Eurographics Symposium on Rendering (2023)
T. Ritschel and A. Weidlich (Editors)

pEt: Direct Manipulation of Differentiable Vector Patterns

Marzia Riso[1] and Fabio Pellacini[2]

[1]Sapienza University of Rome, Italy
[2]University of Modena and Reggio Emilia, Italy

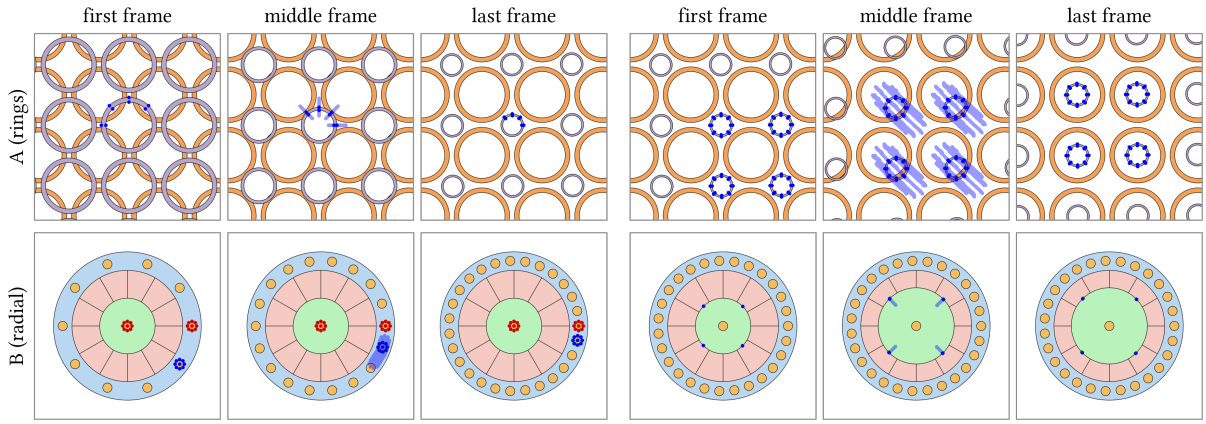

Figure 1: *We propose an interactive method to edit the parameters of procedural programs that generate vector patterns, where users interactively transform a set of points and constrain other ones to fixed locations. During the interaction, we solve for the procedural parameters with a gradient-based method since our patterns are differentiable with respect to the procedural parameters for both boundary and interior points. Here we show, for each pattern, the starting, middle and end frames of two sequential edits. Here, and in all figures, we mark in blue the transformed points, in red the fixed ones, and we draw the trajectory of the transformed points in blue in the middle frame.*

Abstract

Procedural assets are used in computer graphics applications since variations can be obtained by changing the parameters of the procedural programs. As the number of parameters increases, editing becomes cumbersome as users have to manually navigate a large space of choices. Many methods in the literature have been proposed to estimate parameters from example images, which works well for initial starting points. For precise edits, inverse manipulation approaches let users manipulate the output asset interactively, while the system determines the procedural parameters.

In this work, we focus on editing procedural vector patterns, which are collections of vector primitives generated by procedural programs. Recent work has shown how to estimate procedural parameters from example images and sketches, that we complement here by proposing a method for direct manipulation. In our work, users select and interactively transform a set of shape points, while also constraining other selected points. Our method then optimizes for the best pattern parameters using gradient-based optimization of the differentiable procedural functions. We support edits on large variety of patterns with different shapes, symmetries, continuous and discrete parameters, and with or without occlusions.

CCS Concepts

• ***Computing methodologies*** → ***Computer graphics;***

1. Introduction

Procedural methods are often used in computer graphics as they provide controllable, high-quality, and resolution-independent assets such as textures or shapes. Users generate different assets by either writing new procedural programs or by changing the parameters of existing ones. By far, the most common case is the lat-

DOI: 10.2312/sr.20231126

ter, considering that large libraries of programs are readily available [Ado22]. As the number of parameters increases, which is typical for high-quality programs, editing time grows significantly since users have to manually search a large parameter space to find the values that generate a desired asset.

Many recent works have investigated automated methods to estimate the parameters of procedural programs. These prior work differ in the user interaction they support. Example-based methods determine the procedural parameters that best match a given exemplar (see for example [HDR19, LLGRK20, SLH*20, RGF*20, HHD*21, HGH*22]), by formulating the problem as an optimization procedure. These methods are helpful to provide users with a starting point for further editing. Direct manipulation methods let users manipulate the asset directly, while an optimization procedure finds the best procedural parameters that match the edit. These methods differ in the manipulations they support; for example [MB21, CSQ*22] let users directly drag surface points for editing procedural meshes and CAD models, while [PTG02, Pel10] let users drag shadows and highlights to edit illumination. These direct manipulation methods have the advantage of letting the user guide the optimization interactively, while receiving real-time feedback on what the procedural program can achieve, sidestepping the issues that example-based methods exhibit when asked to reproduce an asset that the procedural program cannot achieve.

In this work, we propose *pEt*, a method for direct manipulation of procedural vector patterns. We consider patterns made of collections of vector graphics shapes generated by a procedural program. Editing such patterns by manipulating a slider-based interface remains cumbersome due to the high number of non-independent parameters. Inspired by the user interaction scheme of [MB21], we let users edit procedural vector patterns by interactively transforming user-selected points on the patterns' shapes (see Figure 1 for examples). We formulate the problem as determining, by optimization, the procedural parameters that minimize the distance between the user-transformed points and the corresponding pattern-computed points. The optimization runs interactively for each mouse event, letting users guide patterns toward the desired appearances. This lets users perform final edits in a goal-based manner, possibly starting from the pattern parameters determined by the example-based method of [RSP22].

Our formulation depends on two insights that make this work different from prior works in this area. A naive formulation would only let users transform all selected points, as is done in [MB21]. In the case of vector patterns, this is not sufficient to express complex pattern manipulations.

Instead, we also allow users to set constraints on some pattern points, inspired by similar ideas explored in goal-based illumination [PTG02]. Throughout the paper we show that this simple change is sufficient for users to guide the editing precisely (see Figure 2 for an example). Also, procedural vector patterns are often written imperatively as programs that issue shape drawing commands, such as emitting SVG shapes. In this representation, we cannot directly express our optimization constraints. Instead, we consider procedural vector patterns represented as functions that take as input the parametrized coordinates of each shape point, and output the point positions. This change of representation makes it possible to compute the gradients of the points positions with respect to the pattern parameters, making the pattern end-to-end differentiable.

