SAPIENZA
UNIVERSITÀ DI ROMA

Enhancing Controllability in Procedural and Non-Procedural Asset Editing

Faculty of Information Engineering, Informatics, and Statistics
Ph.D. in Computer Science (XXXVI cycle)

Marzia Riso
ID number 1699713

Advisor
Prof. Fabio Pellacini

Academic Year 2023/2024

Enhancing Controllability in Procedural and Non-Procedural Asset Editing
Ph.D. Thesis. Sapienza University of Rome

© 2024 Marzia Riso. All rights reserved

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: riso@di.uniroma1.it

"Je me demande, dit-il, si les étoiles sont éclairées
pour que chacun puisse un jour retrouver la sienne."
Antoine de Saint-Exupéry, Le Petit Prince

Abstract

The design of 2D and 3D assets is one of the key components of Computer Graphics applications. Recently, the demand for high-quality assets, like textures and materials, or 3D shapes, has seen impressive growth due to their widespread adoption in industrial design workflows as well as in the movie and video game industries. However, designing an asset is still a consuming operation in terms of time and human effort, in particular when a precise target must be matched. Over the years, various techniques, such as real-world material and model captures or inverse modeling, have been developed to ease this process and to support artists in matching a desired target.

This thesis, entitled **Enhancing Controllability in Procedural and Non-Procedural Asset Editing**, explores the common techniques adopted in asset design and proposes novel approaches to reduce the time and human effort involved in this process, aiming to reach an easier and yet more controllable pipeline for asset editing in both for 2D and 3D environments. We firstly investigate the procedural asset fields, proposing inverse procedural modeling solutions for 2D vector patterns and 3d implicit shapes defined as differentiable programs and graphs respectively. As regards the 2D content, we propose an example-based parameter estimation tool, called **pOp**, and a direct manipulation tool named **pEt**. The first one proposes a parameter estimation method based on a user-provided sketch or render using an inverse Signed Distance Field optimization problem as its backbone. By minimizing the differences between the target pattern SDF and the procedurally generated one, it estimates the procedural parameter assignment that better matches the target design. Still in the parameter estimation field, the second approach enables users to directly manipulate the procedural vector pattern content by performing edits in the viewport by selecting and dragging around sets of points or preventing others from moving. In this scenario, the best parameter assignments are determined by minimizing the distance between sets of points, allowing users to edit multiple parameters all at once without relying on counterintuitive GUI slider tweaking operations.

Similarly in the 3D setting, this thesis proposes a **direct manipulation tool** for **procedurally-defined implicit surfaces,** assuming end-to-end differentiability with respect to the procedural parameters. By defining a method for tracking points on the implicit surface, we set up a gradient descent-based optimization loop that solves for the procedural parameters as the user performs edits directly in the viewport, such as dragging surface patches or constraining other ones. This tool allows for the editing of procedural implicit surfaces at interactive rate, supporting all the operations that make implicit modeling a robust and easy-to-use alternative

to regular 3D modeling. In particular, it supports standard CSG operations and their smooth counterparts, which are widely used in the creation of organic assets, while remaining resilient to topology changes.

Finally, we concentrate on non-procedural asset synthesis, specifically focusing on structured texture generation with widespread diffusion models as a backbone. We propose **pAff**, a method for the expansion of a small user-designed sketch to a large-scale, high-quality, and moreover, tileable content. To do so, we enhance previously assessed diffusion pipelines by injecting structured pattern domain knowledge through a LoRA finetuning process, exploiting the Noise Rolling technique to improve quality and ensure tileability.

In conclusion, this thesis makes a significant contribution in improving user control in procedural and non-procedural asset editing. It proposes novel methods that could be either integrated into common 2D or 3D modeling software and further expanded to various asset design workflows, making the design process easier for both novice and experienced users.

Contents

1 Introduction **1**
1.1 Contributions 2

2 Related Work **5**
2.1 Inverse Procedural Modeling 5
2.1.1 Procedural Program Estimation 6
2.1.2 Procedural Parameter Estimation via Examples 6
2.1.3 Procedural Parameter Estimation via Direct Manipulation . . 8
2.2 Non-procedural Asset Control 10

I Editing Procedural Assets **13**

3 pOp: Parameter Optimization of Differentiable Vector Patterns **14**
3.1 Introduction 15
3.2 Method 16
3.2.1 Differentiable Vector Patterns 16
3.2.2 Loss Function. 19
3.2.3 Optimization 23
3.3 Results 26
3.4 Conclusions 28

4 pEt: Direct Manipulation of Differentiable Vector Patterns **29**
4.1 Introduction 30
4.2 Method 32
4.2.1 User Edit 33
4.2.2 Loss Function 35
4.2.3 Optimization 37
4.3 Results 38
4.3.1 Limitations 41
4.4 Conclusions 42

5 Direct Manipulation of Procedural Implicit Surfaces 44
5.1 Introduction 45
5.2 Overview 46
5.3 Scene Graph Model 48
5.4 Coparameterization 50
5.4.1 Definition 50
5.4.2 Augmented Implicit Function 51
5.5 Evaluation and normalization 53
5.5.1 Jacobian Evaluation 53
5.5.2 Jacobian Normalization 54
5.5.3 Jacobian Reduction and Filtering 55
5.6 Solving 55
5.7 Results 56
5.8 Conclusion 60

II Controlling Non-Procedural Assets 63

6 Structured Pattern Expansion with Diffusion Models 64
6.1 Introduction 65
6.2 Method 67
6.2.1 Guided Image Generation 68
6.2.2 Stable Diffusion Finetuning 70
6.2.3 Patterns expansion 71
6.3 Experimental results 72
6.3.1 Datasets 72
6.3.2 Technical details 73
6.3.3 Results and comparisons 74
6.3.4 Ablation Study 76
6.4 Limitations and Future Work 78
6.5 Conclusion 79

7 Conclusion 82
7.1 Future Directions 83

Chapter 1

Introduction

Asset design represents one of the major processes in Computer Graphics applications, playing an important role in digital art fields as well as in architectural design and the entertainment industry. In recent years, the demand for high-quality assets started to grow rapidly, due to the fast expansion of design applications and the progress in high performance hardware systems. 3D assets are commonly used to represent objects, subjects, or environmental elements in virtual scenes, with their final appearance being described in terms of Spatially Varying Bidirectional Reflectance Distribution Function (SVBRDF) materials and textures, which determine their base color or physical properties like roughness or specularity in common rendering pipelines. However, the workflow encompassing the creation of such assets is still demanding in terms of time and human resources. In fact, the pipeline adopted by artists to create a high-quality asset is usually costly and it is often challenging for them to match a specific design. Although the creation of assets from scratch is still a possible approach, many techniques developed to support users in the synthesis and editing process.

For non-procedural asset synthesis, techniques such as 2D or 3D captures have evolved through the years to facilitate content creation. Even though they provide a good basis for further edits, they are controllable in a complex manner, still requiring artists to manually adapt them to reach a desired target appearance. In recent years, generative applications have been adopted in the field of digital art, providing the possibility of expressing a textual or visual constraint to better guide content generation. However, textual input is frequently not sufficient to describe content, whereas image conditioning may aid in the generation process but it still may fail in reconstructing the exact style, specifically in the case of highly structured contents or minute details.

On the other hand, procedural models for materials, textures, and 3D shapes define a family of assets, allowing for significant variations in the design of each

example by assigning different values to the exposed set of parameters. Although this approach reduces the degrees of freedom and gives users more control over the final design, parameter editing still requires a significant amount of resources. In particular, the editing process usually involves tweaking the parameter values using separate GUI sliders and understanding the impact of a parameter change over the final design may still necessitate some training. Furthermore, as the number of exposed parameters increases, it may become more difficult to determine the appropriate assignment for each parameter in order to achieve the desired appearance. Example-based and direct manipulation applications have developed through the years to make this process even easier. The former approach requires artists to provide a render or a sketch as a target, while the system identifies a possible parameter value assignment that better matches the provided design, whereas the latter allows for asset manipulation directly from the viewport, with a system recomputing the correct parameter update that matches edits that can be performed using a click-and-drag or stroke interaction. It is also possible to use a hybrid of the two approaches, firstly identifying a first parameter assignment using an example-based parameter estimation module and then applying local modifications through direct interaction.

This thesis, entitled **"Enhancing Controllability in Procedural and Non-Procedural Asset Editing"**, explores 2D and 3D asset design, reviewing previous approaches from the fields of inverse procedural modeling and texture synthesis, and proposes novel techniques that aim at simplifying the creative design workflow by providing more controllable and yet precise editing tools for both novice and experienced users. In the following section, we will examine more closely the contributions made by each work proposed in this thesis.

1.1 Contributions

This thesis explores different approaches for procedural and non-procedural asset editing, aiming to ease the modelling process by *enhancing control* for both novice and proficient users and *lifting* their workflow by reducing time and resources spent in cumbersome editing processes.

Procedural Assets. In the context of procedural assets, this thesis focuses on example-based and direct manipulation-based interactions, optimizing for the procedural parameters exposed by the asset to find the values that more accurately match the user input, both in terms of visual targets provided in the form of images or sketches and and direct in-viewport interaction.

pOp proposes an example-based approach for estimating the values of the

procedural parameters exposed by a procedural program describing a vector pattern. More in detail, *pOp* handles procedural vector patterns, which are a collection of vector shapes that follow a structured arrangement. By exploiting the end-to-end differentiability of such procedural programs with respect to the procedural parameters, *pOp* performs a gradient descent-based optimization, minimizing the difference between the example or sketch provided by the user and the generated vector pattern. Instead of using an inverse rendering definition of the optimization problem, *pOp* measures the pattern difference in the pattern signed distance field domain, thus introducing a new *OnTop* operator that mimics the overlap between vector shapes and a differentiable definition of a pattern SDF. The example-based parameter optimization framework proposed by *pOp* enabled the estimation of procedural parameter values given a render or a sketch, paving the way for a more controllable interaction between artists and procedural vector patterns content.

The framework proposed in *pOp* is then complemented by **pEt**, which proposes a direct manipulation method to edit the parameters of the same procedural vector pattern programs. Users can now perform edits on the pattern directly in the viewport via a click-and-drag interaction schema. Artists can use mouse movements to express transformations over a set of points, as well as constraints, which prevent other points from moving. Thanks to a point identification schema that enables *pEt* to locate the same point from the procedural parameter assignment, transformations in the selected point positions are translated into procedural parameter updates via a gradient descent-based optimization loop, minimizing the distance between such points and the transformed ones. While *pOp* performs a more global parameter optimization in identifying a good parameter assignment starting from an image or a sketch, local adaptations can be performed using *pEt*, whose combination of approaches enabled a full editing architecture for procedural vector patterns under the assumption of differentiability.

Although previously assessed methods are adopted in the fields of procedural vector patterns, similar techniques can be used to improve the editability of other procedural assets, such as implicitly defined surfaces. In this thesis, a **Procedural Implicit Surface Direct Manipulation tool** is also proposed to circumvent the laborious process of editing implicit surfaces, which involves tuning a large number of sliders, for which the semantics or parameter interconnection may be unclear. By proposing an in-viewport editing approach for implicit surfaces, we allow users to perform edits on the surfaces by clicking and dragging patches of the surfaces or constraining others to prevent them from moving. As the user performs the edits, our optimization tools perform a gradient descent-based optimization loop for each frame, computing an update for the procedural parameters for an end-to-end differentiable definition of procedural implicit surfaces. Leveraging an

ad-hoc co-parameterization designed for implicit surfaces, our system can identify the same point location throughout an edit, allowing parameter optimization by estimating the impact of a procedural parameter at a given point. The proposed framework supports a wide range of operations commonly involved in implicit surface modeling, including boolean and smooth boolean operations, affine transformations, warping operations, and also being resilient to topology changes.

Non-Procedural Assets. On the other hand, in the context of non-procedural assets, this thesis proposes a method to facilitate the synthesis of visually accurate textures with highly structural details and arrangements, oppositely to non-stochastic and natural texture synthesis methodologies that have been widely proposed throughout the year. In **pAff**, we leverage the generative capabilities of stable diffusion models, finetuned on the pattern domain by incorporating a Low-Rank Adaptation (LoRA) model. Given a small sketch of a pattern, our architecture enabled the generation of large-scale and high-quality content, that presents a high fidelity in terms of structure, colors, and fine-grained details with the provided sketch. Leveraging novel techniques such as *Noise Rolling*, our architecture performs the expansion of a user-drawn input to an arbitrarily sized canvas and ensures pattern tilability, paving the way for further application and enabling a higher level of control over Latent Diffusion Models in the context of asset generation.

In conclusion, the approaches proposed in this thesis may be adopted to enhance the controllability of other assets different from 2D structured patterns and 3D implicit surfaces, such as SVBRDF materials, thereby providing assistance in several cumbersome asset design workflows.

