching procedurals. First, users can sketch a lot quicker. Patterns are sketched in a coarse-to-fine manner, thus not requiring drawing all shapes at once, since sketches may cover the canvas only partially. Second, the optimization is faster and more robust, since only a subset of parameters are optimized for each level and the user can guide the optimization of a complex pattern one level at a time. Both of these benefits are noticeable in the supplementary videos.

3.3 Gradient-Free Optimization

To make scaffolding useful in practice and to give users immediate feedback, we need to optimize the pattern parameters interactively. Previous literature relied on gradient-based algorithms for optimizing procedural pattern parameters [Gaillard et al. 2022; Michel and Boubekeur 2021; Riso et al. 2024; Riso and Pellacini 2023; Riso et al. 2022]. However, those methods are slow when applied to optimization formulations similar to ours. To meet our real-time requirement, we evaluated the convergence and execution of a gradient-descent optimizer and the Nelder-Mead's simplex algorithm [Nelder and Mead 1965], which operates without the need for derivatives. Gradient-free optimization methods can sometimes offer a viable alternative to gradient-based methods, especially in scenarios where the objective function lacks smoothness and has many local minima. Moreover, gradient-free methods can optimize discrete parameters, which is not possible with gradient-based methods.

To compare the two algorithms, we relied on their implementations available in the GSL library [Gough 2009]. In the gradient-based scenario, we computed derivatives with forward-mode automatic differentiation using TinyAD [Schmidt et al. 2022], a template-based library in which all gradient computations are inlined and

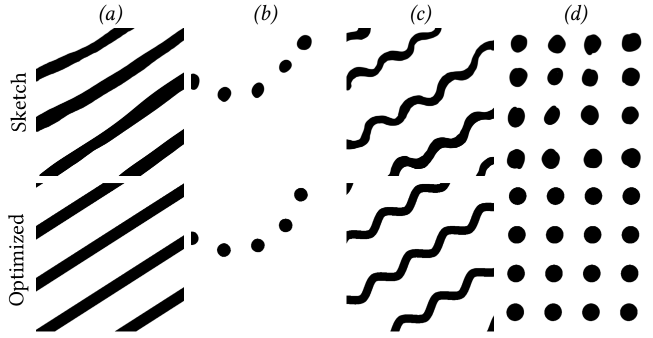

Fig. 4. We compare gradient-free and forward-mode gradient-based optimization by optimizing the parameters of four basic patterns that are often found in scaffolds. Both optimization methods converge well and are capable of replicating the sketched pattern, albeit gradient-free optimization is notably faster since it avoids the overhead of gradient computation.

optimized at compile time, thus avoiding large temporaries. For both methods, we parallelize computation on image rows on the CPU. We tested the two algorithms on a set of four different patterns, depicted in Fig. 4, simple enough not to require scaffolding. Each pattern underwent optimization from 250 distinct random parameter initializations, with a maximum of 250 iterations per optimization. Tests were carried out on a system equipped with a 32-cores AMD 5975WX CPU and 128GB of RAM. Table 1 summarizes the results of the experiment. Both algorithms effectively replicated the appearance of the patterns being optimized, but Nelder-Mead's simplex algorithm demonstrates notably faster execution speed, albeit with a higher iteration count. This efficiency of the simplex method can be attributed to its lack of overhead associated with gradient computation, thus making it a more suitable choice for our real-time scenario.

The evaluation of the pattern SDF and the final loss calculation are the most computationally intensive terms of the optimization process. These terms are executed multiple times during each Nelder-Mead's algorithm iteration. To achieve interactivity, we map these computations on the GPU. We use the CUDA programming model [NVIDIA et al. 2024] and exploit several concurrency levels that it makes available. We generate the CUDA programs for SDF evaluation automatically by compiling the node graphs to CUDA kernels. We evaluate the pattern SDF in parallel over image pixels, use optimized libraries to evaluate the loss [NVIDIA et al. 1999] and optimize the memory copies between CPU and GPU, since the memory traffic

Table 2. Number of parameters, number of scaffolding levels, error upon convergence, and average optimization time per user stroke for each pattern variant showcased in Figs. 1, 9, 10, 11 and 12. Times in milliseconds.

Pattern			Variant 1		Variant 2	
Fig.	Levels	Params.	Final Loss	Avg. Time	Final Loss	Avg. Time
1	7	24	2.993	749		
9(a)	2	7	7.775	139		
9(b)	2	8	27.909	195		
10(a)	3	10	5.667	164	2.369	164
10(b)	3	10	2.798	212	6.252	194
10(c)	3	13	6.639	213	3.723	195
10(d)	3	11	4.486	474	5.076	511
10(e)	3	10	8.503	209	29.916	204
10(f)	3	10	5.331	267	5.720	244
11(a)	4	15	4.519	302	5.604	328
11(b)	5	13	4.925	200	24.595	207
11(c)	5	13	8.656	259	4.525	269
12(a)	3	10	9.498	226		
12(b)	3	11	11.482	507		
12(c)	5	13	14.675	202		

between the processors only requires the exchange of procedural parameters and loss value at each iterations. In the end, combining the aforementioned practices, we achieve a speed-up of up to 12 times on an Nvidia 3090 Ti GPU with 24 GB of RAM, compared to the CPU implementation. Convergence times for patterns in Fig. 4 are reported in Tab. 1.