We tested the method on a variety of patterns with different characteristics, shown throughout the paper. We consider edits that alter the shape positions, the pattern symmetries, the number of shapes and the deformation of the shapes, showing examples where we edit continuous and discrete parameters, as well as parameters are affected by noise functions. Overall we found *pEt* to work well for all cases.

2. Related Works

In this section, we review related works that use direct manipulation or example-based methods.

2.1. Direct Manipulation

The idea of direct manipulation has been previously explored in other domains, such as vector graphics, 3D models and rendering. One of the first examples of direct manipulation dates back to the work of [BB89], that proposes a system for the edit of Bézier curves by selecting a point where the curve should pass through, without directly editing its control points. [HLC19] proposes a bidirectional programming system for the creation of programs that generate vector graphics. In their interface, users can edit the program text or the output shapes, with the result mirrored in both modes.

A similar bidirectional approach is proposed in [CSQ*22] for 3D CAD. In this work, users directly manipulate the output shapes, while the system estimates the parameters of the program and maintains its validity. Inverse edits are performed by minimizing constrained optimization objectives that represent changes in geometry, deformation, program parameters as well as physical performance. [GKG*22] proposes a method for direct manipulation of 3d meshes using a bounding-box hierarchy. Upon selecting an object, the corresponding bounding-box is identified, and its vertices are transformed. The system minimizes the distance between the transformed points and the selected bounding-box vertices to estimate the procedural parameters. [GBLM16] proposes an editing approach that allows users to explore variation of the patterns as the user performs a manipulation. Although the space of possible variations is exponential, this work provides a tool that identifies a set of intuitive and distinct variations the user could choose from.

[IMH05] propose a system for the interactive manipulation of 2D shapes by moving mesh vertices as constrain handles, without requiring a predefined skeleton. Their system recomputes the remaining vertices position by updating the triangles rotation and scale, thus minimizing their distortion. [JBPS11] investigates the use of blending weights for 2D and 3D object deformation controlled by handle points or cages, ensuring a simpler design and easier user control. Further works in the Inverse Kinematic field are extensively discussed in the survey of [ALCS18].

The work that mostly inspired our own is [MB21], which demonstrated inverse control when editing 3D meshes generated by node graphs. This work focuses on amending the graph network to support automatic differentiation that is then used for parameter solving. On the other hand, our work differs since our programs are

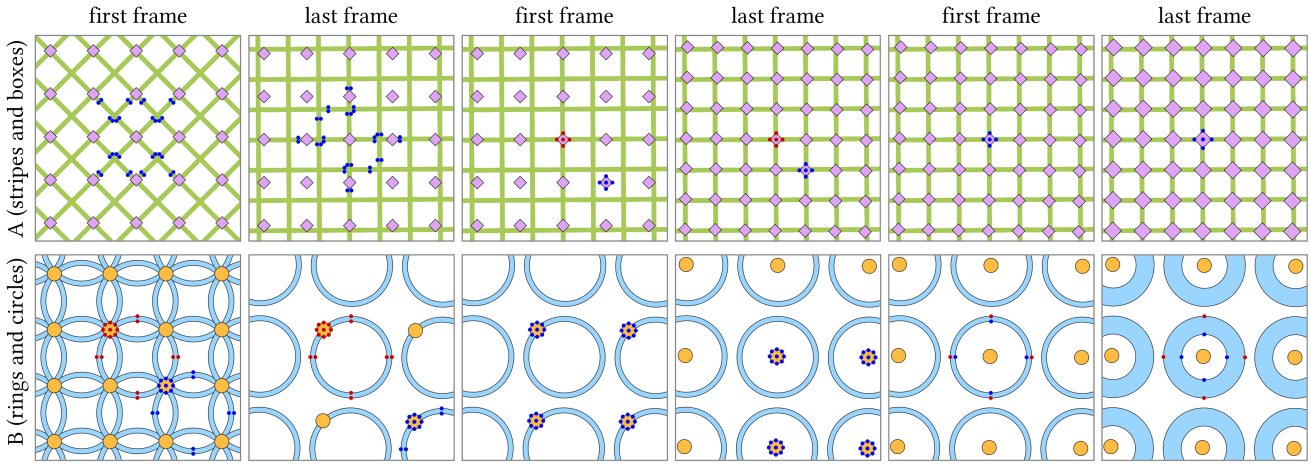

Figure 2: *The same procedural program can produce significantly different results by changing just the parameters assignment. Here two examples program are edited to produce three variations. For each variation, we show the first frame and the last frame of the optimization procedure.*

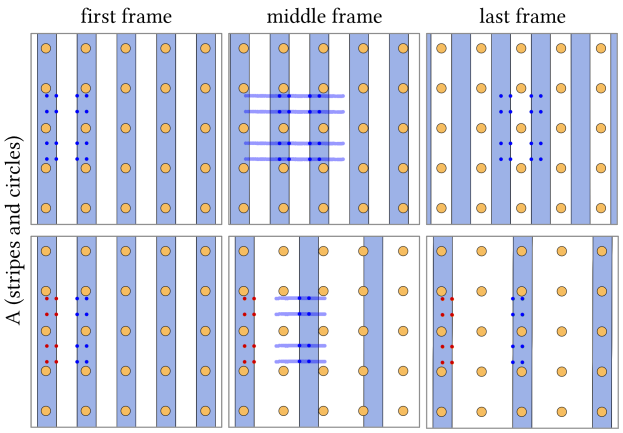

Figure 3: *Different edits can be expressed by selecting different points and either transforming or fixing them. (Top) The stripes are translated by transforming a set of 16 points. (Bottom) The distance between stripes is edited by transforming a set of 8 points, while fixing another 8 points.*

written in a general programming language, namely Python, and are automatically differentiated. Furthermore, we propose changes to the selection, including selecting points in the shapes' interior, which is not supported by their work.

In the rendering domain, [PTG02] proposes an interface for editing shadows by clicking and dragging them, while the algorithm determines the location of the corresponding point lights. The same work also introduces the idea of adding constraints to the edits. [Pel10] extends these ideas to the editing of environment maps. While both these works explore direct manipulation ideas, they do so without requiring an optimizer since it is possible to analytically compute light positions in the case of shadows and highlights. Further appearance, lighting and material editing approaches as well as how the combination of user interaction paradigms and rendering back ends provide a usable system for appearance editing are comprehensively analyzed in the survey of [SPN*14].

2.2. Example-Based Methods

In example-based methods, sometimes called inverse procedural methods, users provide an example as input while the algorithm determines the procedural program parameters or the structure of a procedural program that generate a similar output. This area of research has been heavily explored, so here we focus only on a few works. In general though, example-based methods are complementary to direct manipulation ones since the former works better as starting points during design, while the latter works best when performing final edits.