After a first review of the works that relate the most to the domain studied in this thesis Chapter 2, we will proceed in digging into details about procedural asset editing in Part I, exploring both example-based and direct manipulation based editing for 2D and 3D assets, and non-procedural 2D asset editing in Part II. In conclusion, this thesis will suggest future directions for improving controllability or adapting proposed methodologies to other asset types.

Chapter 2

Related Work

This chapter reviews relevant works concerning the editing and controllability of procedural as well as the guided synthesis of non-procedural ones. Specifically, Section 2.1 will review the literature related to inverse procedural modeling, with an explicit focus on example-based approaches (Section 2.1.2) and direct manipulation methods (Section 2.1.3) on both 2D and 3D content. Lastly, Section 2.2 will cover related works on non-procedural texture synthesis and control, with a particular attention to the ones based deep learning architectures and generative models.

2.1 Inverse Procedural Modeling

With the term *inverse procedural modeling* we usually refer to the problem of finding the procedural description for 2D or 3D assets like materials, textures or shapes that better matches a user-provided target. 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. Although this thesis mainly focuses on approaches that aim at modifying the parameters of a procedural model and not its procedural rules, the following sections will cover in detail both problems as well as the evolution of the proposed solutions through the years. Alongside with the discussion about relevant works, particular emphasis is also given to the contribution of the methods proposed in this thesis for all concerning tasks.

Firstly, procedural program estimation is reviewed by proposing several approaches that aimed at estimating or controlling the definition of 2D or 3D assets like Bèzier curves arrangements, trees, façades or buildings in terms of procedural programs or graphs. Then, focusing only on procedural parameter optimization, both example-based and direct manipulation methods for textures, materials and 3D shape editing are reviewed. In example-based approaches, users provide a target,

usually in the form of an render or sketch, that is used to compute the procedural parameters that better match the given image. On the other hand, in direct manipulation approaches transformations are directly performed on procedural assets with click-and-drag or stroke interactions, with the optimizer recomputing the parameter values that best fit the constraint expressed by the user. 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.

2.1.1 Procedural Program Estimation

The work of [Šta+10] represents one of the first works in 2D procedural program estimation, which will be assessed in this section in conjunction with the 3D counterpart. It propose a method for objects made up of lines or Bèzier curves, providing 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. [Van+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 [Šta+14; Du+18], knitwear [Tru+19], facades or buildings [Wu+13; Zhu+15; Mül+07; NBA18] or the reconstruction of animated sequences [PLL11]. Similar works are the ones by [Sha+18; Lip+19; Guo+20; Krs+20; Rit+15], that propose methods for estimating procedural modeling programs or globally and locally controlling them. Lastly, a detailed overview of the inverse procedural modeling of 3D models for virtual applications is exposed by [Ali+16]. While all these works are examples of inverse procedural modeling, they address the problem of estimating the entire procedural program rather than its parameter values, which in turns is the main focus of the procedural asset editing works proposed in this thesis.

2.1.2 Procedural Parameter Estimation via Examples

In example-based techniques, users provide images like renders or sketches depicting the final design of the asset, being it a texture, material or 3D shape. Given the desired appearance, different approaches developed through the years to find an assignment to the procedural parameters that better match the input. Although

parameter estimation from examples can be applied to different procedural assets, particular attention is given to procedural materials, since usual design workflows encompass materials and textures synthesis by manipulating the parameters of the procedural generator. However, when the number of parameters is high, artists may struggle in finding the parameter values to obtain the desired appearance. This issue has started to be addressed recently for raster textures like in [Guo+19], that 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. [Shi+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 in turns is one of the core contributions of this thesis. [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-procedural style transfer. In [Hu+21b], 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 multi-layer 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. [Gue+22] propose a generative model for procedural materials that are represented as node graphs, and let users to auto-complete those graphs too, while [Hu+22a] make complex node differentiable by using neural network proxies. However, many of these works rely on training Neural Networks to estimate a parameter initialization, thus needing to collect data and performing an offline time-consuming training process, that may also be iterated if new assets need to be supported. Leaving aside raster textures and focusing on the field of vector graphics and patterns, the work of [Li+20] 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. [Red+20] show how to support compositing operations by proposing a method to differentiate

them, in the context of vector patterns. The example-based approach proposed in this thesis, namely *pOp*, focuses on procedural and differentiable vector patterns by estimating their parameters using an gradient descent optimization that minimizes a differentiable SDF-based loss function. Such approach will be discussed more in detail throughout Chapter 3.

2.1.3 Procedural Parameter Estimation via Direct Manipulation

The idea of direct manipulation has been previously explored in many 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. [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. [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. [Gue+16] 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. [Jac+11] 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. Editing procedural shaped directly from screen-space user edits is is applied also to simple joint structures [BM96; AL10] and in the field of Inverse Kinematic, for which [Ari+18] provides a comprehensive survey.

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 [Sch+14].

In the case of procedural 3D meshes, a few papers proposed techniques that allow in-viewport editing. The pioneer work from [Gle94] investigated graphical

interfaces for the direct manipulation of 3D shapes. For procedural 3D CAD models, a bidirectional approach is proposed in [Cas+22]. 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. [Gai+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. However, a key limitation of the two latter methods is that they respectively rely on bounding boxes and mesh representations, and thus do not support operations that result in topological changes. [MB21] demonstrates 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 and provides support for topological changes operations as well.

The work of [MB21], combined with the rendering-based approaches of [PTG02; Pel10] opened up to different direct manipulation based applications in both 2D vector patterns domain as well as implicitly defined surfaces. As regard the former, Chapter 4 will cover *pEt*, a direct manipulation approach for differentiable vector patterns. Oppositely to the work of [MB21], *pEt* is based on automatically differentiated procedural programs, whose parameters can be manipulated by clicking and dragging points from both the shapes' interior and exterior. As regards 3D implicits, the work exposed in Chapter 5 focuses on the overlap of two research fields, namely the editing of procedural shapes or the editing of implicit surfaces, aiming at directly controlling procedural shapes defined as implicit surfaces and inheriting the topology changes resiliency from previous by redefining it for the implicit domain. In fact, in the context of parametric implicit surfaces, the assumption that no change of topologies can occur is impracticable as Constructive Solid Geometry (CSG) operations are widely used to construct complex models. All the methods relying on the parametrization provided by meshes cannot be directly applied to implicits because they have no way to track the evolution of a point on the shape after a change in the procedural parameters.

The following paragraph sets aside discrete implicit representations that are not parametric and focuses solely on procedural implicit surfaces. On top of providing control over affine transformations through 3D Gizmos in modeling software [Wom22; Mag22], previous research on implicit modeling has focused on providing indirect and direct control to the user. The work from [Sch+06] shows an interactive modeling application where the user sketches 2D contours that are interpreted

as new primitives in a BlobTree model [WGG99], which can then be combined with CSG operators to create complex shapes. Despite the system's expressiveness, editing of the parameters is still done indirectly through manual tuning of sliders. Limited to affine transformations, the work of [BGA05] shows how to animate a BlobTree model by defining the procedural parameters values as a function of time. Warp curves [Sug+08; SWS10] are another alternative for editing implicit surfaces in which the user draws and manipulates polylines which locally deform the implicit surface using variational warping. Modifications are however limited to space-deformations, which are computationally intensive, and are not propagated back to the procedural parameters of the shape. Direct manipulation of implicit surfaces have been investigated for blending operators, with new ones that either improve topological control through new parameters [ZCG15], or that match a 2D drawing of the intended blending behavior [Ang+17]. The method exposed in libfive [Kee19], a library for implicit modeling, provides some surface manipulation capabilities. The key limitation is that it only allows the user to specify where there must be some element of surface, not to specify which element exactly. In practice, this limits the user to only inflation and translations operations and without handling other operators like affine transformations and complex warping such as twisting.

2.2 Non-procedural Asset Control

Non-procedural asset editing still aims at producing an asset that is coherent with an user specification, being it a textual definition or a visual suggestion representing a specific style or content. Oppositely to procedural asset editing, it does not aim to produce an appropriate representation for a given content, both in the terms of procedural programs or parameters. So, 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. This section covers relevant works assessing 2D asset synthesis, going from classical texture synthesis approaches up to more novel applications involving deep architectures and generative models.

Texture Synthesis. At first, example-based non-parametric texture synthesis has been explored by [EL99] as regards per-pixel approaches and by [EF01] for per-patch [EF01] stochastic ones, followed by [Kwa+05], that proposes an optimization method that refines the entire texture, and many other techniques reported in the comprehensive survey of [Wei+09]. [GEB15] proposed a method that aims to create a texture exploiting Convolutional Neural Networks by extracting and combining features at different levels, using this stationary representation to learn a new texture from noise. Recently, the work of [Zho+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. On the pattern domain, [Bar+06] and [Hur+09] aim at synthesizing patterns by analyzing elements and properties from reference vector pattern provided by users. [Iji+08] also allow users to specify a local growth area as a constraint to the synthesis process.

Lastly, [Tu+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. [Ma+13] propose a method for generating spatio-temporal repetitions based on a combination of a constrained optimization and a data-driven computation, while [Rov+15] explores repetitive structure synthesis using discrete elements or continuous geometries. A more comprehensive review of the example-based methods can be found in [Gie+21]. [Guo+19; Hu+22b; Zho+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.

Generative models. Image generation is a long-standing challenge in computer vision due to the complexity of visual data and the diversity of real-world scenes. With the advent of deep learning, the generation task has been increasingly posed as a learning problem, with Generative Adversarial Networks (GAN) [Goo+14] enabling the generation of high-quality images [Kar+17; BDS18; Kar+20]. However, GANs are characterized by an unstable adversarial training [ACB17; Gul+17; Mes18], and unable to model complex data distributions [Met+16], exhibiting a *mode collapse* behavior.

Recently, Diffusion Models (DMs) [Soh+15; HJA20; Rom+22] have emerged as an alternative to GANs, achieving state-of-the-art results in image generation tasks [DN21] also thanks to their stable supervised training approach. Furthermore, DMs have enabled a whole new level of classifier-free conditioning [HS22] through cross-attention between latent image representations and conditioning data. More recently, ControlNet [ZRA23] has been proposed to extend generation controllability beyond the typical global-conditioning (e.g.: text prompts) for a fine control over the generation structure. Moreover, approaches like DreemBooth [Rui+23] and LoRA [Hu+21a], allow users to customize large-scale pre-trained models, adapting them to particular tasks or domains or domains, without needing to finetune them and requiring only a limited set of training samples. Regardless of these improvements, however, Diffusion Models tend to be slower and more computationally demanding than GANs, limiting their application in interactive environments. To close this gap, recent approaches have focused on improving the inference performances of

DMs, reducing the number of required diffusion steps from tens or hundreds to just a few steps [Son+23; Sau+23], while other approaches have been proposed to enable high-resolution generation with limited resources [Jim23; Bar+23; Vec+23], achieving 4K or higher resolution.

Generative models for textures synthesis. Several work have assessed the synthesis of patterns in the form of natural textures or BRDF materials involving generative models as their backbone. [Hei+21] address the problem of texture synthesis via optimization by introducing a textural loss based on the statistics extracted from the feature activations of a convolutional neural network optimized for object recognition (e.g. VGG-19). [Vec+23] recently introduced ControlMat to perform SVBRDF estimation from input images, and generation when conditioning via text or image prompts. It employs a novel *noise rolling* technique in combination with patched diffusion to achieve tileable high-resolution generation. MatFuse [Vec+24], on the other hand, focuses on extending generation control via multimodal conditioning and editing of existing materials via *volumetric inpainting*, to independently edit different material properties.

Focusing on non-stationary textures, [Zho+23] introduces a new Guided Correspondence Distance metric that can be employed as a loss function to optimize the texture synthesis process, improving the similarity measurement of output textures to examples. [Zho+24], in contrast, leverages a diffusion model backbone combined with a two-step approach and a *"self-rectification"* technique, to generate seamless texture, faithfully preserving the distinct visual characteristics of a reference example. Although the quality and level of realism of textures of BRDF materials generated via these methods is impressive, they mainly focus on unstructured, natural and non-stochastic content generation with only a few of them explicitly focusing on structured content. This scenario paved the way for structured patterns generation via an expansion process using diffusion models, as covered in Chapter 6.