3.4 Limitations

Overall, we found our approach to perform well across a variety of patterns and to significantly improve upon the state of the art, as shown in the next section. However, a first limitation arises from the fact that users may sketch patterns that are not representable by the procedural program, which remains fixed during optimization. In such cases, the system can only approximate the intended design, often producing configurations that deviate from the user's input. This mismatch can be confusing for novice users. While this limitation is common to all approaches based on optimization over a fixed procedural space, it raises a usability concern: users may lack a clear understanding of what types of patterns are achievable with a given scaffold or how the scaffold constrains the design space. This complexity may lead to trial and error, particularly for novice users. A promising direction for future work would be to provide real-time guidance during sketching, such as visual cues based on the loss or predictive overlays, to help users stay within the expressive range of the current scaffold and better align input with output.

A second limitation concerns the scaffolding sequence itself, which, like the procedural program, is fixed and cannot be modified during interaction. While scaffolds play an essential role in structuring user input and guiding optimization, a poorly designed sequence can hinder the editing experience. For instance, if the sequence of editable elements does not align with the user's mental model of how the pattern should evolve, the interaction may feel awkward or confusing. This issue is intrinsic to procedural modeling,

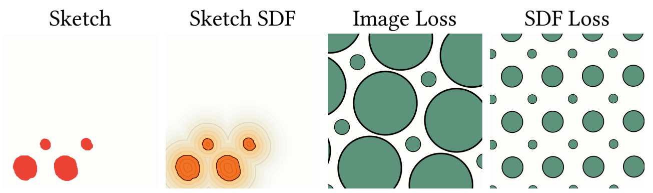

Fig. 5. Because SDFs are continuously defined on the whole canvas, a loss function based on SDFs is a more robust metric than an image-based distance when optimizing geometric patterns made of solid colors.

where editability depends heavily on how the procedural program is authored and parameterized. Modifying or extending scaffolds typically requires manual intervention by users with programming expertise, often involving direct manipulation of the procedural code. This creates a barrier to adoption, especially for users without a technical background, and limits the flexibility of the system. To address this issue, we plan to investigate data-driven alternatives, such as automatically extracting scaffolds from existing node graphs or annotated examples. These often include well-structured elements, like repeating motifs or explicit grids, that could serve as intuitive, user-friendly scaffolds without requiring manual coding.

4 RESULTS

4.1 Interactive Sketching

To validate our approach, we sketched several patterns in real-time editing sessions, shown in Figs. 9, 10, 11, and 12, and in the supplementary videos. All node graphs, and corresponding generated programs, are available as supplemental material. We chose procedural programs that encompass a variety of arrangements, such as square, radial, and hexagonal layouts, as well as different elements, such as circles, squares, and lines. Figure 9 illustrates how users can control the pattern by interactively sketching subsequent strokes at the same scaffold level. Figures 10 and 11 showcase two patterns generated from the same procedural program to demonstrate the flexibility of the programs used. Finally, Fig. 12 illustrates the use of "void" brush strokes to further enhance control. Due to the use of gradient-free optimization, our method scales well to detailed patterns, formed by many shapes, and supports programs with discrete parameters, such as the number of elements in the shown radial arrangements or the orientation of the tiles in Fig. 9(a).

Table 2 shows summary statistics of the real-time editing sessions. We optimized patterns with 7 to 24 optimizable parameters, arranged in 2 to 7 levels, depending on the complexity of the procedural program. During optimization, we run 15 optimization instances from different starting conditions to avoid local minima, letting the optimizer converge till a simplex's dimension threshold is reached or stops at at most 75 iterations. The optimization, which occurs every time the mouse is released, takes less than 800 milliseconds for all the procedural programs we tested, even for the most complex ones, ensuring the system remains interactive. This, in association with progressive refinement through successive scaffolds, consistently enabled us to achieve the desired look across all patterns we tested in just a few seconds. Without scaffolding, the patterns with high parameter counts would be too challenging if optimized as wholes, as demonstrated in the next sections.