The works that are most closely connected to our own are the ones that estimate vector graphics parameters. [LLGRK20] propose a method for differentiating the rendered image of vector graphics primitives and uses it to fit input images. [RGF*20] show how to support compositing operations by proposing a method to differentiate them. [RSP22] propose a method for fitting vector graphics patterns to input images by differentiating the signed distance field of the vector primitives. Our work handles patterns similar to [RSP22], but supports interactive edits rather than offline optimizations.

Among other inverse design tasks, the ones involving textures are the most relevant to our work. [EL99, EF01] introduce non-parametric texture synthesis using greedy stochastic methods. [KEBK05] proposes an optimization method that refines the entire texture. [GEB15, GEB16] use convolutional neural networks for non-parametric textures, followed by [ZZB*18] that uses generative adversarial networks to reproduce textures non-stationary attributes. [BBT*06] and [HLT*09] aim at synthesizing patterns

by analyzing elements and properties from reference vector pattern provided by users. [IMIM08] also allow users to specify a local growth area as a constraint to the synthesis process. Similarly, [TWY*20] explores the stochastic synthesis of curve patterns. [MWLT13] propose a method for generating spatio-temporal repetitions based on a combination of a constrained optimization and a data-driven computation, while [ROM*15] explores repetitive structure synthesis using descrete elements or continuous geometries. A more comprehensive review of the example-based methods can be found in [GAM*21]. [GSH*20, HHG*22, ZHD*22] are recent examples of the many methods that stochastically synthesize realistic material maps guided by input images. These methods focus on non-parametric synthesis, while we concentrate on editing parametric patterns.

Recent work aims at editing materials expressed as parametric models. [GHYZ19] estimate graph parameters by exploring the parameters space with a Markov Chain Monte Carlo approach. [HDR19] use neural networks to select a procedural model from a library and estimate their procedural parameters, while [HHD*21] propose a semi-automatic pipeline for SVBRDFs proceduralization. [GHS*22] propose a generative model for procedural materials that are represented as node graphs, and let users to autocomplete those graphs too. [SLH*20] use differentiable material graphs to fit parameters to input images, while [HGH*22] make complex node differentiable by using neural network proxies.

Inverse procedural methods have been explored in many other areas of 3d graphics, as reviewed in [ADBW16]. These methods differ in the representation of the procedural program, and thus the optimization methods. [SBM*10, GJB*20] propose solutions to derive L-system parameters and rules from input images of vegetation. [SPK*14] estimate the parameters of tree generators, while [TMK*19] do the same for knitwear. [WYD*13, ZZBW15, NBA18] propose methods to define shape grammars for procedural buildings and facades taking as input example images.

3. Algorithm

We consider procedural programs that generate patterns made of collections of vector shapes, such as grids, stripes, radial patterns, optionally with occlusions and deformations of shapes. Users can edit procedural patterns by either changing the program code, or altering the program parameters. In this work, we focus on simplifying the latter task. The output of procedural programs can differ substantially by just changing its parameters, as shown in Figure 2, but as the number of parameters increases, the edits become very time-consuming. [RSP22] show how to estimate the procedural parameters to match an example image, which works well for an initial estimate, but that cannot be further edited with the same method. To fine-tune procedural patterns, we propose an interactive direct manipulation method where users select and transform a set of points on the patterns, while our algorithm solves for the procedural parameters interactively at each mouse event. Our method is general with respect to the pattern type and its parameters, and only requires the pattern to be differentiable, which we obtain using automatic differentiation. In the following sections, we will describe the method and motivate its design, starting from the user interaction.

3.1. User Edit

In *pEt*, users edit patterns by selecting arbitrary sets of points on the pattern shapes and transforming them. We consider sets of points, instead of single ones, since different selections correspond naturally to different edits, for the same points transformations (see Figure 3 for an example).

A possibility for implementing the interface would be to select and transform each set of points separately, as this offers the most control. However, this modality has two main issues. Since each transformation requires a separate mouse action, this would end in a non-interactive optimization of the pattern parameters, preventing users to naturally guide the edit by exploiting a real-time preview of the result. The second, and more important one, relates to optimization, whose interactive execution greatly enhances convergence, as shown in Figure 5.

Running the optimization for each mouse event means that the current procedural parameters are close to the optimal ones, so gradient-based optimization is more likely to find the optimal solution and not get stuck in local minima. Figure 8 shows the loss values throughout the offline optimization with reference to the online optimization ones, using the same program and selection already shown in Figure 5. The descending behavior is visible in both of them, although the online optimization always reached a lower convergence rate with reference to the offline one.

Due to this, we suggest a more straightforward user interface in which users select one set of points that will undergo the same transformation, as defined by mouse operations, and a second set of points, possibly empty, that will remain fixed throughout mouse interaction. In this interface, the transformed points guide the edit interactively, while the fixed points constrain it. This kind of interface has been proposed for interactive goal-based editing of illumination [PTG02, Pel10]. For pattern editing, using transformed and fixed point sets allows users to interactively express a wide variety of pattern transformations while maintaining a simple interface, as shown specifically in Figure 3 and in all figures in this paper.

We also prototyped an interface where users select and transform whole shapes at once, as opposed to points on them, since it feels natural for patterns made of rigid elements. However, we found that this interface is too constraining since it cannot express entire classes of valid pattern edits that we aim to provide. Furthermore, by selecting individual points we can also support patterns with procedurally deformed shapes, as shown in Figure 4.

In our prototype, users transform points by applying an affine transformation, namely translation, rotation or scaling, to the points original positions. While additional transformations can surely be implemented, we did not find it necessary for all the results in this paper. We should note that these transformations do not map directly to most of the parameters in the procedural patterns we tested.