Part I

Editing Procedural Assets

Chapter 3

pOp: Parameter Optimization of Differentiable Vector Patterns

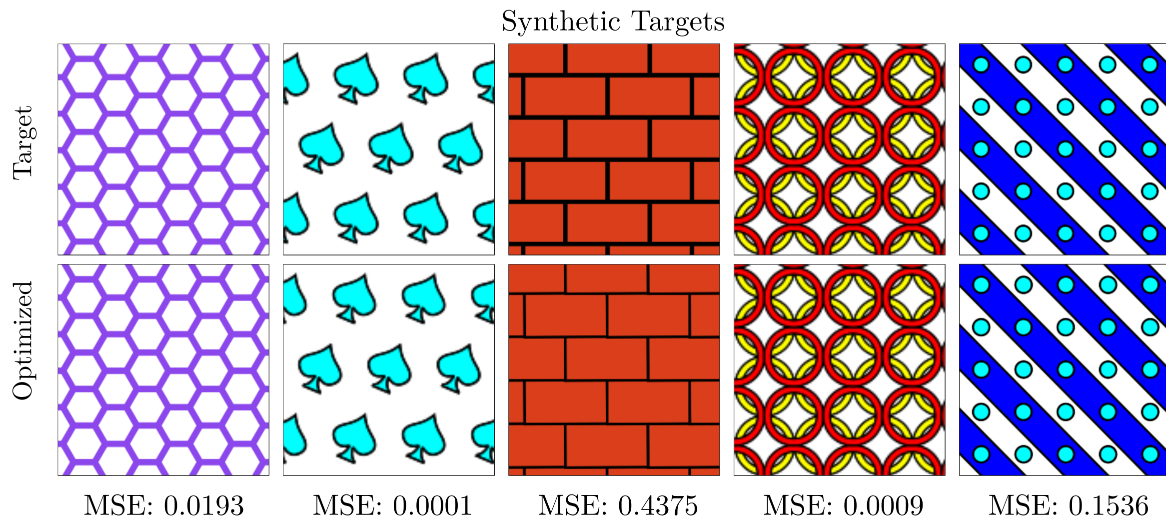


Figure 3.1. *pOp* finds the parameters of procedural vector graphics patterns that best match target images. We support patterns comprised of standard vector graphics elements, e.g. circles, rectangles, lines, and quadratic Bèzier curves, where the translation, rotation and scale of the elements is defined 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.

In this chapter 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

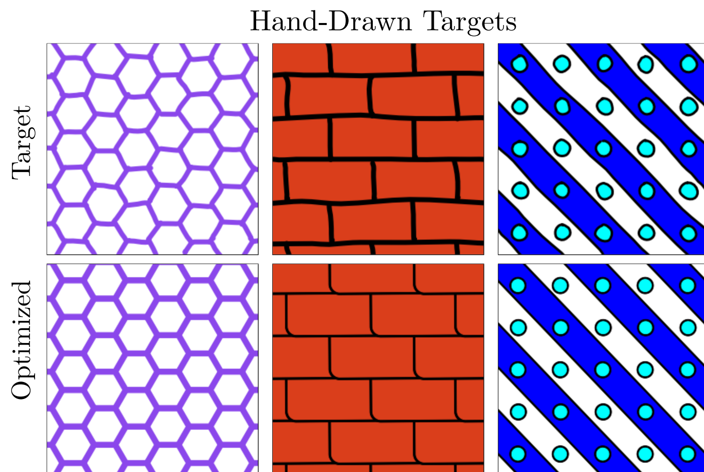


Figure 3.2. We tested *pOp* framework on hand-drawn inputs, mimicking a possible design application. Given a user-sketch, *pOp* is capable of identifying the parameters that better match the input, thus producing a visually similar result.

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.

3.1 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; Guo+19; Shi+20; Hu+21b], use optimization of differentiable textures, optionally combined with neural networks and texture synthesis to provide a solution in this domain.

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, [Li+20] 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 3.1 and 3.2 throughout this chapter, 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 3.3.

3.2 Method

3.2.1 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

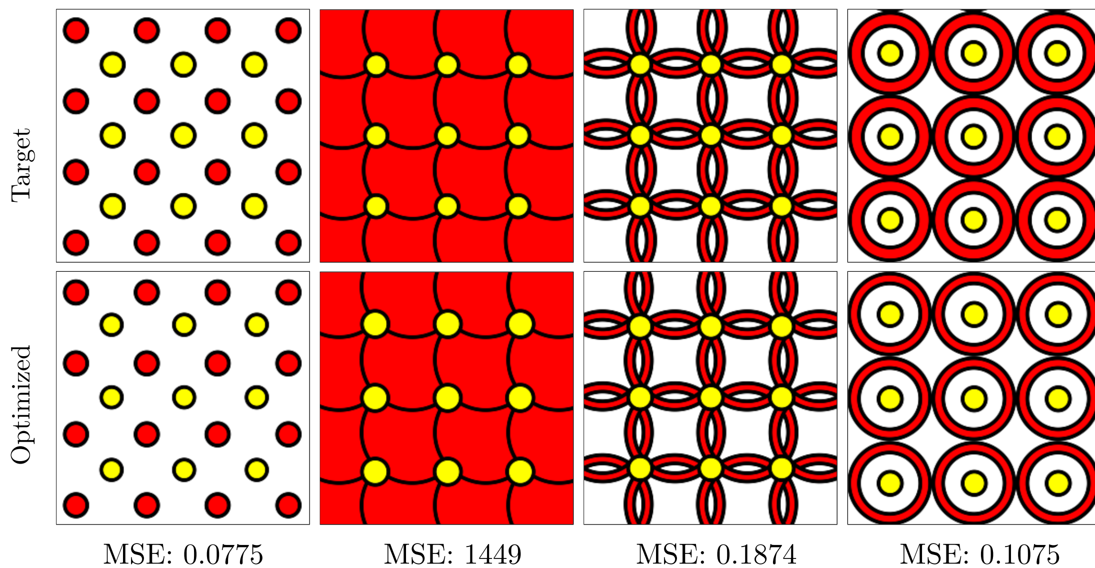


Figure 3.3. The same procedural program can create visually-distinctive patterns. Our algorithm optimizes all pattern variations.

color, stroke color, and stroke thickness. However, in this 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; Guo+19; Shi+20; Hu+21b]. 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.

$$\Phi^* \;=\; argmin_\Phi \; \mathcal{L}(I, G(\Phi)) \tag{3.1}$$

Prior work on inverse procedural materials uses differentiable renderers combined with gradient-based optimization to estimate the procedural parameters [Guo+19; Shi+20; HDR19; Hu+21b]. In the context of vector graphics, [Li+20] present a

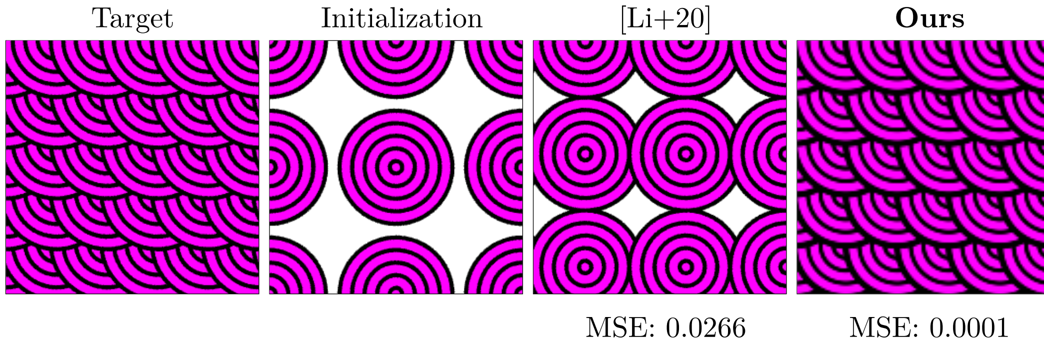


Figure 3.4. When trying to optimize a pattern starting from the same initial configuration, we found that losses based on image difference, and relate differentiable renderers such as [Li+20], do not work well in our problem domain. We instead define a loss based on pattern SDFs that is robust to all pattern variations.

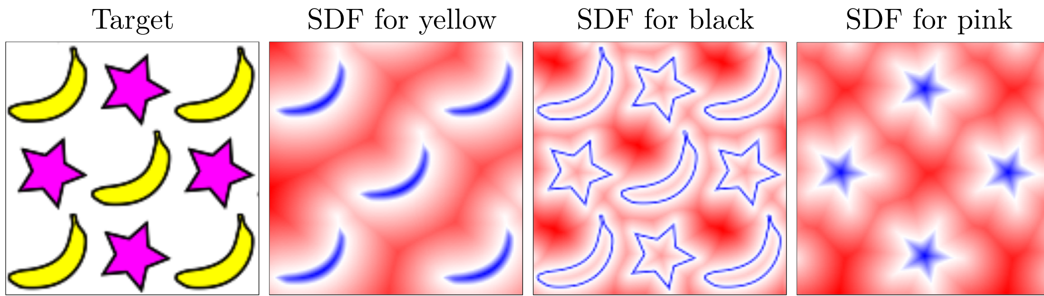


Figure 3.5. Colored patterns are supported by computing an SDF for each color. The loss function is the sum of the contribution corresponding to each color. Here we display SDFs from blue to red for negative to positive values respectively.

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 3.4, 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 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.

3.2.2 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 3.5. 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:

$$\mathcal{L}_S = \frac{1}{|C|} \sum_{c \in C} \mathcal{D}(S_c(I), S_c(G(\Phi))) \tag{3.2}$$

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 3.6.

[Smi+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 [Smi+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:

$$\mathcal{L}_N = \frac{1}{|C|} \sum_{c \in C} \left[1 - \langle \nabla S_c(I), \nabla S_c(G(\Phi)) \rangle^2 \right] \tag{3.3}$$

where gradients are normalized. The final loss $\mathcal{L}$ between a target image and the

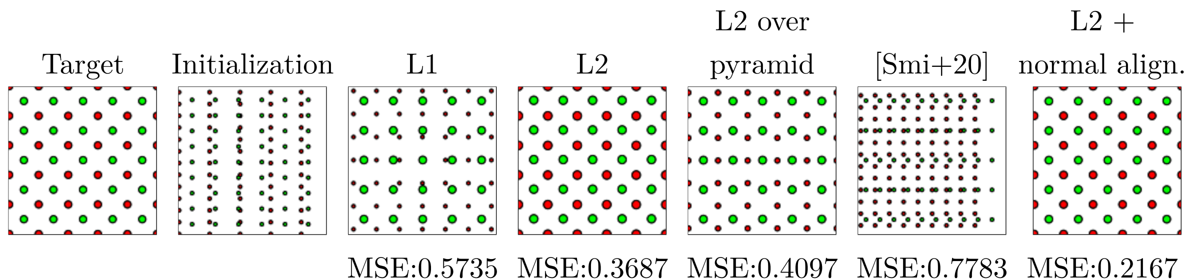


Figure 3.6. Comparison of the results obtained with different loss functions, reporting both the pattern image and the MSE of the optimized pattern parameters, where all parameter sets are optimized from the same initialization. We compare L_1, L_2, L_2 over a gaussian pyramid, the metric proposed in [Smi+20], and the L_2 distance with normal alignment. The latter loss was chosen since it works best in our tests.

procedural pattern is the weighted sum of both terms, written as:

$$\mathcal{L} = \mathcal{L}_S + \alpha_N \cdot \mathcal{L}_N \tag{3.4}$$

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 [Smi+20]. In our implementation, we approximately convert cubic Bèzier to quadratic ones, since the latter have simpler and numerically-robust distance formulas.

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 3.7. 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 3.8.

Let us consider the operation of drawing a shape onto a background. Since we

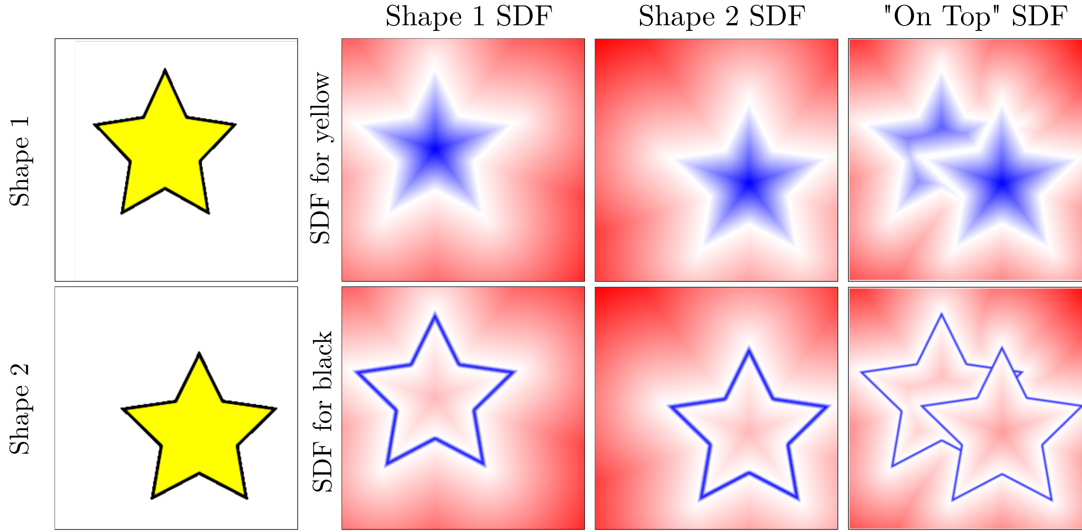


Figure 3.7. SDF corresponding to the operation of drawing a shape onto another, compared to the SDFs for the single shapes.