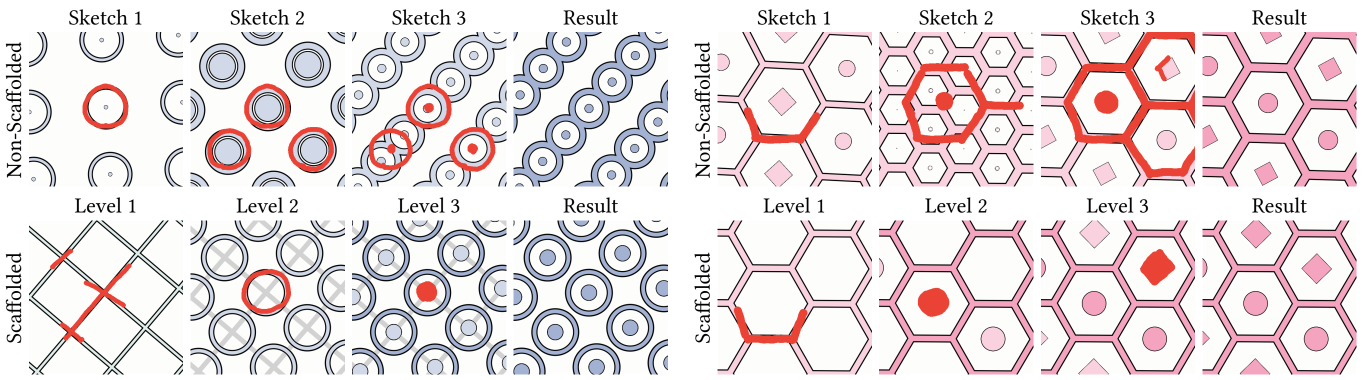

Fig. 6. Non-scaffolded patterns are challenging to optimize interactively since they require users to draw a more complete sketch to optimize all parameters at once. Scaffolded patterns help users sketch fewer elements and favor the optimization that deals with fewer parameters. Left: Example of non-scaffolded pattern that does not converge since many possible solutions would resemble the drawn sketch. Right: Example of non-scaffolded pattern that eventually converges to the right configuration, but produces unhelpful partial fits that do not match user intent.

4.2 Ablation Study

To understand the impact of different components of our framework, we performed an ablation study. Specifically, we examined the influence of using SDFs in the loss calculation and the effect of using the scaffolding mechanism, which are the key ingredients of our approach. Figure 5 illustrates the difference between defining the loss on the rendered pattern or on its SDF. The use of SDFs is more effective at comparing the target sketch and the generated patterns, since SDFs are defined over the entire image, not just the regions where elements are present. This is particularly useful for sparse patterns, where SDF differences capture the overall structure of the pattern, while image differences would not provide enough information to guide the optimization.

The impact of scaffolding is demonstrated in Fig. 6, where we compare the optimization done for all parameters at once to the one done in multiple steps using scaffolds. From a user perspective, optimizing the full set of parameters at once is more challenging since a sketch of the entire pattern is possibly required, while using scaffolding ensure that patterns can be sketched step-by-step by providing only the details added at each level. From a computational perspective, optimizing all parameters at once is hard since non-linear optimizers get stuck easily in local minima. Scaffolding improves significantly on this by reducing the number of parameters optimized at each level, as the previous level's ones are kept constant, letting users guide the optimization level-by-level to the desired outcome.

4.3 Comparison with Existing Techniques

The work most relevant to ours is [Riso et al. 2022], in which gradient-based optimization and backward-mode automatic differentiation are used to optimize procedural patterns. To compare the two optimization methods, we reproduced their honeycomb pattern, which takes 566 seconds to converge with their method and only 13.231 milliseconds with ours. We chose this pattern since it is simple enough to work without scaffolding, showing that our optimizer can achieve a speedup of more than forty thousand times in this case. The computational cost of their approach makes it unsuitable for interactive applications, as the user would have to wait minutes after each mouse stroke.

Furthermore, we replicated the pattern highlighted by [Riso et al. 2022] as a limitation of their method, where their algorithm failed to converge due to the high number of parameters involved. By using scaffolds, we successfully reproduced the pattern, as shown in Fig. 7. The use of scaffold lines let us guide the optimizer to the desired pattern, while the reduction of the number of parameters at each level makes the optimization more robust.

4.4 User Study

We performed a user study to assess the user-facing improvements in sketching procedural patterns with scaffolding. During the experiment, users sketched six procedural patterns, with and without scaffolding, to replicate a given target image. For the scaffolded version, we also provided an image of each level, which serves as documentation of the pattern structure. Before the experiment, we asked users to sketch six additional patterns, again with and without scaffolding, obtained from the same programs but with significantly different parameter configurations, to familiarize themselves with the task. We collected users' feedback in a questionnaire. After each editing task, we asked users to rate, on a scale from 1 (very poor) to 5 (very good), the final patterns quality and editing experience made with and without the use of scaffolds. Once completed all tasks, we asked users to choose between the scaffolded and non-scaffolded approaches, and provide some written feedback.

We recruited 20 subjects among PhD students in computer science and performed the experiment on a system equipped with a graphics tablet. We collect target images, initial configurations, and ratings for each pattern in the supplemental material. Figure 8 shows users' ratings for scaffolded and non-scaffolded approaches for both output pattern quality and editing experience. The scaffolded approach significantly outperforms the non-scaffolded one in both aspects. In fact, all users preferred the scaffolded approach while being asked for a direct preference at the end of the study. Users' comments confirm that they found scaffolded patterns easier to sketch, saying that the scaffolded approach is "easy to use" and that "the application tends to help me when providing my input". Conversely, some of

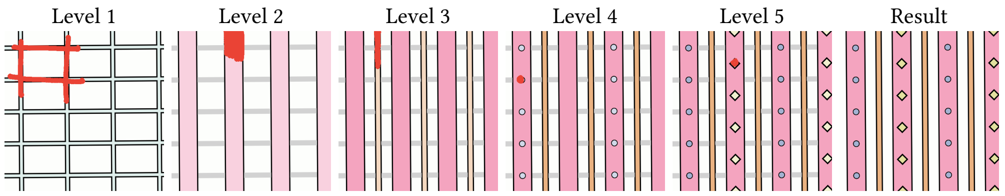

Fig. 7. The optimization of scaffolded patterns scales better with the number of parameters, since the optimization is guided by the user across levels. Here, we show a pattern that could not be optimized by [Riso et al. 2022], which represents the state-of-the-art for non-scaffolded parameter optimization.