In our current implementation, we compute the positions of the transformed points by re-transforming the starting points' locations according to the current mouse position. The optimizer is instead initialized with the procedural parameters' found in the last frame. This makes the optimization more stable and allows users to guide the edit where desired. We also tested a different approach where

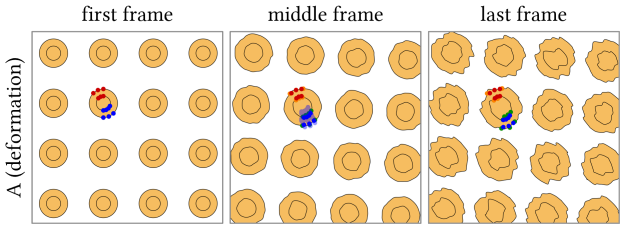

Figure 4: *Procedural deformations, here obtained by applying noise functions to the shape outlines, are controlled in the same manner as other transformations.*

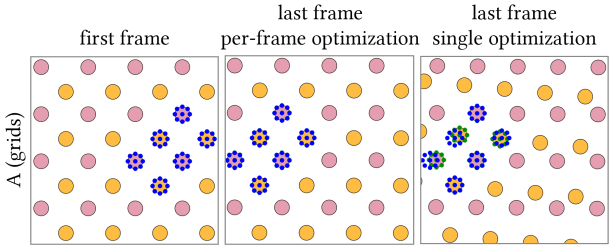

Figure 5: *Besides providing interactive feedback while editing, optimizing procedural parameters per-frame gives better final results, compared to optimizing only once at the end of the interaction sequence (we compare the results with the same total number of iterations).*

transformations are applied to the positions computed using the procedural parameters estimated in a previous optimization step of the edit, thinking that users might be able to better guide the optimizer. Instead, we found that the optimization suffers from drifting since tiny errors accumulate over time and do not cancel out during mouse interaction, as shown in Figure 6.

We can formally write the user interaction in our system as computing the transformed points $T^t p_s$ from the positions of the selected points p_s at the t-th frame of mouse interaction. The transformation T depends on the type of selected points, being a linear transformation M^t for the edited points $s \in E$, or the identity function for the fixed points $s \in F$. In summary, we can write

$$T^t p_s = \begin{cases} M^t p_s & s \in E \\ p_s & s \in F \end{cases} \tag{1}$$

3.2. Loss Function

We determine the procedural parameters by gradient-based optimization. In our formulation, we minimize the L_2 distance between the positions of the edited points and the positions of the same points computed by the procedural function, for each frame of mouse interaction. We treat transformed and fixed points in the same manner in the loss function.

A natural way to implement this loss is to consider points on the

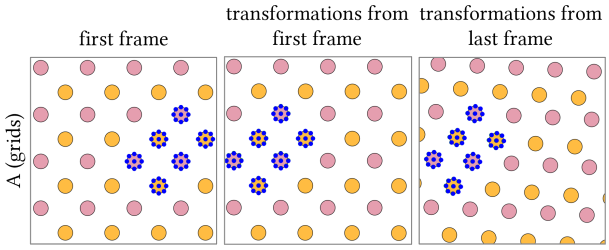

Figure 6: *In our prototype interface, we apply transformations starting always from the initial, correct configuration, since we observed drifting of the solution if the transformations are applied to the configuration of the last frame. In this example, the resulting pattern shows an undesired pattern rotation, as well as a misalignment due to the updated grid spacing.*

boundary of vector shapes since vector primitives are represented by their boundaries, e.g. vertices of polygons or points on tessellated splines. In fact, this is what is done in prior work on goal-based 3d shape editing [MB21], where users can only select the vertices of the boundary meshes. This makes the implementation trivial since boundary vertices are a finite set of uniquely identified items so they can be tracked by both the user interface and the procedural program without any additional work.

Our first prototype was implemented in this manner. But we quickly found that many valid edits cannot be expressed by editing only boundary points, for example as shown in Figure 7. For this reason, we extended the selection to also consider points in the interior of each vector shape.

This slight modification requires significant changes in the evaluation of the loss function since interior points are not uniquely identified, which is necessary to ensure that both user interface and optimizer track the same points. In our prototype, we require the procedural pattern to be able to evaluate the position of all points in each shape, both boundary and interior. We identify points with a shape identifier, which is uniquely defined, and by a parametrization of the shape interior, which allows us to identify all shape points. In our implementation, the user interface determines both shape identifiers and points parameters during selection. We can then freely transform the points by acting only on their location, while the optimization uses the fixed point identifiers to compute the location of the corresponding points.

To put things formally, we model procedural vector patterns as functions $f(i,\Theta)$ that take as input a point identifier i and compute the point location $\tilde{p}$ in the pattern. The point identifier $i = (d,u,v)$ is comprised of a discrete shape identifier d together with two continuous coordinates (u,v) that identify points in the interior and boundary of the shape d. The procedural pattern depends on the procedural parameters $\Theta = \{\theta_k\}$, which are the parameters that we optimize for. With this notation, we can write the optimization we perform at each frame t as

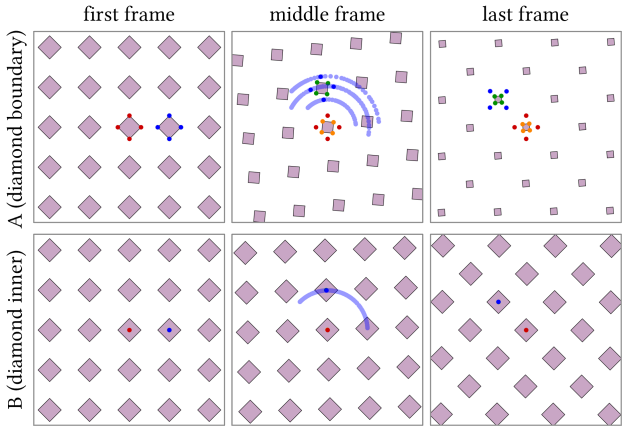

Figure 7: *In this example, the user intent is to rotate the grid without rotating the individual shapes. To do so, the user needs to fix some points to disambiguate between rotation and say translation and scaling during interaction. (Top) Selecting only points on the shape boundary induces unwanted shape transformations. In fact, as shown by the orange and green points, the user intent is not respected in this case. (Bottom) On the contrary, by selecting points in the interior of the shape, a rotation of the grid is specified precisely, and user intent is fully respected.*

$$\Theta^t = \text{argmin}_\Theta \frac{1}{N} \sum_s^N ||T^t p_s - f(i_s, \Theta^t)||^2 \quad (2)$$

$$\text{where } p_s = f(i_s, \Theta^t) \text{ for t = 0} \quad (3)$$

3.3. Optimization

We minimize the previously defined loss using gradient descent, relying on the automatic differentiation computation of the gradient of the procedural functions f with reference to its parameters. By optimizing procedural parameters at each frame, the per-frame optimization converges in few iterations since the procedural parameters Θ^t at frame t are used as initialization when computing the parameters Θ^{t+1} for the next frame $t+1$. In this manner, gradient descent finds the optimal solution with a small number of iterations, likely without incurring in local minima.

The program function f performs a vectorized computation of the points position if a packed vectorized parametrization is provided, thus improving the system speed. In the subsequent section, we will cover the main aspects of our editing tool.

