Eurographics Symposium on Rendering 2022
A. Ghosh and L.-Y. Wei
(Guest Editors)

Volume 41 (2022), Number 4

pOp: Parameter Optimization of Differentiable Vector Patterns

Marzia Riso[1], Davide Sforza[1] and Fabio Pellacini[1]

[1]Sapienza University of Rome, Italy

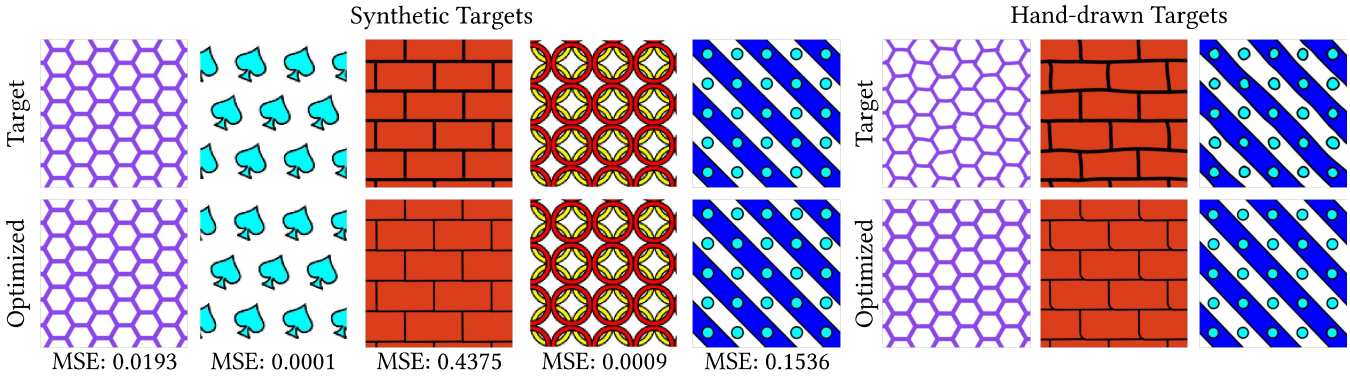

Figure 1: pOp *finds the parameters of procedural vector graphics patterns that best match target images. We support patterns comprised of standard vector graphics elements, e.g. circles, rectangles, lines, and quadratic Bèzier curves, where the translation, rotation and scale of the elements is defined by an arbitrary procedural program. Here we show several examples from different generators. We tested our algorithm with synthetic input generated from a pattern instance, that lets us measure the goodness of fit, here reported as mean squared error (MSE) of the procedural parameters. We also include examples fitted from hand-drawn input, mimicking a possible design application.*

Abstract

Procedural materials are extensively used in computer graphics, since they provide editable, resolution-independent representation of textures. However, tuning the parameters of procedural generators to achieve a desired result remains time-consuming for users. Recently, inverse procedural material algorithms have been developed, exploiting differentiable rendering methods to find the parameters of a procedural model that match a target image. These approaches focus on raster textures. We propose pOp*, a practical method for estimating the parameters of vector patterns, that are formed by collections of vector shapes arranged by an arbitrary procedural program. In our approach, patterns are defined as arbitrary programs, that control the translation, rotation and scale or vector graphics elements. We support elements typical of vector graphics, namely points, lines, circle, rounded rectangles, and quadratic Bèzier drawings, in multiple colors. We optimize the program parameters by automatically differentiating the signed distance field of the drawing, which we found to be significantly more reliable than using differentiable rendering of the final image. We demonstrate our method on a variety of cases, representing the variations found in structured vector patterns.*

CCS Concepts

• ***Computing methodologies*** → ***Texturing;***

1. Introduction

Procedural content creation is heavily used in computer graphics since it produces high-quality, resolution-independent assets that can be easily edited to produce countless variations. Procedural synthesis is particularly well suited to texture generation, since textures are time consuming to create otherwise. A procedural generator can be thought of as a function that produces a texture guided by a set of parameters chosen by artists while modeling. Many such procedural generators are easily available and cover a large class of textures, e.g. [ADO]. But as the number of parameters increases,

determining the parameter values needed to obtain the desired look becomes time-consuming.

To alleviate this issue, inverse procedural modeling techniques attempt to find the parameter set of a given procedural generator that matches a target image. In particular, recent works like [HDR19, GHYZ20, SLH*20, HHD*22], use optimization of differentiable textures, optionally combined with neural networks and texture synthesis to provide a solution in this domain.

In this paper, we focus on procedural vector patterns, formed by a collection of vector shapes. In our application, the procedural generator is an arbitrary function that places shapes according to the desired pattern, by changing their translation, scale and rotation. Our goal is similar to prior work, in that we want to determine the procedural parameters of a pattern that matches a target image. The main difference with prior work is that we focus on vector patterns treated as collections of shapes, rather than raster textures formed by a grid of pixels.

We find procedural parameters with a gradient-based optimization process, that requires that vector patterns are end-to-end differentiable with respect to the pattern parameters. Prior work on inverse procedural texture synthesis uses differentiable rendering to match the final image. For vector graphics, [LLGRK20] proposed an inverse rendering framework suitable for various optimization tasks. When applied to our domain though, inverse rendering is not a suitable solution since the gradient is vanishingly small when the pattern shapes do not overlap during optimization. We instead propose a loss function based on signed distance fields, that works well in our domain. We show how to compute such loss for vector patterns by combining the known shapes' distance field with a proper composition operator, and how to support arbitrary fill and stroke colors. The operation required for the resulting patterns turn out to be all differentiable, so we can easily support arbitrary patterns written as Python functions, by taking advantage of automatic differentiation frameworks. We determine the exact formulation of the loss function by experimentation and optimize it using gradient-based optimization, where proper initialization is determined experimentally.