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:

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

Figure 3.7 shows an example of combining two shapes with the same stroke and fill colors, resulting in the last two cases described here.

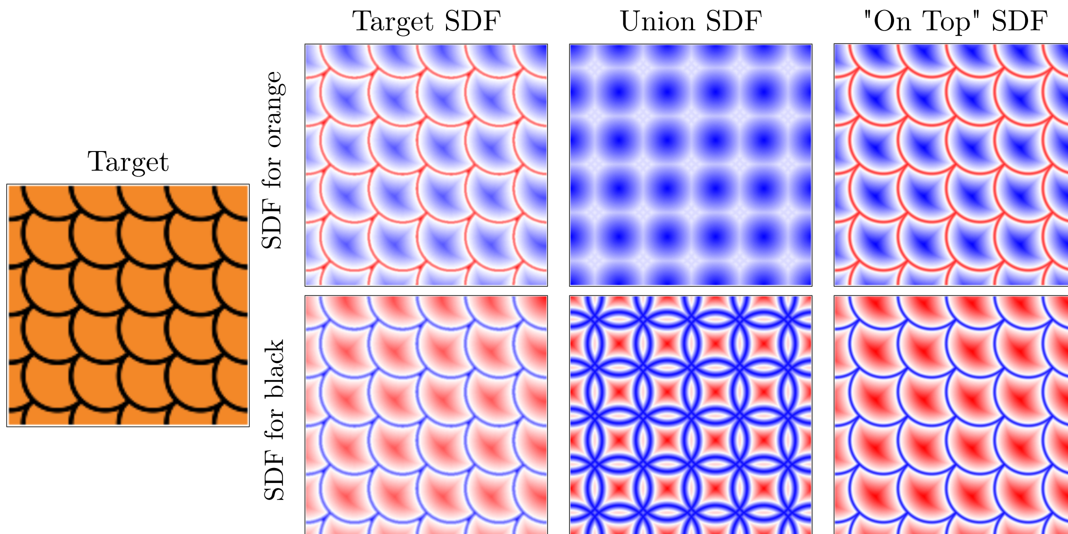


Figure 3.8. Comparison between combining overlapping shapes with boolean unions or with our operator that captures drawing shapes on top of one another. While the union operator captures shapes correctly, it cannot represent shapes that have both stoke and fill color. Our operator matches exactly the SDFs corresponding to the stroke and fill colors extracted from the target image.

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 [Pas+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 3.9. 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.

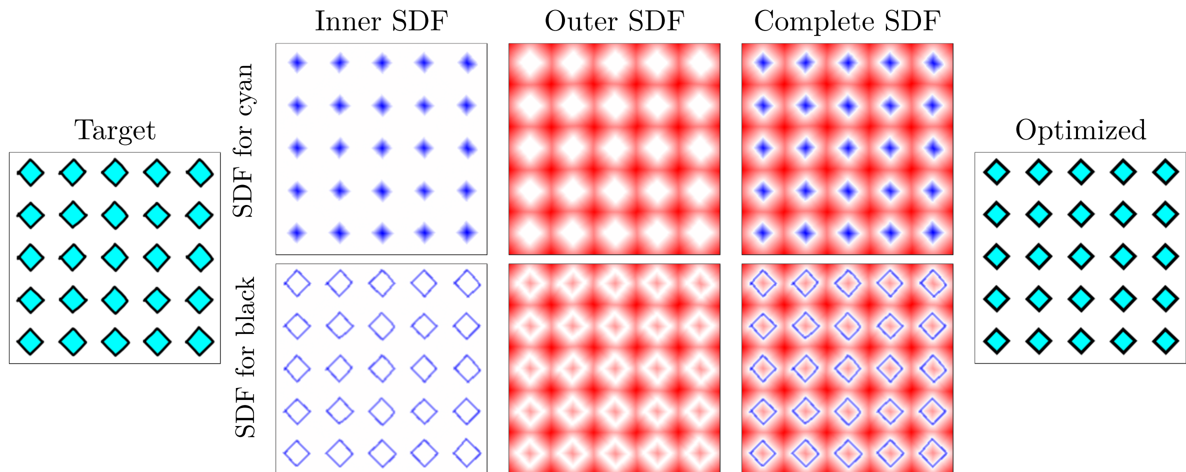


Figure 3.9. SDF extracted from an hand-drawn target image. For each image color, we extract a binary mask that is used to compute both inner and outer distance using the Euclidean distance transform, flipping the mask values accordingly.

3.2.3 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 3.1. 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 3.10.

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

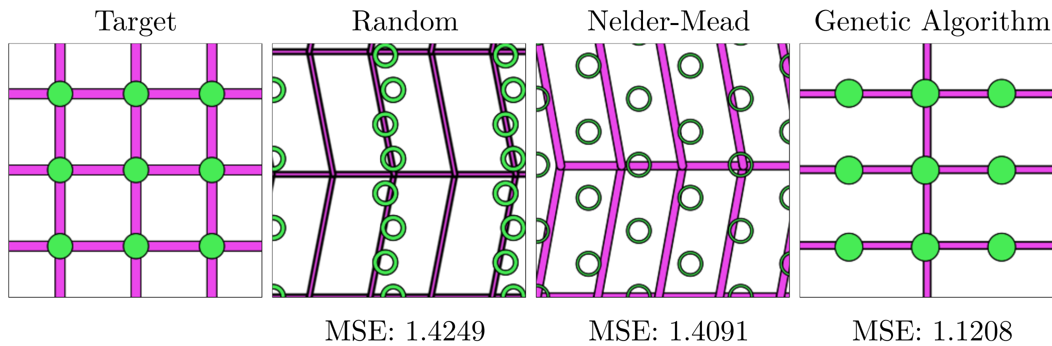


Figure 3.10. Comparison between the parameter initialization procedures. Between random sampling, Nelder-Mead's simplex and genetic algorithms, the latter was selected since it works best for our problem.

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 3.10 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

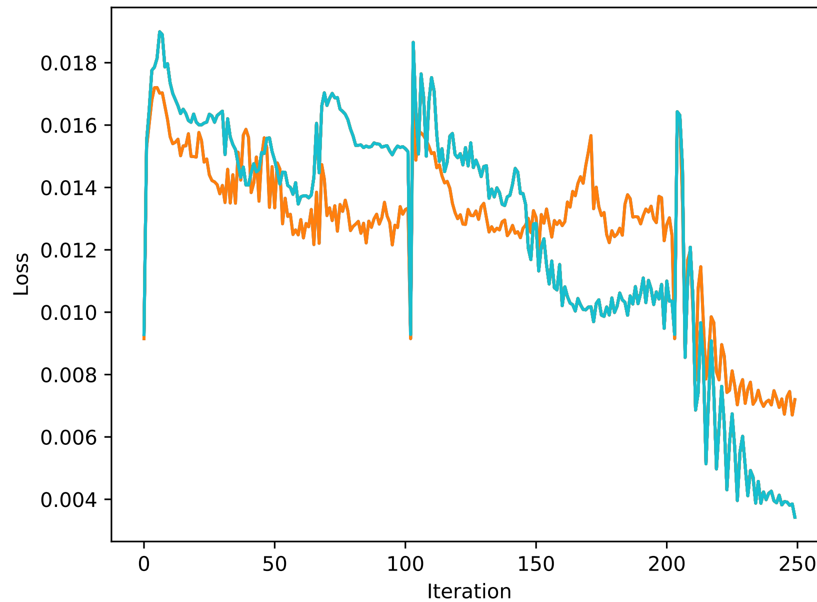


Figure 3.11. Loss changes during optimization of the stripe pattern in Figure 3.1. In our approach, we chose to optimize starting from multiple starting configuration, to reduce the chance of local minima, and pick the best final one. We use 10 candidates in our, but show here only 2 for clarity of presentation.

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 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

Figure	Generator	Parameters	Shapes	MSE
Fig. 3.1	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. 3.3	Circles2	14	25	0.0773
	Circles2	14	25	0.0004
	Circles2	14	25	0.0015
	Circles2	14	18	0.0025
Fig. 3.4	Shingles	6	30	0.0001
Fig. 3.6	Circles2	14	61	0.2167
Fig. 3.8	Circles3	7	36	0.0018
Fig. 3.9	Rectangles	8	25	0.3058

Table 3.1. Statistics of the optimized patterns, including number of parameters, number of shapes, and MSE of the fitted parameter set.

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 3.11.

3.3 Results

Throughout this chapter, 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 were fit with the same optimization setting 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 [Pas+19] and SciPy [Vir+20] for optimization and automatic differentiation. The Genetic Algorithm based initialization is implemented using the

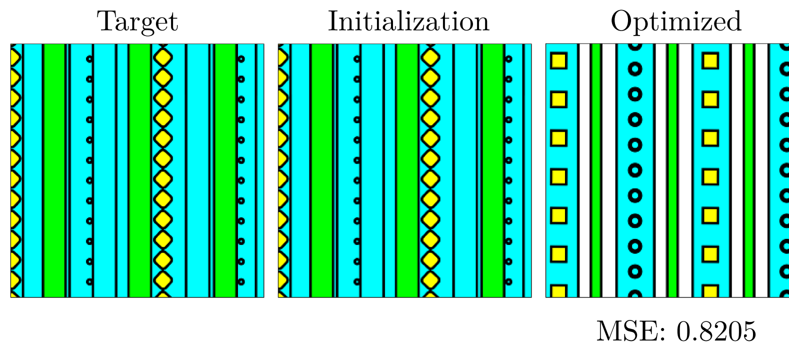


Figure 3.12. For patterns with high number of parameters, 25 in this figure, the algorithm may converge to a local minimum.

DEAP [For+12] framework.

Table 3.1 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 3.12, 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 3.3. Overall, we found our method to work well in all these cases.

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. [Red+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 [Shi+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.

3.4 Conclusions

This chapter introduces *pOp*, a method for computing the parameters of procedural vector patterns that match a given input image. The key idea of this 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.

Chapter 4

pEt: Direct Manipulation of Differentiable Vector Patterns

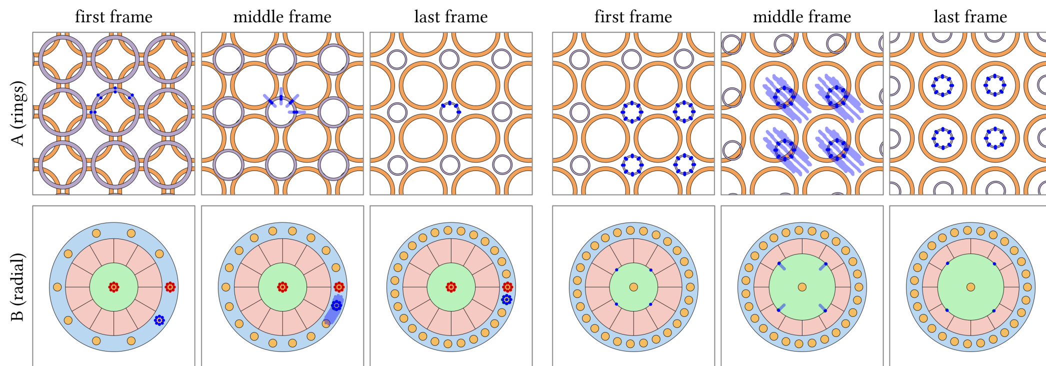


Figure 4.1. We propose *pEt*, an interactive method to edit the parameters of procedural programs that generate vector patterns, where users interactively transform a set of points and constrain other ones to fixed locations. During the interaction, we solve for the procedural parameters with a gradient-based 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.

In previous chapter, we demonstrated how to estimate procedural parameters of procedural vector patterns, which are collections of vector primitives generated by procedural programs, starting from example images and sketches. Although it works well as a first estimation of good initial starting point values, other approaches like asset manipulation may be enable more precise edits, as the system interactively determines the procedural parameter values while the edit occurs. In this chapter we complement such approach by proposing *pEt*, a method for editing procedural vector

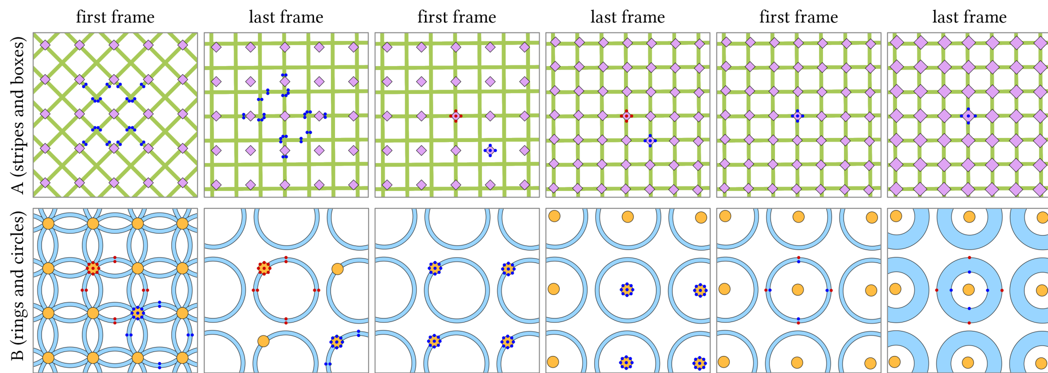


Figure 4.2. The same procedural program can produce significantly different results by changing just the parameters assignment. Here two examples program are edited to produce three variations. For each variation, we show the first frame and the last frame of the optimization procedure.

patterns using direct manipulation. In this approach, 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.