We implemented our prototype using PyTorch since it provides robust automatic differentiation for our patterns. We optimize procedural parameters using the Adam optimizer [DB15] with a learning rate of 0.002. We use a maximum of 125 iterations, but we allow the optimization to stop sooner if the loss is below a threshold of 0.0005, corresponding to a negligible distance between the sets of points.

Our formulation scales trivially to complex patterns since it depends on the selection size and not the number of shapes in the pattern itself. We further improve speed by vectorizing the evaluation of the procedural patterns, to ensure that we evaluate the function for all points at once.

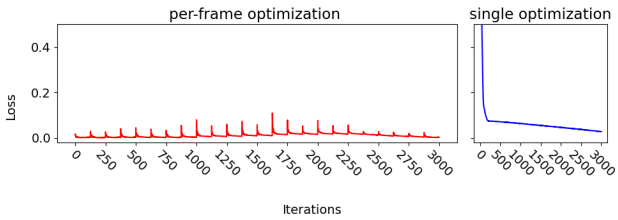

Figure 8: *Comparison between per-frame optimization, shown left, and optimization performed only in the last frame, shown right, for the edit in Figure 5. Besides providing feedback while editing, per-frame optimization has a lower end loss (0.0016) than end-only optimization (0.0327), for the total number of iterations. Peaks in the per-frame optimization correspond to mouse events.*

We support both continuous and discrete pattern parameters. Continuous parameters are left unchanged during optimization, and clamped to their valid range once the optimization terminates in each frame of mouse interaction. Discrete parameters are treated as continuous during optimization, and rounded to their discrete values at the end of each frame. The discrete behavior is implemented in the procedural function itself that rounds of the continuous optimization parameter to the internally discrete one. This rounding does not cause any trouble since shape identifiers remain unique, thus the selected ones remain uniquely identified. We support discrete parameters for which a continuous counterpart is well defined, such as the number of elements in the rings of Figure 1 (B) or the number of elements and subdivisions in Figure 9 (H). On the contrary, discrete parameters such as enums are not handled.

We should also note that occlusion between shapes does not cause any concern during the optimization since points are uniquely identified and the procedural function can compute the position of any parametrized point, whether or not these points are visible in the final rendering. The only implementation detail needed is to support the selection of hidden points in the user interface, which can be done in a manner similar to vertex selection in 3D software.

4. Results

In this section, we collect the results obtained while editing a variety of procedural vector patterns, summarized in Table 1 and shown in all the figures of the paper. We performed all the tests on a machine with an AMD Ryzen 9 CPU with 3.4 GHz frequency. The edit sequences discussed here, and displayed in the supplemental video, were re-computed offline from the original mouse interactions, for reproducibility and for further comparisons with the synthetic tests presented later. On our machine, our implementation reaches in a range of 0.9 ms to 2.8 ms per iteration. In our tests, time increases with the number of points selected, but remains constant throughout all iterations of an optimization step.

We tested patterns with a growing number of parameters from 9

Table 1: *Statistics of the edits shown throughout the paper. For each edit we report the number of parameters and shapes of the pattern, the number of transformed and fixed points, the number of mouse events of the stroke, and the number of iterations performed during the optimization (mean value, minimum and maximum early exit iteration). For real edits, we report the loss at the end of the optimization, while for synthetic tests we report the MSE between the target parameters and the correct values. Each table row corresponds to a different program, where more than an entry is reported when a sequence of edits is performed consecutively on the same pattern.*

Figure Number	Num. Params.	Num. Shapes	Transf. Points	Fixed Points	Mouse Events	Number of Iterations	Final Loss	Average MSE
Fig. 7 B	9	25	1	1	42	25, 6, 42	0.0013	$2 \cdot 10^{-6}$
Fig. 9 A	10	121 | 98	3 | 6	1 | 1	27 | 25	57, 24,125 | 76, 21,125	0.0019 | 0.0013	$1 \cdot 10^{-6}$ | $4 \cdot 10^{-5}$
Fig. 9 E	10	49 | 84	3 | 5	3 | 1	42 | 28	53, 23, 87 | 97, 64,125	0.0020 | 0.0023	$2 \cdot 10^{-4}$ | $6 \cdot 10^{-4}$
Fig. 9 F	10	49 | 105	5 | 10	1 | 8	5 | 5	102, 20,125 | 111, 42,125	0.0083 | 0.0016	$8 \cdot 10^{-4}$ | $1 \cdot 10^{-4}$
Fig. 4 A	10	16	7	6	18	119,114,125	0.0216	$9 \cdot 10^{-5}$
Fig. 9 C	14	20 | 20	6 | 17	4 | 0	36 | 39	117, 26,125 | 104, 32,125	0.0061 | 0.0005	$7 \cdot 10^{-5}$ | $1 \cdot 10^{-4}$
Fig. 9 G	16	31 | 31	10 | 5	1 | 6	41 | 26	73, 29,125 | 107, 84,125	0.0005 | 0.0005	$4 \cdot 10^{-5}$ | $1 \cdot 10^{-4}$
Fig. 3 A	17	31 | 18	16 | 8	0 | 8	45 | 44	65, 19,119 | 51, 16,105	0.0002 | 0.0005	$2 \cdot 10^{-5}$ | $3 \cdot 10^{-5}$
Fig. 6 A	20	32	54	0	18	102, 84,125	0.0051	$4 \cdot 10^{-4}$
Fig. 9 B	20	50 | 50	40 | 12	1 | 13	17 | 18	123,108,125 | 27, 13, 40	0.0010 | 0.0010	$1 \cdot 10^{-4}$ | $1 \cdot 10^{-4}$
Fig. 9 D	20	37 | 37	1 | 4	1 | 0	34 | 27	4, 2, 33 | 19, 3, 64	0.0023 | 0.0025	$1 \cdot 10^{-6}$ | $6 \cdot 10^{-6}$
Fig. 9 H	20	59 | 59	3 | 4	4 | 4	11 | 15	35, 20, 56 | 34, 15,125	0.0006 | 0.0015	$4 \cdot 10^{-6}$ | $6 \cdot 10^{-6}$
Fig. 1 A	20	25 | 32	8 | 64	0 | 0	53 | 39	30, 8, 96 | 46, 20, 97	0.0010 | 0.0007	$2 \cdot 10^{-5}$ | $2 \cdot 10^{-5}$
Fig. 2 B	20	41 | 18	17 | 36	17 | 0	18 | 8	116, 52,125 | 111, 22,125	0.0223 | 0.0006	$3 \cdot 10^{-4}$ | $4 \cdot 10^{-4}$
		18	4	4	18	30, 13, 85	0.0013	$4 \cdot 10^{-4}$
Fig. 2 B	23	39 | 63	32 | 5	0 | 5	37 | 25	86, 45,125 | 124,112,125	0.0007 | 0.0006	$1 \cdot 10^{-4}$ | $1 \cdot 10^{-4}$
		63	5	0	44	16, 6, 29	0.0007	$1 \cdot 10^{-4}$
Fig. 1 B	26	41 | 41	9 | 4	18 | 0	24 | 15	105, 64,123 | 110, 15,125	0.0024 | 0.0016	$3 \cdot 10^{-6}$ | $4 \cdot 10^{-5}$

to 26. In all cases, we converge to a solution with low loss value within the allotted iterations. The final loss value does not depend on the number of parameters, while the pattern complexity impacts the number of iterations. On the contrary, the number of selected points does not significantly influence the number of iterations nor the final loss. The type of edit does slightly influence the number of iterations, that always remains within the maximum budget.