We tested our algorithm with a variety of patterns, as shown in Figure 1 and throughout the paper, using both synthetic and hand-drawn input, and found the proposed method reliable, where prior work was not able to find the pattern parameters. We believe that our method may be particularly helpful for users when the number of parameters increases and when patterns that are visually very different can be obtained from the same procedural program, as shown in Figure 2.

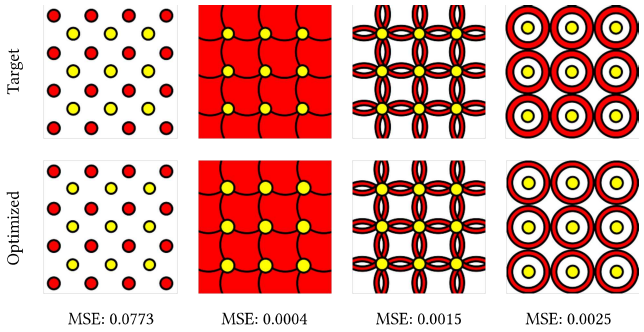

Figure 2: *The same procedural program can create visually-distinctive patterns. Our algorithm optimizes all pattern variations.*

2. Related work

In this section we review relevant works from the Inverse Procedural Modeling literature, focusing on material design tasks.

Inverse Procedural Modeling. The term Inverse Procedural Modeling (IPM) refers to the problem of finding the procedural description given a target model, which can be a texture, a 3D shape or any model that can be procedurally generated. A complete overview of this topic is not possible in this short paper, so we outline here only some contributions in this area.

[SBM*10], one of the first works in this area, proposes a method for objects made up of lines or Bèzier curves. They provide a framework that automatically extracts an L-system, that is a description of a model using compact rules, from a vector image of Bézier curves. [VGDA*12] demonstrates procedural modeling for urban design applications, by proposing a system that optimizes the input parameters of local and global indicators in a 3D procedural model of buildings and cities, based on Markov Chain Monte Carlo (MCMC) during the parameter searching. [BWS10] investigates the inverse procedural modeling of 3D geometry, building a system that automatically creates 3D models that are similar to a target geometry, extrapolating a set of procedural rules that allows fast and reliable object construction. Subsequently, the idea of inverse procedural modeling was adopted in many fields such as the generation of trees from a target model [SPK*14], knitwear [TMK*19], facades or buildings [MZWG07] [NBA18] or the reconstruction of animated sequences [PLL11]. A detailed overview of the inverse procedural modeling of 3D models for virtual applications is exposed by [ADBW16]. While all these works are examples of inverse procedural modeling, they address modeling problems that are quite different from ours.

Inverse Material Design. The works that model textured materials are the ones that mostly relate to our own. In these works, example-based non-parametric texture synthesis has been explored the most, starting with per-pixel [EL99] and per-patch [EF01] stochastic approaches, followed by optimization methods [KEBK05], and many other techniques reported in the comprehensive survey of [WLKT09]. [GEB15] proposed a method that aims to create a texture by extracting and combining features at different levels, using this stationary representation to learn a new texture from noise. Recently, the work of [ZZB*18] focuses on the synthesis of non-stationary textures using Generative Adversarial Networks (GANs), producing a bigger texture that is perceptually similar to a small target image using a generator and discriminator approach. Lastly, [PTWY*20] proposed an example-based framework for continuous curve patterns that extends previous discrete element synthesis methods by involving not only the sample positions but also their topological connections.

While non-parametric texture synthesis works well for many domains, it lacks in "editability", i.e. the ability of users to fine-tune the final texture to match the desired look. Procedural mate-

rial design focuses on that specifically, since the generated textures can be controlled by manipulating the parameters of the generator. However, when the number of parameters is high, users may struggle in finding the parameter values to obtain the desired appearance. This issue has started to be addressed recently for raster textures. [GHYZ20] use Markov Chain Monte Carlo to sample the parameter space to find the procedural parameters that generate the desired texture for unstructured materials such as wood, plastic, leather or metallic paints, starting from a photograph. [SLH*20] presents a more comprehensive method that works on differentiable procedural graphs, automatically converting the procedural nodes in [ADO]. Given a target image, the most promising material graphs are selected using their Gram Matrix computed with a pre-trained VGG network [GEB16]. The material parameters are then refined using gradient-based optimization of the differentiable material graph. This approach works well for color manipulations but does not support effectively pattern generator nodes, which is the focus of our work.

[HDR19] combines inverse procedural textures and texture synthesis in a comprehensive framework for inverse material design. The parameters of an inverse procedural program are estimated via clustering and by using a Convolutional Neural Network (CNN) trained for parameter estimation. To better match the desired look, the result is augmented via non-parametric style transfer. In [HHD*22], the authors improve upon their previous work by presenting a semi-automatic pipeline for material proceduralization given SVBRDFs maps. The framework hierarchically decomposes them into sub-materials, that are proceduralized using a multilayer noise model capable to capture local variations. They reconstruct procedural material maps using a differentiable rendering-based optimization that minimizes the distance between the generated procedural model and the input material pixel-map.