4.1 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 latter, considering that large libraries of programs are readily available [ADO]. 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; Li+20; Shi+20; Red+20; Hu+21b; Hu+22a]), 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; Cas+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.

With our framework *pEt*, we propose 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 4.1 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 *pOp*, exposed in Chapter 3.

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]. We demonstrate that this simple change is sufficient for users to guide the editing precisely (see Figure 4.2 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, as shown in all the images of the current chapter. We consider edits that alter the shape positions, the pattern symmetries, the number of shapes and the deformation

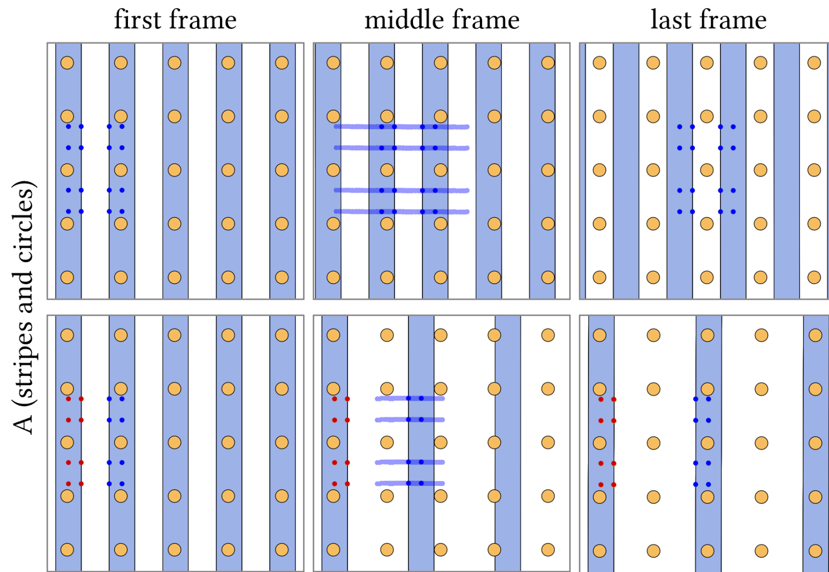


Figure 4.3. Different edits can be expressed by selecting different points and either transforming or fixing them. (Top) The stripes are translated by transforming a set of 16 points. (Bottom) The distance between stripes is edited by transforming a set of 8 points, while fixing another 8 points.

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.

4.2 Method

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 4.2, but as the number of parameters increases, the edits become very time-consuming. The approach proposed by *pOp* in Chapter 3 shows 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

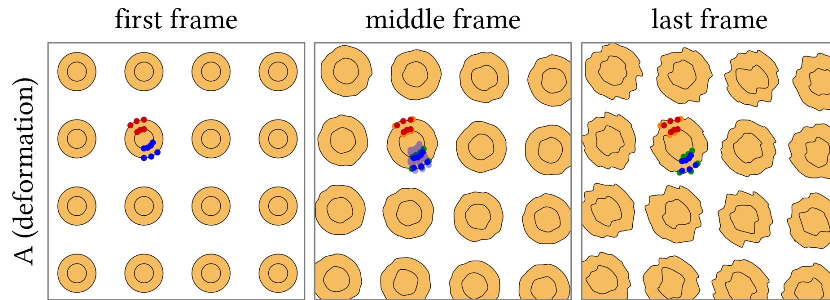


Figure 4.4. Procedural deformations, here obtained by applying noise functions to the shape outlines, are controlled in the same manner as other transformations.

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.

4.2.1 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 4.3 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 4.5.

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 4.8 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 4.5. 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

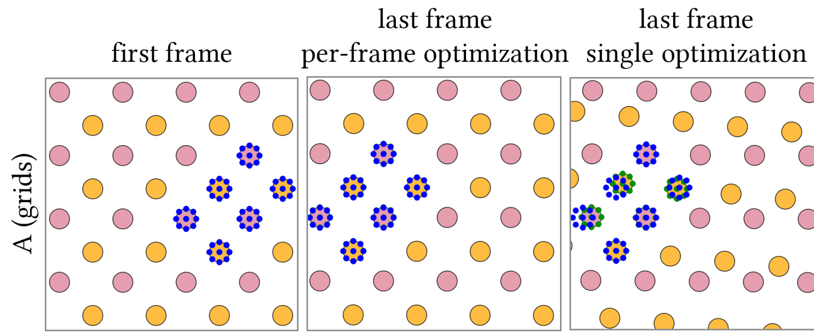


Figure 4.5. Besides providing interactive feedback while editing, optimizing procedural parameters per-frame gives better final results, compared to optimizing only once at the end of the interaction sequence (we compare the results with the same total number of iterations).

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 4.3 and as also visible in the remaining figures in this chapter.

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 4.4.

In our prototype, users transform points by applying an affine transformation, namely translation, rotation or scaling, to the points original positions, although additional transformations can surely be implemented. 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 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 4.6.

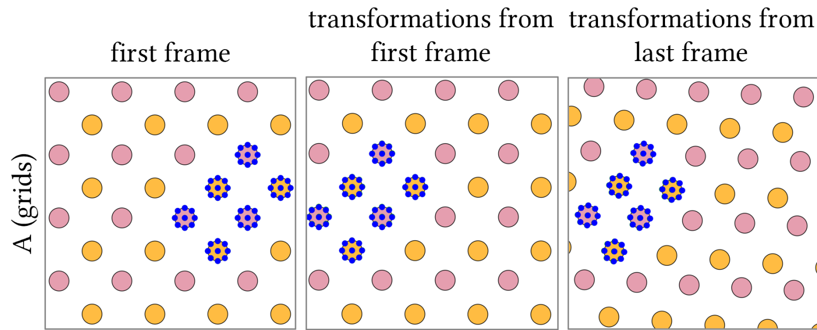


Figure 4.6. In our prototype interface, we apply transformations starting always from the initial, correct configuration, since we observed drifting of the solution if the transformations are applied to the configuration of the last frame. In this example, the resulting pattern shows an undesired pattern rotation, as well as a misalignment due to the updated grid spacing.

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

$$T^t p_s = \begin{cases} M^t p_s & s \in E \\ p_s & s \in F \end{cases} \tag{4.1}$$

4.2.2 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 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

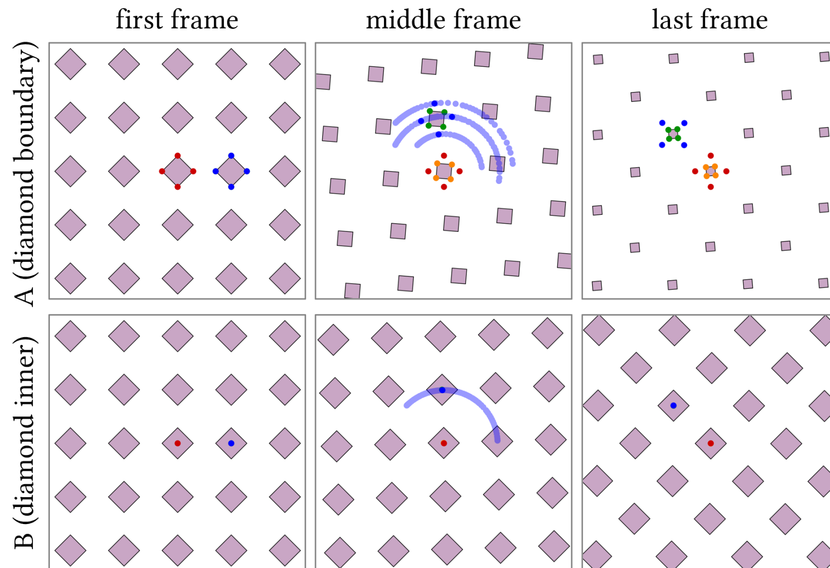


Figure 4.7. In this example, the user intent is to rotate the grid without rotating the individual shapes. To do so, the user needs to fix some points to disambiguate between rotation and say translation and scaling during interaction. (Top) Selecting only points on the shape boundary induces unwanted 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.

as shown in Figure 4.7. 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

$$\Theta^t = \text{argmin}_\Theta \frac{1}{N} \sum_s^N ||T^t p_s - f(i_s, \Theta^t)||^2 \quad (4.2)$$

$$\text{where } p_s = f(i_s, \Theta^t) \text{ for t} = 0 \quad (4.3)$$

4.2.3 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.

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 4.1 (B) or the number of elements and subdivisions in Figure 4.9 (H). On the contrary, discrete parameters such as

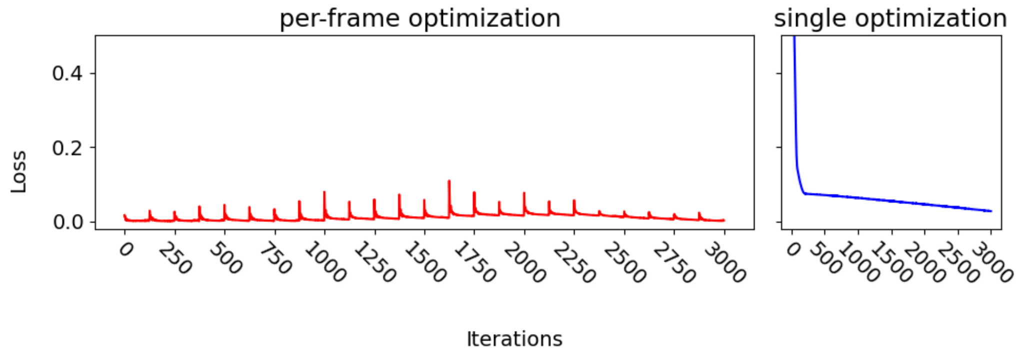


Figure 4.8. Comparison between per-frame optimization, shown left, and optimization performed only in the last frame, shown right, for the edit in Figure 4.5. Besides providing feedback while editing, per-frame optimization has a lower end loss (0.0016) than end-only optimization (0.0327), for the total number of iterations. Peaks in the per-frame optimization correspond to mouse events.

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.

4.3 Results

In this section, we collect the results obtained while editing a variety of procedural vector patterns, summarized in Table 4.1 and shown in all the figures of the current chapter. We performed all the tests on a machine with an AMD Ryzen 9 CPU with 3.4 GHz frequency. The edit sequences discussed here 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 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.

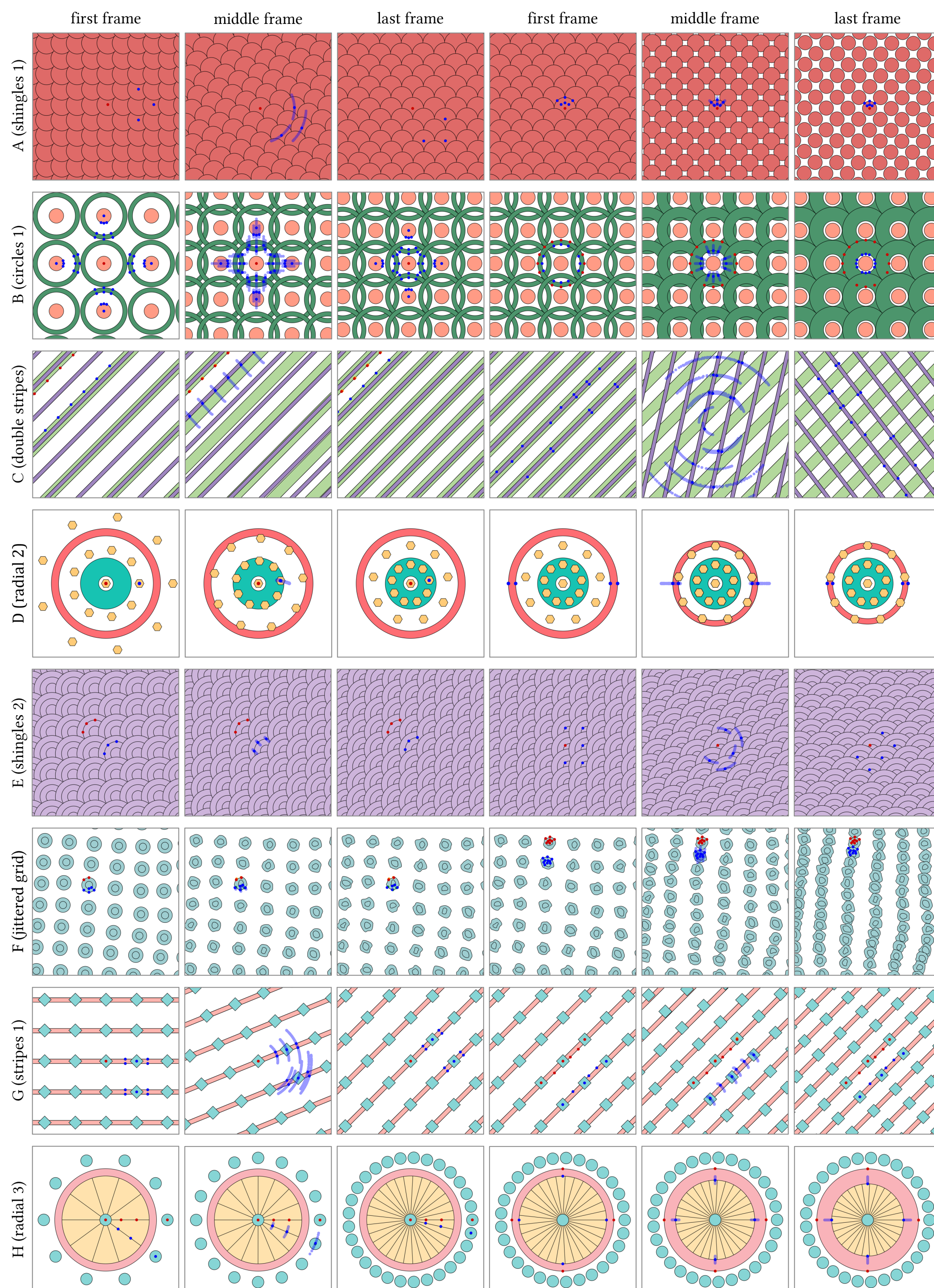


Figure 4.9. Gallery of edits on a variety of differentiable vector patterns such as grids, stripes, and radial patterns, with possibly occlusion, deformation and jittering. For each pattern, we show two consecutive edits.