We tested patterns with different types and number of shapes as well as with different symmetries. In our tests, we included patterns with continuous and discrete parameters. The latter are shown when editing the circular patterns of Figure 1 (B) and Figure 9 (D, H), and they are handled as discussed in the previous section. We finally verified that shape occlusion can be handled without concerns by editing the shingles patterns in Figure 9 (A, E) where we selected points on the topmost shape without introducing unwanted constraints on the occluded shapes. None of these factors significantly affect the final loss nor the number of iterations.

We optimize all parameters at once. This allows us perform complex edits that require the concurrent change of multiple parameters, as shown in Figure 2 (B–first edit), Figure 9 (B–first edit) and Figure 9 (H–first edit), where, respectively, 4, 4 and 2 parameters are modified concurrently. Furthermore, the use of a gradient-based solver ensures that parameters that do not affect the current edit are left unchanged during interaction, like in the edit of Figure 2 (A) or Figure 3.

From a user perspective, our method is simpler than using sliders since artists do not have to find which parameters produce a desired change. This becomes important as the number of parameters increase, which does not affect our method but makes slider-based manipulation more cumbersome. The example in Figure 1 (B) shows a complex case with 26 parameters, for which finding the correct slider to change could be time-consuming in a manual scenario.

Besides structured patterns, we support deformations in the pattern shapes and in their placement, as shown in Figure 4 and Figure 9 (E, F). We obtained these deformations by applying noise functions to the shapes boundaries and their transformations, as is common in procedural texturing. These results show that we can control procedural deformation just as well as structured patterns.

We also tested the accuracy of the optimized procedural parameters with respect to correct values. We consider the same edits as in Figure 9 and use the estimated procedural parameters of the last frame as target parameters. Then, for each frame, we compute a set of parameters that is linearly interpolated between the starting parameters and the target parameters. We compute the positions of the selected points by evaluating the procedural function at each frame with the linearly interpolated parameters. With these positions, we estimate the procedural parameters with our method and compare them with the correct ones used to generate the points positions. We perform the comparison by computing a mean squared error. As shown in Table 1, these errors are very small in every test we performed, and do not depend on the pattern, selection and edit complexity. This tests confirms that we compute accurate pa-

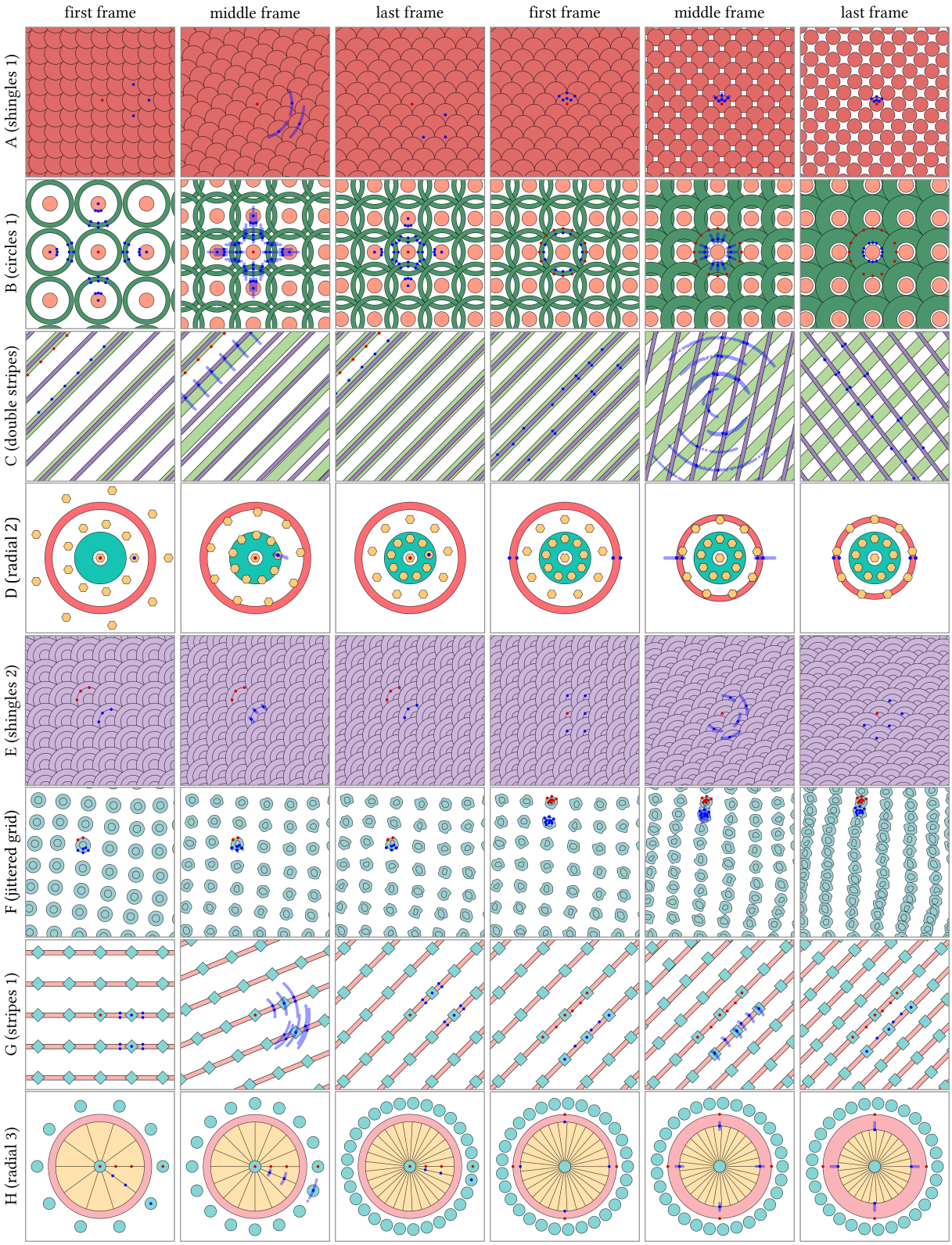

Figure 9: *Gallery of edits on a variety of differentiable vector patterns such as grids, stripes, and radial patterns, with possibly occlusion, deformation and jittering. For each pattern, we show two consecutive edits.*

rameters throughout the mouse interaction, thus proving users with precise feedback while editing.