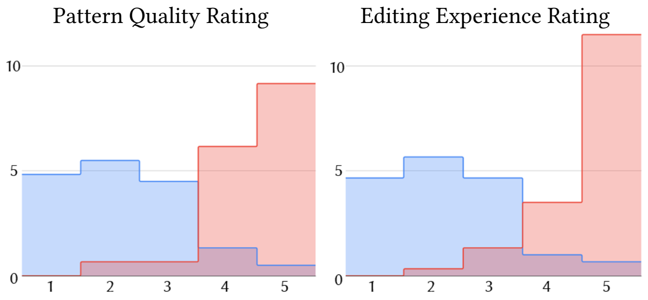

Fig. 8. Number of votes in terms of ratings for final pattern quality and editing experience, averaged over all patterns on a scale from 1 (very poor) to 5 (very good). In both cases, the scaffolded approach ratings, in red, are shifted towards higher values than the non-scaffolded approach ones, in blue.

them reported that "the other algorithm causes anxiety problem to the user", since it "is often frustrating as the feeling is that it is almost impossible to 'correct' or steer the output in the right direction".

5 CONCLUSIONS AND FUTURE WORK

In conclusion, we propose a scaffolded approach for real-time procedural pattern optimization. In our method, the parameters of procedural patterns are optimized level-by-level using an SDF-based loss and gradient-free optimization that runs interactively on the GPU, allowing users to interactively sketch patterns. We demonstrate that the use of scaffolding outperforms existing approaches in both ablation and user studies. In the future, we plan to investigate the possibility of automatically extracting scaffolds from existing procedural programs expressed as node graphs.

ACKNOWLEDGMENTS

This research was partially supported by the 2024 University of Modena and Reggio Emilia FAR fund and by the ANR-NSF NaturalCAD project (ANR-23-CE94-0003).

REFERENCES

Daniel G. Aliaga, İlke Demir, Bedrich Benes, and Michael Wand. 2016. Inverse Procedural Modeling of 3D Models for Virtual Worlds. In *ACM SIGGRAPH 2016 Courses (SIGGRAPH '16)*. ACM, Article 16, 316 pages. https://doi.org/10.1145/2897826.2927323

Rahul Arora, Ishan Darolia, Vinay Namboodiri, Karan Singh, and Adrien Bousseau. 2017. SketchSoup: Exploratory Ideation Using Design Sketches. *Computer Graphics Forum* 36 (02 2017). https://doi.org/10.1111/cgf.13081

Seok-Hyung Bae, Ravin Balakrishnan, and Karan Singh. 2008. ILoveSketch: as-natural-as-possible sketching system for creating 3d curve models *(UIST '08)*. 151–160. https://doi.org/10.1145/1449715.1449740

Dan Cascaval, Mira Shalah, Phillip Quinn, Rastislav Bodik, Maneesh Agrawala, and Adriana Schulz. 2021. Differentiable 3D CAD Programs for Bidirectional Editing.

Chris De Paoli and Karan Singh. 2015. SecondSkin: sketch-based construction of layered 3D models. *ACM Trans. Graph.* 34, 4, Article 126 (jul 2015), 10 pages. https://doi.org/10.1145/2766948

Mathieu Gaillard, Vojtech Krs, Giorgio Gori, Radomír Mech, and Bedrich Benes. 2022. Automatic Differentiable Procedural Modeling. *Computer Graphics Forum* (2022). https://doi.org/10.1111/cgf.14475

Lena Gieseke, Paul Asente, Radomir Mech, Bedrich Benes, and Martin Fuchs. 2021. A Survey of Control Mechanisms for Creative Pattern Generation. *Computer Graphics Forum* (2021). https://doi.org/10.1111/cgf.142658

Brian Gough. 2009. *GNU scientific library reference manual.* Network Theory Ltd.

Yulia Gryaditskaya, Felix Hähnlein, Chenxi Liu, Alla Sheffer, and Adrien Bousseau. 2020. Lifting freehand concept sketches into 3D. *ACM Trans. Graph.* 39, 6, Article 167 (nov 2020), 16 pages. https://doi.org/10.1145/3414685.3417851

Paul Guerrero, Milos Hasan, Kalyan Sunkavalli, Radomir Mech, Tamy Boubekeur, and Niloy Mitra. 2022. MatFormer: A Generative Model for Procedural Materials. *ACM Trans. Graph.* 41, 4, Article 46 (2022). https://doi.org/10.1145/3528223.3530173

Yu Guo, Miloš Hašan, Lingqi Yan, and Shuang Zhao. 2020. A Bayesian Inference Framework for Procedural Material Parameter Estimation. *Computer Graphics Forum* 39, 7 (2020).