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 4.1 (B) and Figure 4.9 (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 4.9 (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 4.2 (B–first edit), Figure 4.9 (B–first edit) and Figure 4.9 (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 4.2 (A) or Figure 4.3.

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 4.1 (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 4.4 and Figure 4.9 (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 4.9 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 4.1, 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 parameters throughout the mouse

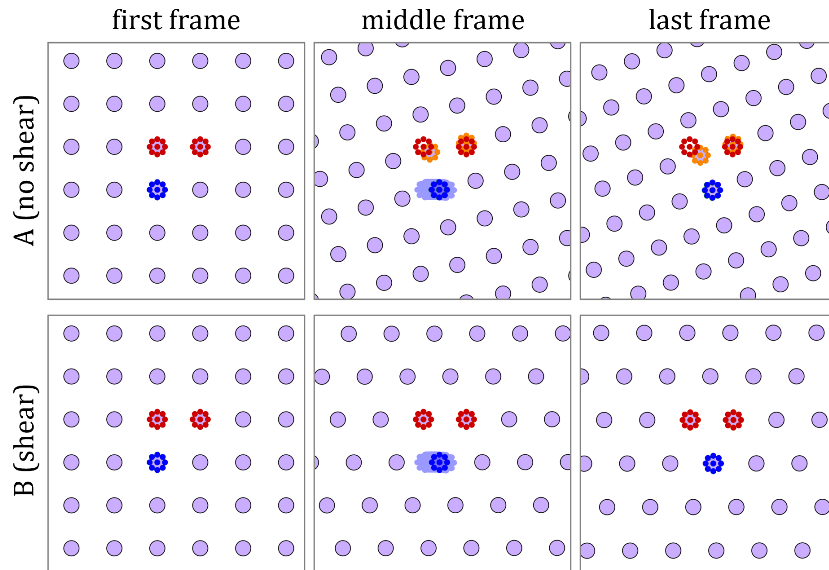


Figure 4.10. A limitation of our approach is that user edits can produce undesired results if the procedural program has no parameter combination that can fit user edits. In this figure, the user intent of shearing the grid ends in a rotation update in a procedural program that cannot express shearing (top), while it is correctly handled by a program that can model the edit (bottom).

interaction, thus proving users with precise feedback while editing.

4.3.1 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. 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 4.10, 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.

4.4 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.

Table 4.1. Statistics of the edits shown throughout the current chapter. 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 Num.	Num. Prms.	Num. Shapes	Transf. Points	Fixed Points	Mouse Events	Number of Iterations	Final Loss	Average MSE
4.7 B	9	25	1	1	42	25, 6, 42	0.0013	$2 \cdot 10^{-6}$
4.9 A	10	121	3	1	27	57, 24, 125	0.0019	$1 \cdot 10^{-6}$
	10	98	6	1	25	76, 21, 125	0.0013	$4 \cdot 10^{-5}$
4.9 E	10	49	3	3	42	53, 23, 87	0.0020	$2 \cdot 10^{-4}$
	10	84	5	1	28	97, 64, 125	0.0023	$6 \cdot 10^{-4}$
4.9 F	10	49	5	1	5	102, 20, 125	0.0083	$8 \cdot 10^{-4}$
	10	105	10	8	5	111, 42, 125	0.0016	$1 \cdot 10^{-4}$
4.4 A	10	16	7	6	18	119, 114 125	0.0216	$9 \cdot 10^{-5}$
4.9 C	14	20	6	4	36	117, 26, 125	0.0061	$7 \cdot 10^{-5}$
	14	20	17	0	39	104, 32, 125	0.0005	$1 \cdot 10^{-4}$
4.9 G	16	31	10	1	41	73, 29, 125	0.0005	$4 \cdot 10^{-5}$
	16	31	5	6	26	107, 84, 125	0.0005	$1 \cdot 10^{-4}$
4.3 A	17	31	16	0	45	65, 19, 119	0.0002	$2 \cdot 10^{-5}$
	17	18	8	8	44	51, 16, 105	0.0005	$3 \cdot 10^{-5}$
4.6 A	20	32	54	0	18	102, 84, 125	0.0051	$4 \cdot 10^{-4}$
4.9 B	20	50	40	1	17	123,108,125	0.0010	$1 \cdot 10^{-4}$
	20	50	12	13	18	27, 13, 40	0.0010	$1 \cdot 10^{-4}$
4.9 D	20	37	1	1	34	4, 2, 33	0.0023	$1 \cdot 10^{-6}$
	20	37	4	0	27	19, 3, 64	0.0025	$6 \cdot 10^{-6}$
4.9 H	20	59	3	4	11	35, 20, 56	0.0006	$4 \cdot 10^{-6}$
	20	59	4	4	15	34, 15, 125	0.0015	$6 \cdot 10^{-6}$
4.1 A	20	25	8	0	53	30, 8, 96	0.0010	$2 \cdot 10^{-5}$
	20	32	64	0	39	46, 20, 97	0.0007	$22 \cdot 10^{-5}$
4.2 B	20	41	17	17	18	116, 52, 125	0.0223	$3 \cdot 10^{-4}$
	20	18	36	0	8	111, 22,125	0.0006	$34 \cdot 10^{-4}$
	20	18	4	4	18	30, 13, 85	0.0013	$4 \cdot 10^{-4}$
4.2 B	23	39	32	0	37	86, 45, 125	0.0007	$1 \cdot 10^{-4}$
	23	63	5	5	25	124, 112, 125	0.0006	$1 \cdot 10^{-4}$
	23	63	5	0	44	16, 6, 29	0.0007	$1 \cdot 10^{-4}$
4.1 B	26	41	9	18	24	105, 64, 123	0.0024	$3 \cdot 10^{-6}$
	26	41	4	0	15	110, 15, 125	0.0016	$4 \cdot 10^{-5}$

Chapter 5

Direct Manipulation of Procedural Implicit Surfaces

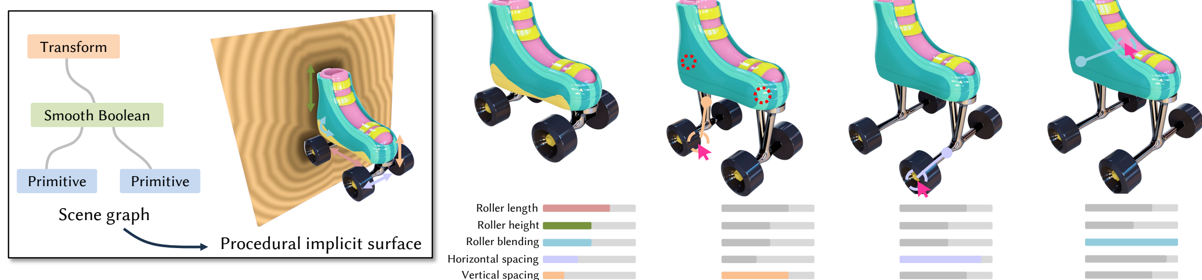


Figure 5.1. Our method enables the direct manipulation of procedural implicit surfaces through mouse strokes in the viewport. We estimate an update of the procedural parameters of the implicit surface that matches the user intent thanks to the auto-differentiation of an augmented version of the implicit function (Section 5.4.2). As opposed to the typical workflow of updating parameters through sliders, our method enables a more direct and intuitive editing process.

After analyzing problems and possible solutions to ease the editing of procedural vector patterns, we now shift to the 3D scenario by focusing on a popular procedural representation for shape modeling, namely procedural implicit surfaces. They provide a simple framework for complex geometric operations such as Booleans, blending and deformations. However, even in this case their editability remains a challenging task: as the definition of the shape is purely implicit, direct manipulation of the shape cannot be performed. Thus, parameters of the model are often exposed through abstract sliders, which have to be non-trivially created by the user and understood by others for each individual model to modify. Further, each of these sliders needs to be set one by one to achieve the desired appearance.

In this chapter, we propose a method to directly manipulate the implicit surface

in the viewport, this lifting this laborious process while preserving editability, We let the user naturally interact with the output shape, leveraging points on a co-parameterization we design specifically for implicit surfaces, to guide the parameter updates and reach the desired appearance faster. We exploit our automatic differentiation of the procedural implicit surface to propagate interactions made by the user in the viewport to the shape parameters themselves. We further design a solver that uses such information to guide an intuitive and smooth user workflow, demonstrating different editing processes across multiple implicit shapes and parameters that would be tedious by tuning sliders.

5.1 Introduction

When creating virtual worlds and prototypes, authoring 3D assets is crucial. In particular, procedural modeling has gained significant traction in the industry in the past decade, relying heavily on implicit surfaces [JQ14; Mag22; Wom22] – defined as the zero level set of a function. This representation is particularly interesting as it allows for hierarchical combinations of various functions representing primitives (e.g. spheres or boxes) and operators (e.g. Boolean operations, affine transformations or deformations) in a tree or a graph [WGG99; RMD11]. Each of these operators and primitives comes with its own set of procedural parameters, which can typically be adjusted through sliders for non-destructive authoring. However, editing the shape by adjusting individual sliders requires a comprehensive understanding of its parameterization, as multiple parts can be affected by a single procedural parameter. Conversely, editing one part of a shape may require modifications of several interdependent procedural parameters. To be able to circumvent this tedious process, we propose a direct manipulation approach to editing. This approach allows users to directly interact with the end surface in the viewport and propagating the changes to the relevant procedural parameters. While this kind of technique saw recent success for mesh-based parametric modeling [MB21; Cas+22; Gai+22], none of these approaches can be readily applied to implicit surfaces. Tracking of explicit points on the surface during manipulations cannot be achieved easily in implicit surfaces due to the lack of surface parameterization.

Our method allows the user to perform edits by simply selecting and dragging any desired parts on the implicit shape over the 3D viewport, while optionally expressing constraints on other patches to remain unchanged throughout the edit, thus increasing expressiveness. We automatically update the procedural parameters of the implicit surface to modify the shape to best match the user manipulation. However, to enable manipulation of a procedural shape we need to be able to characterize an element of surface in a way that is robust to changes in the procedural parameters.

As this is not trivially defined for implicit surfaces, we extend [MB21]'s framework of *co-parameterization*, enabling the definition of a point's location and its Jacobian with respect to the shape parameters. We adapt this framework to implicit surfaces and show how it can be used for direct manipulation purposes. We further refine the parameter update through a new solver that exploits the Jacobian computed in an automatic differentiation fashion. We compute the Jacobian for multiple groups of points, each of which represents a patch of the implicit surface. Our framework supports the direct manipulation of procedural parameters for classical implicit primitives combined with complex operators such as smooth Boolean, deformations and affine transformations (Figure 5.1).

Contributions We enable in-viewport editing of procedural implicit surfaces thanks to the following contributions:

- A mapping between point positions and unique identifiers for procedural implicit surfaces, allowing the proper tracking of a point during an edit.
- A solver that, given user mouse-strokes and multi-point constraints, interactively updates the values of dozens of procedural parameters to best match the user intent.
- A mean to evaluate the local influence of parameters on individual points of a shape, which could be applied in other optimization pipelines than direct manipulation.