4.1. Limitations

We believe that our work has three main limitations. First, we support only shapes with parametrized interiors to perform selection and optimization. While most vector graphics primitives are easily parametrizable, there may be others for which this is problematic. One possibility would be to tessellate the shape interior as planar meshes, and use the mesh vertices, that are unique, as interior points. This method is trivial to implement, and would certainly solve the concern presented, but it may slow down the computation.

The second limitation, inherent in all optimization methods, is the increase in computation time as the number of procedural parameters increases. In this paper, we show patterns with more than twenty parameters, which would be quite cumbersome to manipulate with sliders. However, scaling to hundreds or thousands of parameters might become problematic. To support these cases, we need a method that optimizes only the parameters that control the users intended edit, expressed by both selection and mouse motion. We leave this investigation to future work.

Finally, our method may fail when the user edits cannot be reproduced by changes in the program parameters. For example, In Figure 10, the user performs an edit equivalent to shearing the grid. If the procedural program cannot generate a sheared grid, the optimization may incorrectly update other parameters to better fit the transformed points with the ones computed by the program, as shown in the top row. On the contrary, the resulting edit matches the user intent perfectly if the procedural program can represent the given edit, as shown in the bottom row.

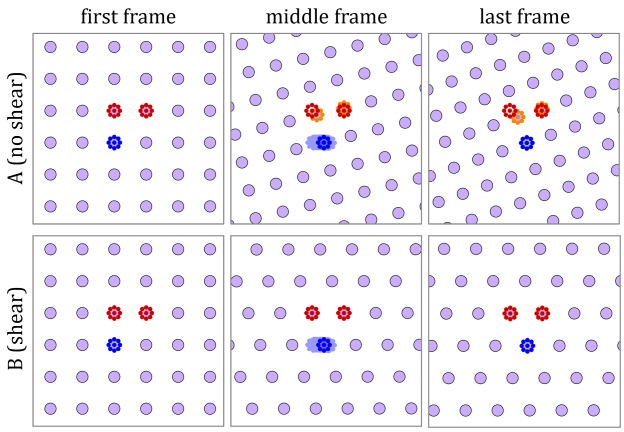

Figure 10: *A limitation of our approach is that user edits can produce undesired results if the procedural program has no parameter combination that can fit user edits. In this figure, the user intent of shearing the grid ends in a rotation update in a procedural program that cannot express shearing (top), while it is correctly handled by a program that can model the edit (bottom).*

5. Conclusions

In conclusion, we presented a method for the direct manipulation of procedural vector patterns. We support patterns expressed as differentiable functions that take parametrized points as input and compute the point positions as output. Users manipulate these patterns by transforming sets of points, while constraining other points. During the interaction, we optimize patterns at each frame to give users real-time feedback on the edit, while ensuring an accurate estimation of the pattern parameters. In the future, we plan to explore methods to write procedural patterns automatically using neural networks and language models.

References

[ADBW16] ALIAGA D., DEMIR I., BENES B., WAND M.: Inverse procedural modeling of 3d models for virtual worlds. In *ACM SIGGRAPH Courses* (07 2016), pp. 1–316. 4

[Ado22] ADOBE: Adobe substance 3d designer. https://www.adobe.com/products/substance3d-designer.html, 2022. 2

[ALCS18] ARISTIDOU A., LASENBY J., CHRYSANTHOU Y., SHAMIR A.: Inverse kinematics techniques in computer graphics: A survey. *Computer Graphics Forum 37* (2018). 2

[BB89] BARTELS R. H., BEATTY J. C.: A technique for direct manipulation of spline curves. In *Proceedings of Graphics Interface '89* (1989), GI '89, pp. 33–39. 2

[BBT*06] BARLA P., BRESLAV S., THOLLOT J., SILLION F. X., MARKOSIAN L.: Stroke Pattern Analysis and Synthesis. In *Proc. of Eurographics 2006 : Computer Graphics Forum* (2006), vol. 25 of *663-671*, ACM. 3

[CSQ*22] CASCAVAL D., SHALAH M., QUINN P., BODIK R., AGRAWALA M., SCHULZ A.: Differentiable 3d cad programs for bidirectional editing. *Computer Graphics Forum 41*, 2 (2022), 309–323. 2

[DB15] DIEDERIK P. K., BA J.: Adam: A method for stochastic optimization. In *ICLR 2015* (2015). 6

[EF01] EFROS A., FREEMAN W.: Image quilting for texture synthesis and transfer. *Proc. SIGGRAPH'01 35* (07 2001). 3

[EL99] EFROS A., LEUNG T.: Texture synthesis by non-parametric sampling. In *Proc. of ICCV* (1999). 3

[GAM*21] GIESEKE L., ASENTE P., MĚCH R., BENES B., FUCHS M.: A survey of control mechanisms for creative pattern generation. *Computer Graphics Forum 40*, 2 (2021), 585–609. 4

[GBLM16] GUERRERO P., BERNSTEIN G., LI W., MITRA N. J.: PATEX: Exploring pattern variations. *ACM Trans. Graph. 35*, 4 (2016). 2

[GEB15] GATYS L., ECKER A. S., BETHGE M.: Texture synthesis using convolutional neural networks. In *Advances in Neural Information Processing Systems* (2015), vol. 28. 3

[GEB16] GATYS L. A., ECKER A. S., BETHGE M.: Image style transfer using convolutional neural networks. In *Proc. of CVPR* (June 2016). 3

[GHS*22] GUERRERO P., HASAN M., SUNKAVALLI K., MECH R., BOUBEKEUR T., MITRA N. J.: Matformer: A generative model for procedural materials. *ACM Trans. Graph. 41*, 4 (2022). 4

[GHYZ19] GUO Y., HASAN M., YAN L.-Q., ZHAO S.: A bayesian inference framework for procedural material parameter estimation. *Computer Graphics Forum 39* (2019). 4

[GJB*20] 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 (2020), 155:1–155:13. 4

[GKG*22] GAILLARD M., KRS V., GORI G., MECH R., BENES B.: Automatic differentiable procedural modeling. *Computer Graphics Forum 41* (05 2022), 289–307. 2