Felix Hähnlein, Changjian Li, Niloy J. Mitra, and Adrien Bousseau. 2022. CAD2Sketch: Generating Concept Sketches from CAD Sequences. *ACM Trans. Graph.* 41, 6, Article 279 (nov 2022), 18 pages. https://doi.org/10.1145/3550454.3555488

Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, 281–292. https://doi.org/10.1145/3332165.3347925

James W. Hennessey, Han Liu, Holger Winnemöller, Mira Dontcheva, and Niloy J. Mitra. 2017. How2Sketch: generating easy-to-follow tutorials for sketching 3D objects. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '17)*. Article 8, 11 pages. https://doi.org/10.1145/3023368.3023371

Yiwei Hu, Julie Dorsey, and Holly Rushmeier. 2019. A novel framework for inverse procedural texture modeling. *ACM Trans. Graph.* 38, 6, Article 186 (nov 2019), 14 pages. https://doi.org/10.1145/3355089.3356516

Yiwei Hu, Paul Guerrero, Milos Hasan, Holly Rushmeier, and Valentin Deschaintre. 2022a. Node Graph Optimization Using Differentiable Proxies. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Proceedings.* ACM. https://doi.org/10.1145/3528233.3530733

Yiwei Hu, Chengan He, Valentin Deschaintre, Julie Dorsey, and Holly Rushmeier. 2022b. An Inverse Procedural Modeling Pipeline for SVBRDF Maps. *ACM Trans. Graph.* 41, 2, Article 18 (jan 2022), 17 pages. https://doi.org/10.1145/3502431

Haibin Huang, Evangelos Kalogerakis, Ersin Yumer, and Radomir Mech. 2017. Shape Synthesis from Sketches via Procedural Models and Convolutional Networks. *IEEE Transactions on Visualization and Computer Graphics* 23, 8 (2017), 2003–2013. https://doi.org/10.1109/TVCG.2016.2597830

Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. 1999. *Teddy: A Sketching Interface for 3D Freeform Design* (1 ed.). Association for Computing Machinery, New York, NY, USA.

Changjian Li, Hao Pan, Adrien Bousseau, and Niloy J. Mitra. 2022. Free2CAD: Parsing Freehand Drawings into CAD Commands. *ACM Trans. Graph. (Proceedings of SIGGRAPH 2022)* 41, 4 (2022), 93:1–93:16. https://doi.org/10.1145/3528223.3530133

Hugo Loi, Thomas Hurtut, Romain Vergne, and Joëlle Thollot. 2017. Programmable 2D Arrangements for Element Texture Design. *ACM Transactions on Graphics* 36, 3 (June 2017), Article No. 27. https://inria.hal.science/hal-01520258

Gilda Manfredi, Nicola Capece, Ugo Erra, and Monica Gruosso. 2023. TreeSketchNet: From Sketch to 3D Tree Parameters Generation. *ACM Trans. Intell. Syst. Technol.* 14, 3, Article 41 (March 2023), 29 pages. https://doi.org/10.1145/3579831

Elie Michel and Tamy Boubekeur. 2021. DAG Amendment for Inverse Control of Parametric Shapes. *ACM Transactions on Graphics* 40, 4 (2021), 173:1–173:14.

Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. 2007. Image-based procedural modeling of facades. *ACM Trans. Graph.* 26 (07 2007), 85. https://doi.org/10.1145/1275808.1276484

Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. 2007. FiberMesh: designing freeform surfaces with 3D curves. *ACM Trans. Graph.* 26, 3 (jul 2007),

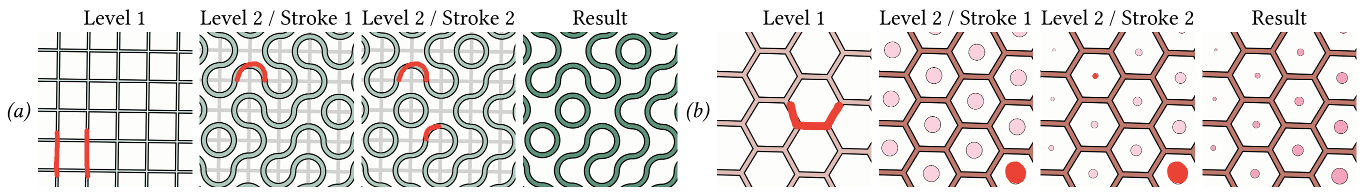

Fig. 9. Since our system is interactive, users can guide the optimization while sketching, shown here with two subsequent strokes (middle columns) for patterns named *(a)* and *(b)*.