We evaluate our method in terms of editing capacity (e.g. can we reach a desired shape) through a user study and a comparison to existing direct manipulation techniques for analytic implicit surfaces.

5.2 Overview

We aim at providing direct manipulation tools for procedural implicit surfaces where users interact with the shape itself directly in the viewport, rather than indirectly setting a value by moving an indicator on a track bar, namely a *slider*, as in traditional procedural modeling. Formally, a user edit consists in the selection of multiple points $\mathbf{p}_i$ over the surface, and their expected screen space movement ΔT_i. Typically, ΔT_i either matches the movement of the user mouse cursor, or is equal to zero to mean that the element should not move during the edit. The goal of our solver is to update the shape according to this edit while maintaining its global consistency. Our pipeline handles procedural implicit shapes described by scene graphs, as detailed in the next paragraph.

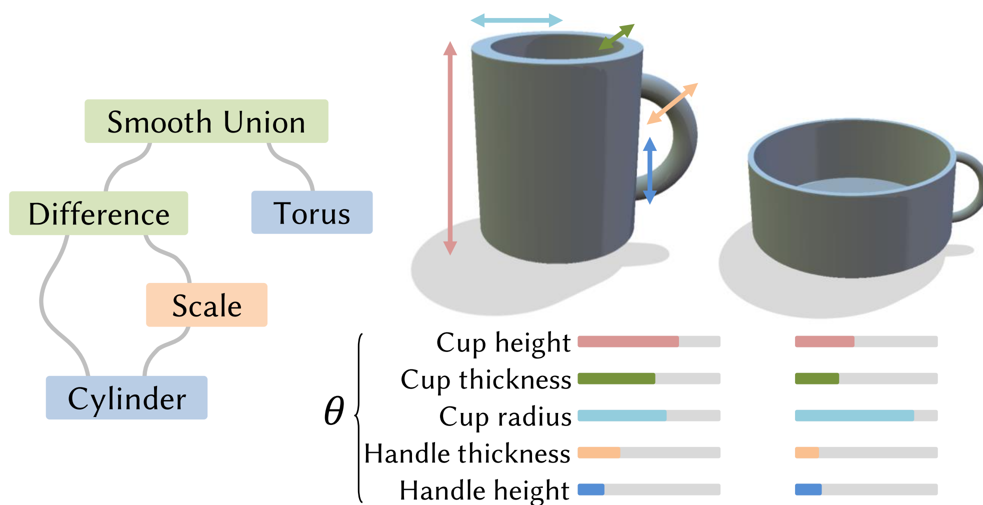


Figure 5.2. Our input is a procedural implicit shape represented by a scene graph (left), combining primitives, transformations, Boolean operations, etc. Procedurally-defined shapes allow users to create a large variety of instances $\Phi(\boldsymbol{\theta})$ by tweaking the procedural parameters $\boldsymbol{\theta}$ (right), but this is a non-trivial task as the user must understand the influence of each individual parameter over the model. In each figure of the current chapter, bidirectional arrows are used as a simplified semantical representation of procedural parameters.

Background. An implicit surface is defined as the zero level set of a function $f : R^3 \to R$. A point $\mathbf{p} \in R^3$ belongs to the implicit surface S if and only if it satisfies $f(\mathbf{p}) = 0$. Representing 3D shapes using implicit surfaces thus inherits from interesting properties of function objects, like their compact analytical representation or the possibility to compose them together.

A procedural implicit surface is a generalization of an implicit function f with a second argument from a procedural parameters space $\Theta \subset R^n$, where n is the number of procedural parameters.

These parameters, commonly used in general procedural modeling, are exposed by the designer of the initial shape to its end user.

The surface S for a particular value $\boldsymbol{\theta} \in \Theta$ is called an *instance* of the procedural shape Φ:

$$S = \Phi(\boldsymbol{\theta}) = \{\mathbf{p} \in R^3 \,|\, f(\mathbf{p}, \boldsymbol{\theta}) = 0\} \tag{5.1}$$

We support procedural implicit functions that are derived from a *scene graph*, like for instance BlobTrees [WGG99] or analytical Signed Distance Fields. A scene graph is a directed acyclic graph (or sometimes more simply a tree) whose nodes are either *primitives* such as spheres or boxes, or *operators* such as CSG operators or affine transformations (see Figure 5.2). Complex implicit shapes arise from the

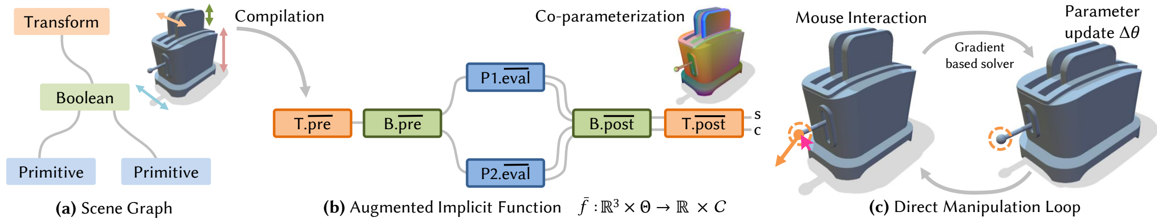


Figure 5.3. Starting from a scene graph representation of an implicit surface **(a)**, we augment it so that the resulting implicit function $\bar{f}$ computes both the scalar value s and a *co-parameter* $\boldsymbol{c}$ that identifies the evaluation point in space **(b)**. We do this by replacing the `eval`, `pre`, `post` functions of the different nodes. This allows estimating the derivative of a position with respect to the procedural parameters, which is then used to modify them to match the user stroke **(c)**.

combination of *primitives* via multiple boolean *operators* such as union, intersection of difference. In the implicit domain, a blended variation of regular boolean operations called *smooth boolean* is greatly exploited to create more organic shapes.

A formal derivation of the implicit function f from the scene graph is described is Section 5.3.

Problem setting. In our context, a *manipulation* of the procedural implicit surface means a change of the procedural parameters $\boldsymbol{\theta}$ through user interaction. This ensures that the deformed surface globally remains a valid instance of the procedural shape Φ. To comply with the user input, we minimize for each manipulated point the following loss:

$$\mathcal{L}_i := \frac{1}{2} \left\| \Delta \operatorname{Proj}\left(\mathbf{p}_i\right) - \Delta T_i \right\|_2^2 \tag{5.2}$$

where $\Delta \operatorname{Proj}(\mathbf{p}) = \operatorname{Proj}\left(\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}\right) - \operatorname{Proj}\left(\mathbf{p}^{\boldsymbol{\theta}}\right)$ is the effective movement of the point $\mathbf{p}$ after an update $\Delta\boldsymbol{\theta}$ of the procedural parameters, projected by Proj onto the screen. The first problem we address, in Section 5.4, is the definition of the new position $\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ of the dragged point. Indeed, while the initial position $\mathbf{p}^0$ is simply found by casting a ray onto the surface, tracking what is semantically the *same element of geometry* after the change of procedural parameters is challenging. We then derive in Section 5.5 the gradient of $\mathcal{L}_i$, and in particular the Jacobian matrix of each dragged position $\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ with respect to $\Delta\boldsymbol{\theta}$. Lastly, Section 5.6 details our gradient descent based solver. Our complete pipeline is summarized in Figure 5.3.

5.3 Scene Graph Model

In a typical scene graph used for implicit modeling, an oriented edge is used in two ways: from the root to the leaves, it carries a position $\mathbf{p}$ at which primitives

ALGORITHM 1: Derivation of the implicit function from a scene graph.
The expression of the implicit function f is derived from a scene graph by recursively compiling its root node into the expression of an evaluation function $R^3 \mapsto R$.

```
Input: A node n of the scene graph.
Output: The implicit function represented by the node n.
function CompileNode(n):
    if IsPrimitive(n)
        return n.eval;
    else
        children ← GetChildren(n);
        return n.post ∘ map(CompileNode, children) ∘ n.pre;
```

must be evaluated, then from the leaves back to the root, it carries the returned scalar value s. We formalize this by having each primitive provide an `eval` function, which maps a position $\mathbf{p} \in R^3$ to a scalar value $s \in R$, and each operator that has m input provide a function `pre` : $R^3 \rightarrow (R^3)^m$ that prepares the m positions fed to its inputs and a function `post` : $R^m \rightarrow R$ that reduces the m values returned by the inputs. For instance, the `eval` function of a sphere primitive of radius r is `eval` : $\mathbf{p} \mapsto \|\mathbf{p}\| - r$ and here are examples of operators:

Scaling by a factor x	**Union of 2 shapes**
`pre` : $\mathbf{p} \mapsto \mathbf{p}/x$	`pre` : $\mathbf{p} \mapsto (\mathbf{p}, \mathbf{p})$
`post` : $s \mapsto s \cdot x$	`post` : $(s_1, s_2) \mapsto \min(s_1, s_2)$

The final expression of f is obtained by recursively chaining the `pre`, `eval` and `post` expressions as detailed in Algorithm 1. The free variables of the expression – e.g. the scale factor x or the radius r in the examples above – constitute the vector $\boldsymbol{\theta}$ of procedural parameters. In practice there is usually a remapping between the parameters that are publicly exposed to the end user and the low-level parameters of the graph nodes, but we consider without loss of generality that this is part of the `eval`, `pre` and `post` functions.

We assume the resulting function f to be continuous and differentiable around its zero level-set. In order to render shapes using sphere tracing [Har96], we also assume that f is Lipschitz, ie. that there is a bound λ on the magnitude of ∇f, ensuring that $|f(\mathbf{p}, \boldsymbol{\theta})|/\lambda$ is always lower than the distance from $\mathbf{p}$ to the surface. This in turn allows to compute points on the surface using sphere tracing for direct manipulation purposes.

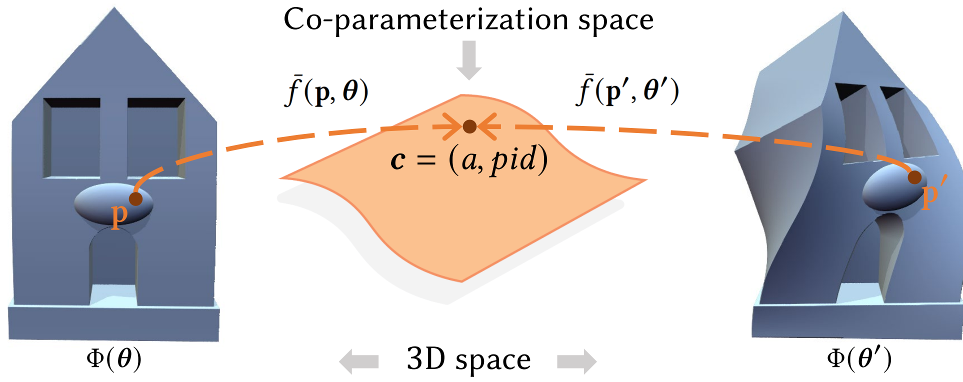


Figure 5.4. Co-parameterization enables the identification of the same point location before and after an edit. The two points $\mathbf{p}$ and $\mathbf{p}'$, selected in different shape instances $\Phi(\boldsymbol{\theta})$ and $\Phi(\boldsymbol{\theta}')$, have two different locations in R^3, but are mapped to the same co-parameter $\boldsymbol{c} \in \mathcal{C}$ because they represent the same semantic element of the shape. This enables to define the influence $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ of the procedural parameters at a given point $\mathbf{p}$.

5.4 Coparameterization

To enable direct manipulation, we need to robustly identify the same point location throughout the edit as it allows us to estimate the local influence of the procedural parameters. We formalize $\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ as the position of a point $\mathbf{p}$ after an update $\theta + \Delta\theta$ of the procedural parameters (Section 5.4.1), and describe how to use the structure of the scene graph to track the identity of manipulated points (Section 5.4.2).

5.4.1 Definition

The sole expression of the implicit function f cannot provide the position $\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ of a dragged point for an arbitrary change $\Delta\boldsymbol{\theta}$ of the procedural parameters, because in its compiled form it lacks the semantic awareness of the original scene graph. We propose to define an *augmented* implicit function $\bar{f} : (\mathbf{p}, \boldsymbol{\theta}) \mapsto (s, \boldsymbol{c})$ that not only return the scalar value $s \in R$ but also a feature vector $\boldsymbol{c} \in \mathcal{C}$ meant to uniquely identify what role the position $\mathbf{p}$ plays in the instance $\Phi(\boldsymbol{\theta})$. This so-called co-parameter $\boldsymbol{c}$ formally characterizes the notion of *same point*: $\bar{f}(\mathbf{p}, \boldsymbol{\theta}) = \bar{f}(\mathbf{p}', \boldsymbol{\theta}')$ if and only if $\mathbf{p}$ and $\mathbf{p}'$ are two positions of the same element of geometry under different procedural parameters $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ (Figure 5.4). Hence $\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ is defined as the only point such that $\bar{f}(\mathbf{p}^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}, \boldsymbol{\theta} + \Delta\boldsymbol{\theta}) = \bar{f}(\mathbf{p}^{\boldsymbol{\theta}}, \boldsymbol{\theta})$. Note that s is always 0 for surface points, so $\boldsymbol{c}$ is what enforces point identity.