Our work is mostly related to these techniques. We follow an inverse procedural approach using optimization to determine material parameters from input images that supports arbitrary differentiable procedural programs. The main difference with prior work is that we focus on vector graphics textures rather than raster ones. Furthermore, many previously described techniques rely on training neural networks to estimate a parameter initialization. However, collecting data and offline training are time-consuming processes, that may also be iterated if new assets need to be supported. On the contrary, we decided to rely on an online parameter initialization, which is included in our pipeline.

[MB21] demonstrates interactive manipulation of parametric procedural shapes by parameter optimization using differentiable procedural graphs, obtained by augmentation of shape modeling graph with differentiation features. The application of these ideas to material modeling is an interesting avenue of future work.

Lastly, the work of [LLGRK20] proposes a differentiable rasterizer for vector graphics that fills the gap between vector and raster graphics. The authors demonstrate that their differentiable renderer supports interactive editing, image vectorization, painterly rendering, seam carving and generative modeling in a gradient-based optimization process. We initially based our work on this approach, but found that it does not work well for patterns made of many shapes, as demonstrated in the next chapters.

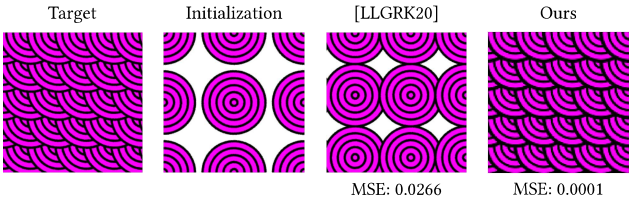

Figure 3: *When trying to optimize a pattern starting from the same initial configuration, we found that losses based on image difference, and relate differentiable renderers such as [LLGRK20], do not work well in our problem domain. We instead define a loss based on pattern SDFs that is robust to all pattern variations.*

3. Differentiable Vector Patterns

Procedural Vector Patterns. We focus on patterns made of collections of vector primitives instantiated on a canvas. In our implementation, we support a subset of the SVG (Scalable Vector Graphic) standard shapes such as circles, rectangles, line segments and SVG paths consisting of quadratic Bézier curves. Every shape is described by its geometric parameters, as well as its rendering style comprised of fill color, stroke color, and stroke thickness. In our work, we do not support semi-transparent shapes as well as linear and radial color gradients.

Each pattern is represented as an arbitrary function that describes the position, rotation and scale of vector primitives, controlled by a set of pattern parameters. For example, a grid pattern is a function parametrized over the grid offsets. In this work, we focus on structured and non-stochastic patterns, since many works have already focused on stochastic procedural materials [HDR19, GHYZ20, SLH*20, HHD*22]. In our implementation patterns are written as arbitrary Python functions, which are significantly more expressive than node graphs, and support completely general shape arrangements.

Inverse Procedural Patterns. Given a target image and a procedural function able to reproduce such image, we seek to estimate the function parameters that reproduce the target image. In other words, our goal is to identify the parameter set Φ^* of the given procedural function G that minimizes a loss $\mathcal{L}$ between the target image I and the one generated by G.

$$\Phi^* = argmin_{\Phi}\, \mathcal{L}(I, G(\Phi)) \tag{1}$$

Prior work on inverse procedural materials uses differentiable renderers combined with gradient-based optimization to estimate the procedural parameters [GHYZ20, SLH*20, HDR19, HHD*22]. In the context of vector graphics, [LLGRK20] present a differentiable rasterizer for SVG elements that we attempted to apply in our work. In this case, a loss is computed between a target image and the rendered image obtained by rasterization. Since the rasterizer is differentiable, gradient-based optimization is used to find the parameter set that best fits the target image.

However, as shown in Figure 3, measuring the loss as an image difference together with a differentiable rasterizer has significant issues for vector patterns comprised of many small shapes. The case shown in this previous image is particularly problematic since

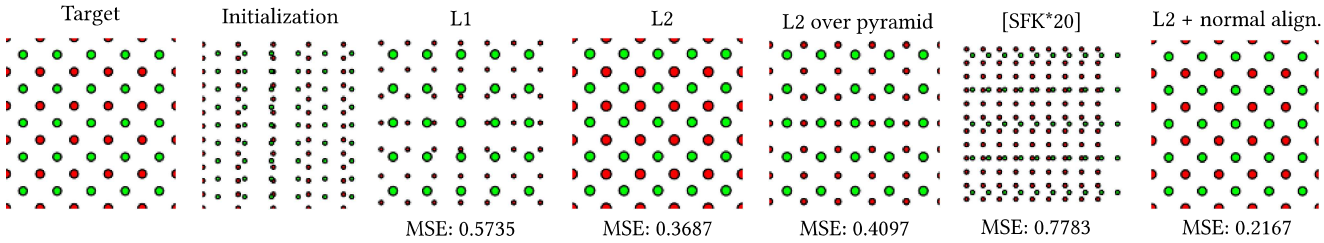

Figure 4: *Comparison of the results obtained with different loss functions, reporting both the pattern image and the MSE of the optimized pattern parameters, where all parameter sets are optimized from the same initialization. We compare L_1, L_2, L_2 over a gaussian pyramid, the metric proposed in [SFK*20], and the L_2 distance with normal alignment. The latter loss was chosen since it works best in our tests.*