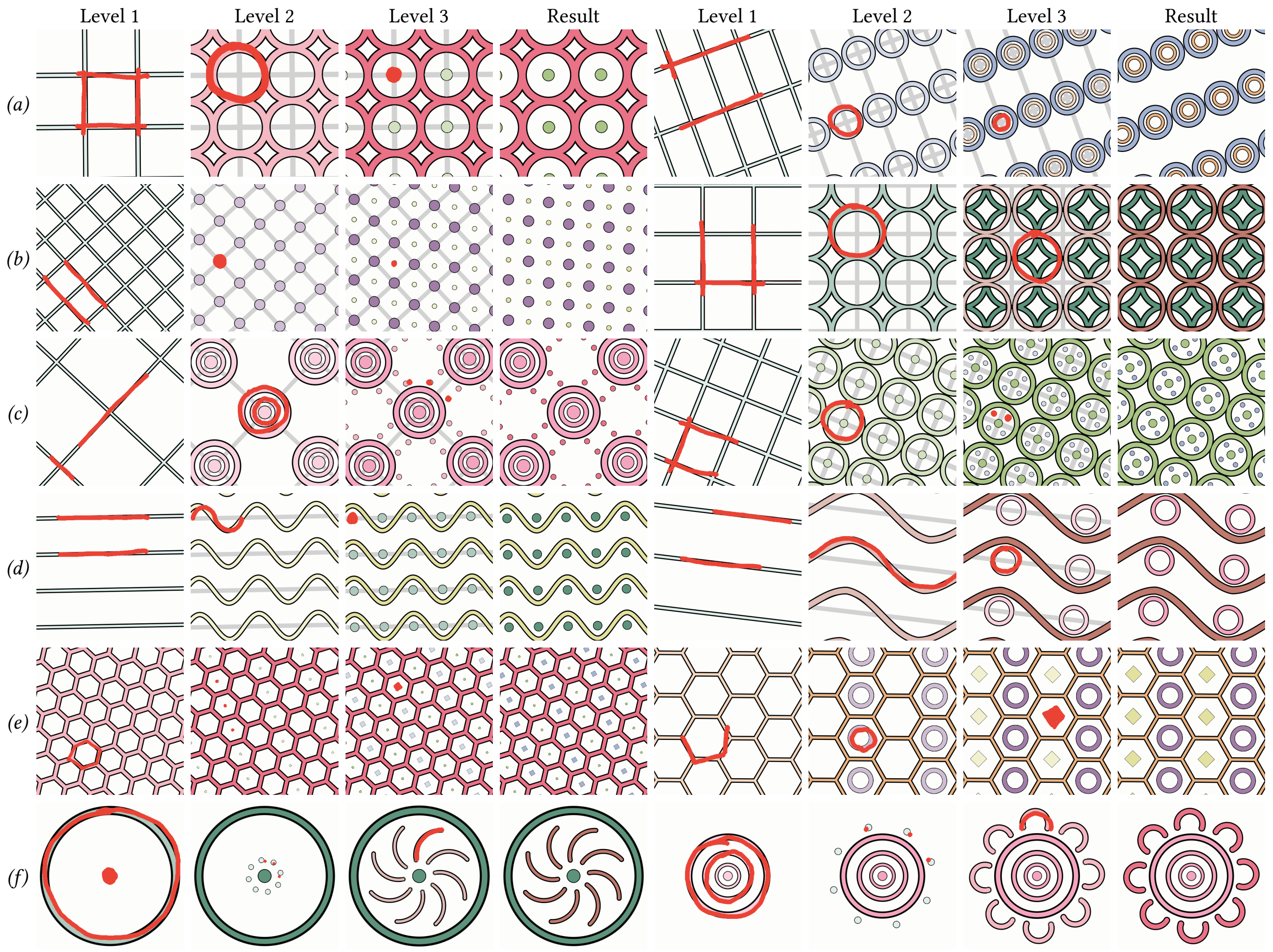

Fig. 10. We show the generality of our method by sketching various scaffolded procedural patterns indicated as *(a)* to *(f)*. For each pattern, we sketch two variants for each row, in the first four and the subsequent four images, to demonstrate how easily the final look is controlled using scaffolding.

41–es. https://doi.org/10.1145/1276377.1276429

John A. Nelder and Roger Mead. 1965. A Simplex Method for Function Minimization. *Comput. J.* 7 (1965), 308–313. https://api.semanticscholar.org/CorpusID:2208295

Gen Nishida, Adrien Bousseau, and Daniel G. Aliaga. 2018. Procedural Modeling of a Building from a Single Image. *Computer Graphics Forum* 37 (2018).

Gen Nishida, Ignacio Garcia-Dorado, and Daniel Aliaga. 2015. Example-Driven Procedural Urban Roads. *Computer Graphics Forum* 35 (09 2015). https://doi.org/10.1111/cgf.12728

Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. 2016. Interactive sketching of urban procedural models. *ACM Trans. Graph.* 35, 4, Article 130 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925951

NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 1999. *cuBLAS, release 10.2.* https://docs.nvidia.com/cuda/archive/10.2/cublas/index.html

NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2024. *CUDA, release: 12.4.* https://developer.nvidia.com/cuda-toolkit

Luke Olsen, Faramarz Samavati, Mario Costa Sousa, and Joaquim Jorge. 2009. Technical Section: Sketch-based modeling: A survey. *Computers and Graphics* 33 (02 2009), 85–103. https://doi.org/10.1016/j.cag.2008.09.013

Jong Pil Park, Kang Hoon Lee, and Jehee Lee. 2011. Finding Syntactic Structures from Human Motion Data. *Computer Graphics Forum* (2011). https://doi.org/10.1111/j.1467-8659.2011.01968.x