[GSH*20] GUO Y., SMITH C., HASAN M., SUNKAVALLI K., ZHAO S.: Materialgan: reflectance capture using a generative svbrdf model. *ACM Transactions on Graphics (TOG) 39* (2020), 1 – 13. 4

[HDR19] HU Y., DORSEY J., RUSHMEIER H. E.: A novel framework for inverse procedural texture modeling. *ACM Transactions on Graphics (TOG) 38* (2019), 1 – 14. 2, 4

[HGH*22] HU Y., GUERRERO P., HASAN M., RUSHMEIER H., DESCHAINTRE V.: Node graph optimization using differentiable proxies. In *ACM SIGGRAPH 2022 Conference Proceedings* (2022), SIGGRAPH '22. 2, 4

[HHD*21] HU Y., HE C., DESCHAINTRE V., DORSEY J., RUSHMEIER H. E.: An inverse procedural modeling pipeline for svbrdf maps. *ACM Transactions on Graphics (TOG) 41* (2021), 1 – 17. 2, 4

[HHG*22] HU Y., HAŠAN M., GUERRERO P., RUSHMEIER H., DESCHAINTRE V.: Controlling material appearance by examples. *Computer Graphics Forum 41*, 4 (2022), 117–128. 4

[HLC19] HEMPEL B., LUBIN J., CHUGH R.: Sketch-n-sketch: Output-directed programming for svg. *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (2019). 2

[HLT*09] "HURTUT T., LANDES P.-E., THOLLOT J., GOUSSEAU Y., DROUILHET R., COEURJOLLY J.-F.: Appearance-guided synthesis of element arrangements by example. In *Proc. of the 7th International Symposium on Non-photorealistic Animation and Rendering (NPAR'09)* (2009), ACM Press, pp. 51–60. 3

[IMH05] IGARASHI T., MOSCOVICH T., HUGHES J. F.: As-rigid-as-possible shape manipulation. *ACM Trans. Graph. 24*, 3 (2005). 2

[IMIM08] IJIRI T., MECH R., IGARASHI T., MILLER G.: An Example-based Procedural System for Element Arrangement. *Computer Graphics Forum* (2008). 4

[JBPS11] JACOBSON A., BARAN I., POPOVIĆ J., SORKINE O.: Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph. 30*, 4 (2011). 2

[KEBK05] KWATRA V., ESSA I., BOBICK A., KWATRA N.: Texture optimization for example-based synthesis. *ACM Trans. Graph.* (jul 2005), 795–802. 3

[LLGRK20] LI T.-M., LUKÁČ M., GHARBI M., RAGAN-KELLEY J.: Differentiable vector graphics rasterization for editing and learning. *ACM Trans. Graph. 39*, 6 (2020). 2, 3

[MB21] MICHEL E., BOUBEKEUR T.: Dag amendment for inverse control of parametric shapes. *ACM Transactions on Graphics (TOG) 40* (2021), 1 – 14. 2, 5

[MWLT13] MA C., WEI L.-Y., LEFEBVRE S., TONG X.: Dynamic element textures. 4

[NBA18] NISHIDA G., BOUSSEAU A., ALIAGA D. G.: Procedural modeling of a building from a single image. *Computer Graphics Forum 37* (2018). 4

[Pel10] PELLACINI F.: envylight: an interface for editing natural illumination. *ACM Transactions on Graphics 29* (07 2010), 1. 2, 3, 4

[PTG02] PELLACINI F., TOLE P., GREENBERG D. P.: A user interface for interactive cinematic shadow design. *ACM Trans. Graph. 21*, 3 (2002), 563–566. 2, 3, 4

[RGF*20] REDDY P., GUERRERO P., FISHER M., LI W., MITRA N. J.: Discovering pattern structure using differentiable compositing. *ACM Trans. Graph. 39*, 6 (2020). 2, 3

[ROM*15] ROVERI R., ÖZTIRELI A. C., MARTIN S., SOLENTHALER B., GROSS M.: Example based repetitive structure synthesis. Eurographics Association, p. 39–52. 4

[RSP22] RISO M., SFORZA D., PELLACINI F.: pop: Parameter optimization of differentiable vector patterns. *Computer Graphics Forum 41* (2022). 2, 3, 4

[SBM*10] STAVA O., BENES B., MECH R., ALIAGA D. G., KRISTOF P.: Inverse procedural modeling by automatic generation of l-systems. *Computer Graphics Forum* (2010). 4

[SLH*20] SHI L., LI B., HASAN M., SUNKAVALLI K., BOUBEKEUR T., MECH R., MATUSIK W.: Match: differentiable material graphs for procedural material capture. *ACM Trans. Graph. 39* (2020), 196:1–196:15. 2, 4

[SPK*14] STAVA O., PIRK S., KRATT J., CHEN B., MECH R., DEUSSEN O., BENES B.: Inverse procedural modelling of trees. *Computer Graphics Forum* (2014), n/a–n/a. 4

[SPN*14] SCHMIDT T.-W., PELLACINI F., NOWROUZEZAHRAI D., JAROSZ W., DACHSBACHER C.: State of the art in artistic editing of appearance, lighting, and material. *Computer Graphics Forum 35* (01 2014). 3

[TMK*19] TRUNZ E., MERZBACH S., KLEIN J., SCHULZE T., WEINMANN M., KLEIN R.: Inverse procedural modeling of knitwear. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), pp. 8622–8631. 4

[TWY*20] TU P., WEI L.-Y., YATANI K., IGARASHI T., ZWICKER M.: Continuous curve textures. *ACM Trans. Graph. 39*, 6 (2020). 4

[WYD*13] WU F., YAN D.-M., DONG W., ZHANG X., WONKA P.: Inverse procedural modeling of facade layouts. *ACM Transactions on Graphics (TOG) 33* (2013), 1 – 10. 4

[ZHD*22] ZHOU X., HASAN M., DESCHAINTRE V., GUERRERO P., SUNKAVALLI K., KALANTARI N. K.: Tilegen: Tileable, controllable material generation and capture. In *SIGGRAPH Asia Conference Papers* (2022), pp. 34:1–34:9. 4

[ZZB*18] ZHOU Y., ZHU Z., BAI X., LISCHINSKI D., COHEN-OR D., HUANG H.: Non-stationary texture synthesis by adversarial expansion. *ACM Trans. Graph. 37*, 4 (jul 2018). 3

[ZZBW15] ZHUO H., ZHOU S., BENES B., WHITTINGHILL D.: User-assisted inverse procedural facade modeling and compressed image rendering. In *Advances in Visual Computing* (2015), pp. 126–136. 4