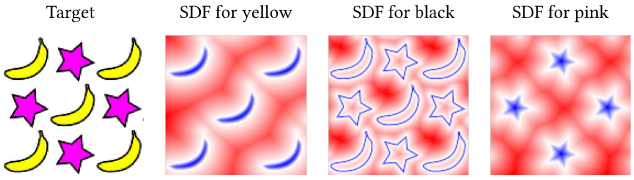

Figure 5: *Colored patterns are supported by computing an SDF for each color. The loss function is the sum of the contribution corresponding to each color. Here we display SDFs from blue to red for negative to positive values respectively.*

many shapes with constant colors overlap. In this case, gradients derived from an image difference metric become small and inaccurate. A similar problem occurs when small shapes do not overlap, where the gradient becomes unreliable since it captures differences only on shape boundaries. Overall these issues make the gradient vanishingly small for most parameters' configurations, making image differences unstable in our context.

Loss Function. To overcome this issue, *we measure pattern differences based on the signed distance fields of the pattern's shape elements*. This ensures that gradients are well defined for all parameters' configurations. A signed distance field for a pattern is a function that measures the minimum distance between a point and the boundaries of the shapes. By convention, we assign a negative distance if the point is inside a shape, and a positive distance otherwise.

Since a pattern may have multiple colors, we consider the distance to the shape of each color separately, as shown in Figure 5. Note that if a shape is drawn with a different stroke and fill colors, its SDF is different for each color. For the stroke color we consider the geometry of the shape boundary, while for the fill color we consider the geometry of the shape interior. If we indicate with S_c the signed distance corresponding to color c, the loss between the target image I and the generated pattern $G(\Phi)$ is the sum of the a per-color difference between their SDFs, which can be written as:

$$\mathcal{L}_S = \frac{1}{|C|} \sum_{c \in C} \mathcal{D}(S_c(I), S_c(G(\Phi))) \tag{2}$$

where C is the set of pattern colors.

We measure the difference $\mathcal{D}$ between SDFs with the L_2 distance, which was chosen by experimentation. In particular, we experimentally compared this metric with the L_1 metric and the L_2 metric applied over the levels of a Gaussian image pyramid, and found L_2 to work best for our problem. An example of this comparison is shown in Figure 4.

[SFK*20] introduces a loss between SDFs for the case of fitting the control points of quadratic Bèzier curves for a single shape. Their metric is composed of a distance metric of the shape SDFs, together with a normal alignment term that measures the alignment of the shape normals. For our problem domain, the L_2 distance worked significantly better. This is due to the fact that [SFK*20] measures SDF differences only in the image regions near shape boundary, which makes the gradient vanishingly small for most parameter sets in our case, as we have already discussed.

At the same time, we observe that combining the normal alignment metric with the L_2 metric improves optimization convergence when shapes positions are close to the target image, making the SDF loss vanishingly small, while the normal alignment remains significant. In our notation, the normal alignment loss is written as:

$$\mathcal{L}_N = \frac{1}{|C|} \sum_{c \in C} \left[1 - \langle \nabla S_c(I), \nabla S_c(G(\Phi)) \rangle^2 \right] \tag{3}$$

where gradients are normalized. The final loss $\mathcal{L}$ between a target image and the procedural pattern is the weighted sum of both terms, written as:

$$\mathcal{L} = \mathcal{L}_S + \alpha_N \cdot \mathcal{L}_N \tag{4}$$

where the weight α_N is set to 0.05 according to the results of a hyperparameter tuning process.

Pattern Distance Fields. To evaluate the loss, we need to compute the distance field of each pattern color. We can compute the distance field by considering each vector element separately, and then combining those fields appropriately.

For basic vector graphics elements, the signed distance field can be analytically computed. The formulas for circles, rectangles and line segments can be found in [Qui], while the formulas for quadratic Bèzier curves are presented in [SFK*20]. In our implementation, we approximately convert cubic Bèzier to quadratic ones, since the latter have simpler and numerically-robust distance formulas.

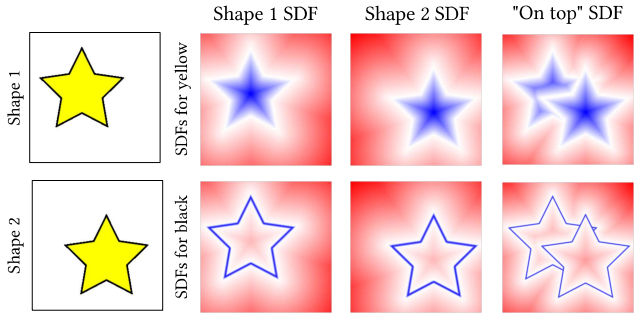

Figure 6: *SDF corresponding to the operation of drawing a shape onto another, compared to the SDFs for the single shapes.*