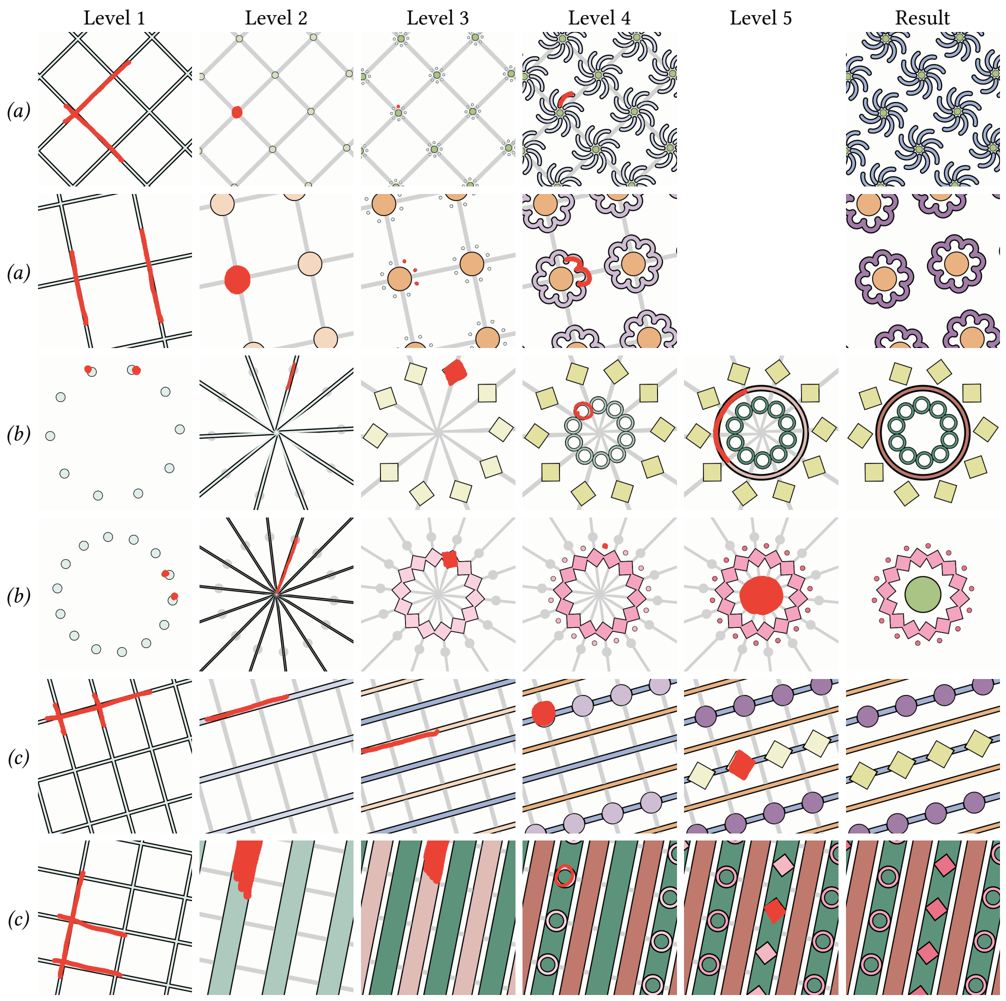

Fig. 11. Scaffolding makes sketching and optimizing complex patterns easier by increasing the number of scaffolding steps. Here, we sketch two variants, shown in subsequent rows, for more challenging procedural patterns named *(a)*, *(b)*, and *(c)*.

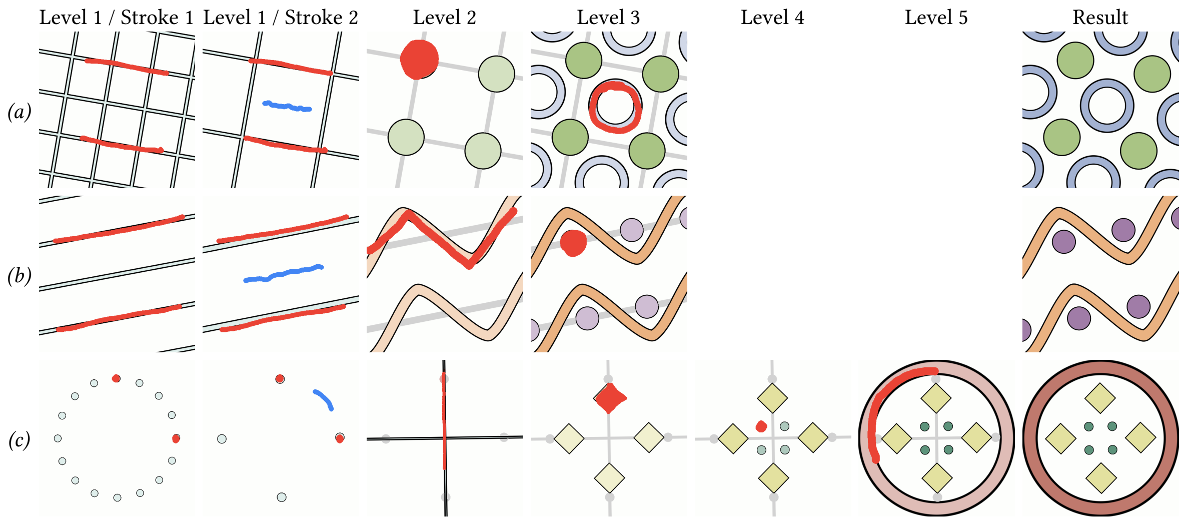

Fig. 12. Marking empty areas provides better control for patterns with coarser details. We highlight pattern strokes in red and marks to empty areas in blue.