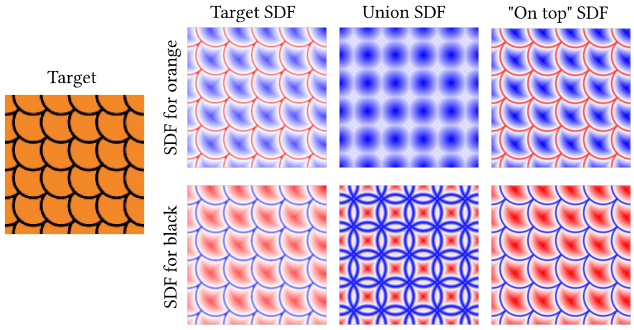

Figure 7: *Comparison between combining overlapping shapes with boolean unions or with our operator that captures drawing shapes on top of one another. While the union operator captures shapes correctly, it cannot represent shapes that have both stoke and fill color. Our operator matches exactly the SDFs corresponding to the stroke and fill colors extracted from the target image.*

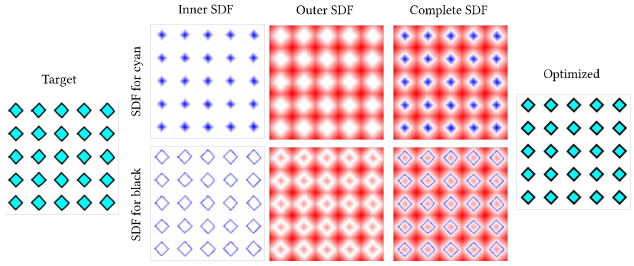

Figure 8: *SDF extracted from an hand-drawn target image. For each image color, we extract a binary mask that is used to compute both inner and outer distance using the Euclidean distance transform, flipping the mask values accordingly.*

We combine the shapes SDFs into the pattern SDF using a variety of operators supported in vector graphics. SDFs support boolean operators, namely union, intersection and difference, in a straightforward manner. We adopt these operations in our prototype implementation. But, in the vast majority of times, vector graphics elements are combined by drawing elements on top of the previous ones, which does not correspond to any boolean operations for shapes with fill and stroke colors. For this case, we introduce a new operator that computes the SDF of a shape drawn on top of another, illustrated in Figure 6. The reason why the union operator cannot be used to combine shapes drawn one after the other is that stroke and fill colors do not properly combine, as shown in Figure 7.

Let us consider the operation of drawing a shape onto a background. Since we compute SDFs for each color, we focus on a single color c, and consider three SDFs: the background SDF for c, indicated as S_c^B, and the SDFs for the stroke of fill of the foreground, indicated as S_s^F and S_f^F. The resulting SDF for the selected color depends on the fill and stroke colors of the foreground, giving us four cases. (1) If the selected color is the same as both the stroke and fill colors, then we want to include the whole foreground into the SDF, which can be done using a boolean union. (2) Conversely, if the selected matches neither the foreground stroke and fill, then we want to remove the whole foreground from the background SDF, which can be done using a boolean difference. (3) If the selected color matches the foreground stroke color, but not its background, then we have to include the former and exclude the latter resulting in a union followed by a difference. (4) Finally, if the selected color matches the background fill, we perform a difference followed by a union. Formally, we summary write:

$$\mathrm{OnTop}(B,F,c) = \begin{cases} \mathrm{union}(S_c^B, \mathrm{union}(S_s^F, S_f^F)) & \text{for } c = s \wedge c = f \\ \mathrm{diff}(S_c^B, \mathrm{union}(S_s^F, S_f^F)) & \text{for } c \neq s \wedge c \neq f \\ \mathrm{diff}(\mathrm{union}(S_c^B, S_s^F), S_f^F) & \text{for } c = s \wedge c \neq f \\ \mathrm{union}(\mathrm{diff}(S_c^B, S_s^F), S_f^F) & \text{for } c \neq s \wedge c = f \end{cases}$$

Figure 6 shows an example of combining two shapes with the same stroke and fill colors, resulting in the last two cases described here.

Differentiable Pattern SDFs. We estimate the procedural parameters using gradient-based optimization, which requires gradients to be computed for all parameters. To this end, we employ the automatic differentiation facilities in PyTorch [PGM*19]. Our patterns are written starting from single shapes that are either drawn on top of one another or combined with boolean operations. All these operations are just a sequence of calls between automatically differentiable functions, making automatic differentiation feasible. In all cases, automatic differentiation can compute derivatives via backpropagation, without any further intervention from the pattern author.

Target Image SDF. SDFs can be computed analytically when shapes are provided explicitly. For our application, only target images are provided, so the target SDFs need to be computed from the raster representation. The SDF extraction procedure is shown in Figure 8. Since we compute an SDF per color, we first compute a binary mask of the image that selects the areas that match the given color. Once the mask is extracted, we apply the Euclidean distance transform to extract the distance inside of the shape, and apply the same procedure to the inverted mask to extract the positive distance. In the end, we combine the two distance transformations to obtain the complete target signed distance.

4. Parameter Optimization

Optimization Procedure. As described before, the goal of our work is to determine the procedural parameters so that the generated vector pattern matches a target image. Such parameters are computed by minimizing the loss specified in Equation 1. We consider as optimizable parameters only the ones that influence the pattern geometric properties. Conversely, the shape element types and their colors are given apriori and thus not considered in the optimization process.

We determine the procedural parameters using an iterative gradient-based optimization process that relies on the end-to-end differentiability of our pattern SDFs. More specifically, we use the Adam optimizer [DB15], with a learning rate of *0.025*. We choose Adam since it works well for our problem. The target SDFs are computed as described above and do not need to be differentiable.

Parameter Set Initialization. At the beginning of the optimization, a proper initialization is necessary to ensure that we do not get stuck in local minima. We tested several initialization strategies and pick the one that works best for our case. In particular, we considered random sampling, Nelder-Mead's simplex and Genetic Algorithms, as illustrated in Figure 9.

Random sampling picks a set of initialization candidates randomly in the whole parameter set and maintains the set of candidates that most closely matches the target image. Even with relatively few parameters, we found that this procedure achieves poor results. One of the main issues is that an error introduced by even a single parameter might reject a candidate, even if all other parameters were selected well. For this reason, we iteratively sample a single parameter at each iteration, thus not losing some good parameters initialization discovered in the previous ones.

The Nelder-Mead's simplex algorithm [NM65] solves the unconstrained optimization of a function by evaluating it at a set of points that form a high-dimensional simplex. Then, it iteratively shrinks the simplex by replacing the point with the highest error with another obtained using the reflection, expansion or contraction operations. This step is repeated a desired number of times.

Genetic Algorithms [For96] are a class of computational models inspired by the idea of evolution. In our setting, the variant we use first samples a set of candidates randomly in the whole domain. Candidates are ranked based on their error, and the best ones are picked to participate in the next generation. New candidates are then generated by using a 2-point crossover operation, with further random mutations. This step is repeated a desired number of times.

In our experiments, we found random sampling to perform worst, with the simplex method and genetic algorithms to work well. For example, in Figure 9 we compare the three initialization both visually and by computing the Mean Squared Error (MSE) between the target parameters and the computed initializations. In this example, and in general, the Genetic Algorithms achieves the best results, due to their capacity of propagating partial best parameters from an iteration to the following ones.

Parameter Set Optimization. To reduce the chance to get stuck in local minima, we tested two commonly-used approaches. At first, we adopted an Iterated Local Search [LMS03] approach. This procedure consists in applying iteratively a local search algorithm, represented in our case by gradient descent. At the end of each iteration, a single parameter of the best configuration found so far is modified at random. The new configuration obtained this way is then used to initialize the next local optimization pass. This helps to avoid local minima by perturbing a solution that might not be optimal. However, only the best configuration found by the initialization phase is exploited to initialize the first local search. Alternatively, the best candidates selected during initialization are all optimized separately with gradient descent, but without perturbation. We then select the parameter set with the lowest loss. This technique reduces the chance of local minima by exploring more points in the space. In our experiment, the latter procedure worked best, especially in the case where the number of parameters increases. We chose to use 10 initial candidates for the exploration.

Finally, during an iteration, the loss could keep increasing or oscillating around a minimum. To reduce such behavior we adopt a patience and refinement technique. If the loss between the target pattern and the one computed by the generation increases for a considerable amount of iterations, the best parameters assignment

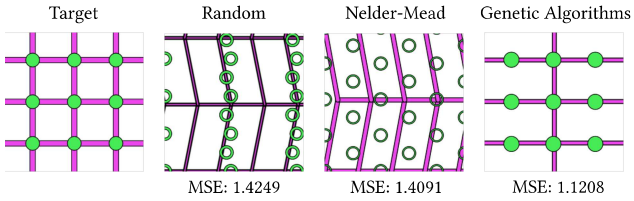

Figure 9: *Comparison between the parameter initialization procedures. Between random sampling, Nelder-Mead's simplex and genetic algorithms, the latter was selected since it works best for our problem.*

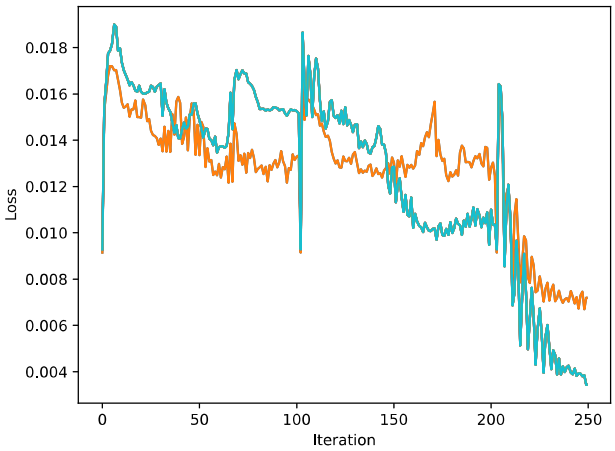

Figure 10: *Loss changes during optimization of the stripe pattern in Figure 1. In our approach, we chose to optimize starting from multiple starting configuration, to reduce the chance of local minima, and pick the best final one. We use 10 candidates in our, but show here only 2 for clarity of presentation.*

is reloaded and the optimizer learning rate is reduced. This strategy helps in case of an oscillation around a minimum since the adopted optimizer parameters could produce high jumps in the loss landscape without gradually leaning towards a minimum. If the optimized parameters are refined with a high frequency, then the iteration is stopped and the best parameters are again reloaded and perturbed before the next iteration starts. We adopted a patience of 100 iterations and a total number of 3 refinements during each framework iteration. At the end of the process, the overall best parameters identified during one of the iterations are returned. An example of optimization is shown in Figure 10.

5. Results

Throughout the paper, we have already shown several patterns whose parameters were fitted with our algorithm. In this section, we analyze the generator functions that were used to obtain the given patterns.