Fabio Pellacini. 2010. envyLight: an interface for editing natural illumination. *ACM Trans. Graph.* 29, 4, Article 34 (jul 2010), 8 pages. https://doi.org/10.1145/1778765.1778771

Fabio Pellacini, Frank Battaglia, R. Keith Morley, and Adam Finkelstein. 2007. Lighting with paint. *ACM Trans. Graph.* 26, 2 (2007), 9–es. https://doi.org/10.1145/1243980.1243983

Fabio Pellacini, Parag Tole, and Donald Greenberg. 2002. A user interface for interactive cinematic shadow design. *ACM Trans. Graph.* 21 (07 2002), 563–566. https://doi.org/10.1145/566570.566617

Marzia Riso, Élie Michel, Axel Paris, Valentin Deschaintre, Mathieu Gaillard, and Fabio Pellacini. 2024. Direct Manipulation of Procedural Implicit Surfaces. *ACM Trans. Graph.* 43, 6, Article 238 (2024), 12 pages. https://doi.org/10.1145/3687936

Marzia Riso and Fabio Pellacini. 2023. pEt: Direct Manipulation of Differentiable Vector Patterns. In *Eurographics Symposium on Rendering*, Tobias Ritschel and Andrea Weidlich (Eds.). https://doi.org/10.2312/sr.20231126

Marzia Riso, Davide Sforza, and Fabio Pellacini. 2022. pOp: Parameter Optimization of Differentiable Vector Patterns. *Computer Graphics Forum* (2022). https://doi.org/10.1111/cgf.14595

Patrick Schmidt, Janis Born, David Bommes, Marcel Campen, and Leif Kobbelt. 2022. TinyAD: Automatic Differentiation in Geometry Processing Made Simple. *Computer Graphics Forum* 41, 5 (2022).

Ryan Schmidt, Tobias Isenberg, Pauline Jepp, Karan Singh, and Brian Wyvill. 2007. Sketching, scaffolding, and inking: a visual history for interactive 3D modeling. In *Proceedings of the 5th International Symposium on Non-Photorealistic Animation and Rendering (NPAR '07).* 23–32. https://doi.org/10.1145/1274871.1274875

Ryan Schmidt, Azam Khan, Karan Singh, and Gord Kurtenbach. 2009. Analytic drawing of 3D scaffolds. *ACM Trans. Graph.* 28, 5 (dec 2009), 1–10. https://doi.org/10.1145/1618452.1618495

Liang Shi, Beichen Li, Miloš Hašan, Kalyan Sunkavalli, Tamy Boubekeur, Radomir Mech, and Wojciech Matusik. 2020. Match: differentiable material graphs for procedural material capture. *ACM Trans. Graph.* 39, 6, Article 196 (nov 2020), 15 pages. https://doi.org/10.1145/3414685.3417781

Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. 2014. A Survey on Procedural Modelling for Virtual Worlds. *Comput. Graph. Forum* 33, 6 (sep 2014), 31–50. https://doi.org/10.1111/cgf.12276

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. https://doi.org/10.1111/j.1467-8659.2009.01636.x

Ondrej Stava, Sören Pirk, Julian Kratt, Baoquan Chen, Radomir Mech, Oliver Deussen, and Bedrich Benes. 2014. Inverse Procedural Modelling of Trees. *Computer Graphics Forum* 33 (03 2014). https://doi.org/10.1111/cgf.12282

Elena Trunz, Jonathan Klein, Jan Müller, Lukas Bode, Ralf Sarlette, Michael Weinmann, and Reinhard Klein. 2024. Neural inverse procedural modeling of knitting yarns from images. *Computers & Graphics* 118 (2024), 161–172. https://doi.org/10.1016/j.cag.2023.12.013

Elena Trunz, Sebastian Merzbach, Jonathan Klein, Thomas Schulze, Michael Weinmann, and Reinhard Klein. 2019. Inverse Procedural Modeling of Knitwear. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* 8622–8631. https://doi.org/10.1109/CVPR.2019.00883

Baoxuan Xu, William Chang, Alla Sheffer, Adrien Bousseau, James McCrae, and Karan Singh. 2014. True2Form: 3D curve networks from 2D sketches via selective regularization. *ACM Trans. Graph.* 33, 4, Article 131 (jul 2014), 13 pages. https://doi.org/10.1145/2601097.2601128

Emilie Yu, Rahul Arora, J. Andreas Bærentzen, Karan Singh, and Adrien Bousseau. 2022. Piecewise-smooth surface fitting onto unstructured 3D sketches. *ACM Trans. Graph.* 41, 4, Article 88 (jul 2022), 16 pages. https://doi.org/10.1145/3528223.3530100

Xue Yu, Stephen DiVerdi, Akshay Sharma, and Yotam Gingold. 2021. ScaffoldSketch: Accurate Industrial Design Drawing in VR. In *Proceedings of ACM Symposium on User Interface Software and Technology.* 372–384. https://doi.org/10.1145/3472749.3474756