Pattern Optimization. All results in this paper were fit with the same hyper-parameters to further demonstrate the robustness of the approach. In particular, we use 250 gradient descent iterations from the 10 best candidates selected during an initialization of 25 generations, starting from a population of 250 individuals. We execute all operations on a 16-core Ryzen 9 CPU with an NVIDIA 3090 GPU. All code was written in Python and use PyTorch [PGM*19] and SciPy [VGO*20] for optimization and automatic differentiation. The Genetic Algorithm based initialization is implemented using the DEAP [FDG*12] framework.

Table 5 shows the statistics of the optimized patterns. We tested 7 different procedural patterns, that have from 4 to 14 parameters. In our tests, we found that our algorithm can reliably find the parameter sets in all these cases. For the synthetic patterns, we measure the goodness of fit with the MSE of the pattern parameters, which we found to be very low in all cases, going from 0.0001 to 0.4375.

Patterns with tens of parameters are cumbersome for users since all parameters need to be appropriately set. This is sometimes ameliorated by careful parametrization, which makes a single pattern easier to use, but also makes writing new patterns complex. On the contrary, in our work patterns are arbitrary Python functions that we did not specifically write to provide convenient parametrization.

In Figure 11, we show a failure example involving a pattern that is parametrized by 25 parameters. While the number of shapes and colors does not affect convergence negatively, there is a dependence on the number of parameters. Higher number of parameters may influence the initialization stage, providing a poor guess of the initial parameters, that may lead to a convergence in some local minima of the loss function.

Pattern Types. We chose patterns that are quite different in the types of elements they are comprised of. In particular, we showed examples using lines, circles, rectangles, as well as SVG drawings represented as quadratic Bèzier curves. By changing element types and pattern structure, our framework supports a large variety of examples. In fact, patterns that are visually quite different can be obtained from the same procedural function, as shown in Figure 2. Overall, we found our method to work well in all these cases.

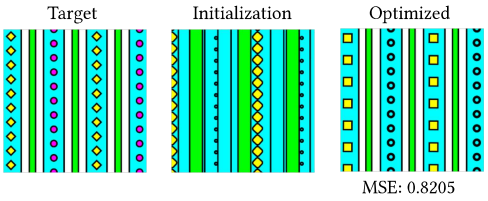

Figure 11: *For patterns with high number of parameters, 25 in this figure, the algorithm may converge to a local minimum.*

Figure	Generator	Parameters	Shapes	MSE
Fig. 1	Honeycomb	4	198	0.0193
	Spades	7	11	0.0001
	Rectangles	8	21	0.4375
	Circles1	12	30	0.0009
	Stripes	14	34	0.1536
Fig. 2	Circles2	14	25	0.0773
	Circles2	14	25	0.0004
	Circles2	14	25	0.0015
	Circles2	14	18	0.0025
Fig. 3	Shingles	6	30	0.0001
Fig. 4	Circles2	14	61	0.2167
Fig. 7	Circles3	7	36	0.0018
Fig. 8	Rectangles	8	25	0.3058

Table 1: *Statistics of the optimized patterns, including number of parameters, number of shapes, and MSE of the fitted parameter set.*

Unsupported Patterns. While our algorithm supports a large variety of patterns, some cases are still not supported, since they would require changes to how patterns are specified via a target image. We leave them for future work.

In particular, we do not support opacity in the color definition, since we treat the element as individual shapes. The concern, in this case, is how to disambiguate non-opaque colors from opaque ones in the target images. [RGF*20] shows a promising direction using texture synthesis, that we might be able to adapt to procedural patterns as well. Furthermore, we only support solid fill and stroke colors since the per-color SDF definition is not well defined for linear or radial color gradients.

Stochastic patterns are also a concern since it is unclear what the target image should be. The authors of [SLH*20] acknowledge this issue and provide an ad-hoc solution for raster texture with small variations that effectively freezes the randomization elements in the pattern. While this idea may work in their domain, fully-stochastic vector patterns may take any arrangement, so it is unclear if a single target image is sufficient to determine their parameters.

6. Conclusions

In this paper, we present a method for computing the parameters of procedural vector patterns that match a given input image. The key idea of our work is to cast the optimization problem in terms of

pattern distance fields, that are made differentiable using automatic differentiation and a careful choice of shape combination operators. In future work, we plan to extend our work to investigate new formulations that support opacity and stochastic patterns, together with exploring the possibility of optimizing the shape element control points as well.

7. Acknowledgments

All the SVG icons used in this work are licensed under Creative Commons. We thank Giacomo Nazzaro and Marco Esposito for their helpful comments and constructive discussions about optimization.

References

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

[ADO] ADOBE: Adobe substance 3d designer. https://www.adobe.com/products/substance3d-designer.html.

[BWS10] BOKELOH M., WAND M., SEIDEL H.: A connection between partial symmetry and inverse procedural modeling. *ACM Trans. Graph. 29*, 4 (2010).

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

[EF01] EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. In *CVPR* (2001), p. 341–346.

[EL99] EFROS A. A., LEUNG T. K.: Texture synthesis by non-parametric sampling. In *CVPR* (1999), vol. 2, pp. 1033–1038.

[FDG*12] FORTIN F., DE RAINVILLE F., GARDNER M., PARIZEAU M., GAGNÉ C.: DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research 13* (2012), 2171–2175.

[For96] FORREST S.: Genetic algorithms. *ACM Computing Surveys (CSUR) 28*, 1 (1996), 77–80.

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

[GEB16] GATYS L. A., ECKER A. S., BETHGE M.: Image style transfer using convolutional neural networks. In *CVPR* (2016), pp. 2414–2423.

[GHYZ20] GUO Y., HAŠAN M., YAN L., ZHAO S.: A bayesian inference framework for procedural material parameter estimation. *Computer Graphics Forum 39*, 7 (2020).

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

[HHD*22] HU Y., HE C., DESCHAINTRE V., DORSEY J., RUSHMEIER H. E.: An inverse procedural modeling pipeline for svbrdf maps. *ACM Trans. Graph. 41*, 2 (2022).

[KEBK05] KWATRA V., ESSA I., BOBICK A., KWATRA N.: Texture optimization for example-based synthesis. 795–802.

[LLGRK20] LI T., LUKÁČ M., GHARBI M., RAGAN-KELLEY J.: Differentiable vector graphics rasterization for editing and learning. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 39*, 6 (2020), 193:1–193:15.

[LMS03] LOURENÇO H. R., MARTIN O. C., STÜTZLE T.: Iterated local search. In *Handbook of metaheuristics*. Springer, 2003, pp. 320–353.

[MB21] MICHEL E., BOUBEKEUR T.: Dag amendment for inverse control of parametric shapes. *ACM Trans. Graph. 40*, 4 (2021).

[MZWG07] MÜLLERL P., ZENG G., WONKA P., GOOL L. V.: Image-based procedural modeling of facades. *ACM Trans. Graph. 26*, 3 (2007), 85–es.

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

[NM65] NELDER J. A., MEAD R.: A Simplex Method for Function Minimization. *The Computer Journal 7*, 4 (1965), 308–313.

[PGM*19] PASZKE A., GROSS S., MASSA F., LERER A., BRADBURY J., CHANAN G., KILLEEN T., LIN Z., GIMELSHEIN N., ANTIGA L., DESMAISON A., KOPF A., YANG E., DEVITO Z., RAISON M., TEJANI A., CHILAMKURTHY S., STEINER B., FANG L., BAI J., CHINTALA S.: Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[PLL11] PARK J. P., LEE K. H., LEE J.: Finding syntactic structures from human motion data. *Computer Graphics Forum 30* (2011).

[PTWY*20] P-TU, WEI L., YATANI K., IGARASHI T., ZWICKER M.: Continuous curve textures. *ACM Trans. Graph. 39*, 6 (11 2020).

[Qui] QUILEZ I.: Inigo quilez. https://iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm.

[RGF*20] REDDY P., GUERRERO P., FISHER M., LI W., MITRA N. J.: Discovering pattern structure using differentiable compositing. *ACM Trans. Graph. (TOG) 39*, 6 (2020), 1–15.

[SBM*10] STAVA O., BENES B., MECH R., ALIAGA D. G., PETER K.: Inverse procedural modeling by automatic generation of l-systems. *Computer Graphics Forum 29* (2010), 1467–8659.

[SFK*20] SMIRNOV D., FISHER M., KIM V. G., ZHANG R., SOLOMON J.: Deep parametric shape predictions using distance fields. In *CVPR* (2020).

[SLH*20] SHI L., LI B., HAŠAN M., SUNKAVALLI K., BOUBEKEUR T., MECH R., MATUSIK W.: Match: Differentiable material graphs for procedural material capture. *ACM Trans. Graph. 39*, 6 (2020), 1–15.

[SPK*14] STAVA O., PIRK S., KRATT J., CHEN B., MŽCH R., DEUSSEN O., BENES B.: Inverse procedural modelling of trees. *Comput. Graph. Forum 33*, 6 (2014), 118–131.

[TMK*19] TRUNZ E., MERZBACH S., KLEIN J., SCHULZE T., WEINMANN M., KLEIN R.: Inverse procedural modeling of knitwear. *CVPR* (2019), 8622–8631.

[VGDA*12] VANEGAS C. A., GARCIA-DORADO I., ALIAGA D. G., BENES B., WADDELL P.: Inverse design of urban procedural models. *ACM Trans. Graph. 31*, 6 (2012), 168:1–168:11.

[VGO*20] VIRTANEN P., GOMMERS R., OLIPHANT T. E., HABERLAND M., REDDY T., COURNAPEAU D., BUROVSKI E., PETERSON P., WECKESSER W., BRIGHT J., VAN DER WALT S. J., BRETT M., WILSON J., MILLMAN K. J., MAYOROV N., NELSON A. R. J., JONES E., KERN R., LARSON E., CAREY C. J., POLAT I., FENG Y., MOORE E. W., VANDERPLAS J., LAXALDE D., PERKTOLD J., CIMRMAN R., HENRIKSEN I., QUINTERO E. A., HARRIS C. R., ARCHIBALD A. M., RIBEIRO A. H., PEDREGOSA F., VAN MULBREGT P., SCIPY 1.0 CONTRIBUTORS: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods 17* (2020), 261–272.

[WLKT09] WIE L., LEFEBVRE S., KWATRA V., TURK G.: State of the art in example-based texture synthesis. In *Eurographics 2009 - State of the Art Reports* (2009), The Eurographics Association.

[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. (Proc. SIGGRAPH) 37*, 4 (2018), 49:1–49:13.