Our co-parameter space $\mathcal{C} = R^3 \times N$ is detailed in Section 5.4.2, as well as how we build $\bar{f}$ in practice. Note that contrary to [MB21], we define the co-parameterization on the whole space rather than only on the surface, due to the implicit nature of

our shapes.

5.4.2 Augmented Implicit Function

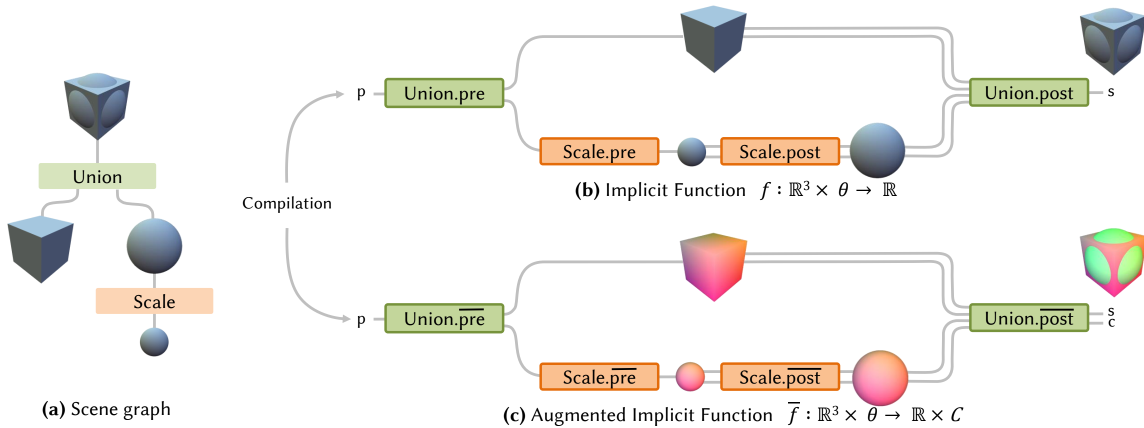
The scene graph from which our implicit function is derived carries semantic information that also suggests a notion of what *same point* means. This section describes how we modify the construction of the implicit function to encode this extra information as a co-parameterization that we can then use in our solver. Our requirements for the definition of the co-parameterization are **(a)** to uniquely and robustly identify elements of geometry **(b)** to be locally differentiable (Section 5.5) and **(c)** to be automatically constructed for all of our 3D implicit shapes. To ensure this last point, we build the co-parameterization together with the implicit function f, by defining an *augmented* version of the `eval`, `pre` and `post` functions we described in Section 5.3:

$$\begin{aligned} \overline{\texttt{eval}} &: R^3 \longrightarrow R \times \mathcal{C} \\ \overline{\texttt{pre}} &: R^3 \longrightarrow (R^3)^m \qquad \text{(Unchanged)} \\ \overline{\texttt{post}} &: (R \times \mathcal{C})^m \longrightarrow R \times \mathcal{C} \end{aligned}$$

We provide these augmented functions only once for each type of primitive and operator, and the very same Algorithm 1 hence defines an *augmented* implicit function $\bar{f} : (\mathbf{p}, \boldsymbol{\theta}) \mapsto (s, \boldsymbol{c})$ that returns both the scalar value $s \in R$ and the co-parameter $\boldsymbol{c} \in \mathcal{C}$ at the evaluated position. The co-parameter $\boldsymbol{c} = (\boldsymbol{a}, pid)$ is made of a differentiable part $\boldsymbol{a} \in R^3$ and a *path index* part $pid \in N$ that uniquely identifies the path followed in the scene graph during the evaluation of a point. The following paragraphs describe rules of thumb for defining the augmented version of $\overline{\texttt{eval}}$ and $\overline{\texttt{post}}$.

Primitive nodes They represent atomic shapes, and are often derived from a fixed *canonical shape* that is only rigidly transformed by the procedural parameters (e.g., ellipsoid, prism, cylinder, etc.). In this case, we use the position of a point $\mathbf{p}$ in this canonical space as its unique identifier $\boldsymbol{a}$, and the path index pid is always 0 for a primitive. This idea can be generalized to other shapes, sometimes at the cost of a local discontinuity (e.g., we parameterize the torus as a bent cylinder). While these primitives represent basic shapes, our framework enables highly complex shape design through operator nodes, allowing for vast combination of primitives, as it is typically done in implicit modeling [WGG99].

Operators nodes Basic operators simply forward the co-parameter received from their input, but in general operators may introduce ambiguity in the co-parameterization in two ways: by combining multiple inputs, and by duplicating



(a) Scene graph

(c) Augmented Implicit Function $\overline{f} : \mathbb{R}^3 \times \theta \to \mathbb{R} \times C$

Figure 5.5. The implicit function f and the augmented implicit function $\bar{f}$ can be derived from the scene graph **(a)** via a compilation process. While the former **(b)** only provides the distance output for each position $\mathbf{p}$ it is queried on, the latter **(c)** augments such information by also returning its co-parameter $\boldsymbol{c} = (\boldsymbol{a}, pid)$.

geometry. Both potentially lead to multiple points sharing the same co-parameter, which we avoid by adding the integer part *pid* of the co-parameterization. A 2-input operator offsets the path index *pid* that it receives from its second input by 1+ the maximum *pid* it may receive from its first input. Its output *pid* is forwarded from either of its input depending on its behavior. Operators with more than 2 inputs are decomposed into sub steps, and duplication operators are treated as chains of 2-input unions. While this allows us to manipulate many different implicit operators (including Boolean and smooth Boolean operators, affine transformations, and warping), we discuss in Section 5.7 more challenging operators that we do not support. Note that this integer part *pid* is ignored when differentiating the co-parameter, but is used in the solver to control that the dragged point is properly tracked (see Section 5.6).

Compilation Example The scene from Figure 5.5 is made up of 2 primitives, namely the *Sphere* and the *Box*; the former is transformed using the unary operator *Scale* and finally combined with the latter via the binary *Union* operator to create the final shape. By recursively following Algorithm 1, we start from the *Union* node and apply the `CompileNode` function to its children, which are the *Box* primitive and modifier *Scale*, that can be further unrolled until the base case *Sphere* is reached. By recursively applying Algorithm 1, the implicit function $f : R^3 \times \Theta \to R$ derived from the scene graph.

While the function f only evaluates the distance of a point in space from the implicit shape, we need additional information to allow the direct manipulation of the implicit shape itself. We apply the same Algorithm 1 but replace each node's

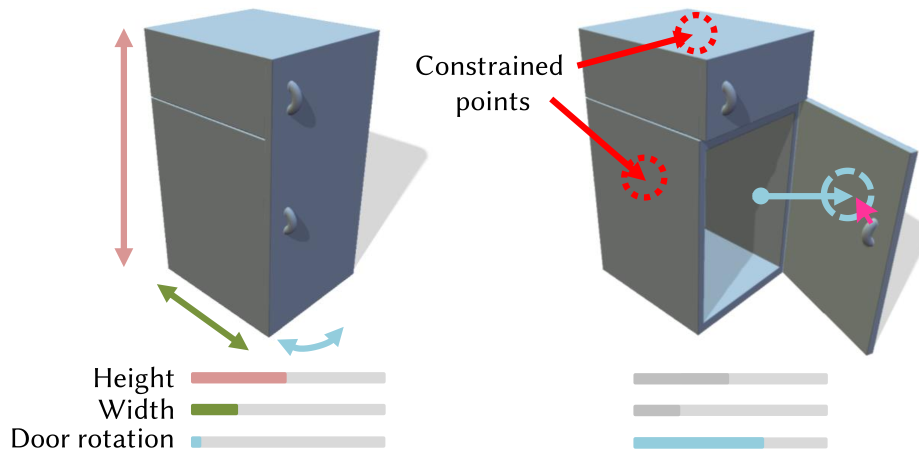


Figure 5.6. To reduce ambiguities during an edit, our framework enables multi-point constraints over parts of the shape. This guides the optimization towards an procedural parameter update that does not affect the constrained areas, thus increasing the framework expressiveness.

function with their augmented version $\overline{\texttt{pre}}$, $\overline{\texttt{eval}}$ and $\overline{\texttt{post}}$, thus computing and propagating both the distance and the co-parameter for each evaluated position. In this case, Algorithm 1 derives the augmented implicit function $\bar{f} : R^3 \times \Theta \rightarrow (R \times \mathcal{C})$.

Exploiting the information propagated by the augmented output makes it possible to track the same point during an edit, and allows estimating the influence of the procedural parameters over them.

5.5 Evaluation and normalization

The computation of Jacobians with respect to procedural parameters is a core step in the optimization process. Here we define how to compute and refine them accordingly to the user selection (Figure 5.7).

5.5.1 Jacobian Evaluation

Minimizing the direct manipulation loss $\mathcal{L}_i$ from Equation 5.2 requires to evaluate at $\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ the Jacobian $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ of the position $\mathbf{p}$ of a point with respect to procedural parameters $\boldsymbol{\theta}$, at a fixed point identity $\boldsymbol{c}$, as described in Sec. 5.6. Fortunately, this $3 \times n$ matrix can be derived from the Jacobian of $\bar{f} : (\mathbf{p}, \boldsymbol{\theta}) \mapsto (s, \boldsymbol{c})$ by applying the implicit function theorem:

$$\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}} = -\left(\frac{\partial \bar{f}}{\partial \mathbf{p}}\right)^{+} \cdot \frac{\partial \bar{f}}{\partial \boldsymbol{\theta}} \tag{5.3}$$

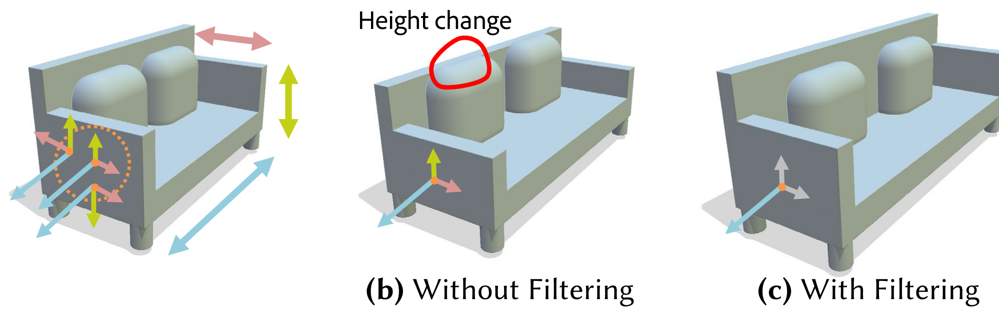


Figure 5.7. We automatically filter out some columns of the Jacobian matrix to deactivate procedural parameters that are deemed less relevant to the edit. Here, a selection on the side of the sofa suggests an edit involving the width rather than the height or depth. Without filtering **(b)**, all dimensions are updated, resulting in undesirable changes in height and depth, while our filtering discards the height and depth dimensions, focusing on width **(c)**.

where $(\cdot)^+$ denotes the pseudo-inverse of a matrix. We use automatic differentiation to evaluate the Jacobian of all outputs of $\bar{f}$ with respect to both the procedural parameters $\boldsymbol{\theta}$ and the position $\mathbf{p}$.

However, simply evaluating the point-wise Jacobian of our procedural shape has two major drawbacks that we need to address: first, the different columns of the Jacobian – which relate to different procedural parameters – are *not homogeneous* in terms of scales and units. For this, we perform a normalization step that we detail in Section 5.5.2. Second, the differential information is only valid for a single point, potentially resulting in a noisy interaction as the user typically drags a patch of surface. There is a need for a filtering step where we zero the values that relates to procedural parameters that we do not aim at modifying (Section 5.5.3).

5.5.2 Jacobian Normalization

Switching e.g., a length parameter from meters to millimeters divides by a factor 1000 the corresponding columns of the Jacobian $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ and thus leads to gradient descent updates 1000 times slower only for this parameter. To prevent this, we estimate a normalization factor for each procedural parameter of the model. We randomly sample 50 rays from the six views aligned with the canonical axes, and at each intersection between a ray and the implicit shape we evaluate the Jacobian $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$. The normalization factor m_i of the i-th procedural parameter is then defined as the maximum magnitude of the i-th column of the Jacobian over all samples. We similarly update the normalization factors m_i after each user edit and each change of viewpoint. Normalization factors are exploited in the Jacobian filtering process (Section 5.5.3), enabling a direct and robust comparison between parameters, and

are also used to scale the gradient of the loss function $\nabla\mathcal{L}$ during the optimization (Section 5.6).

5.5.3 Jacobian Reduction and Filtering

To increase the robustness of the Jacobian, we estimate it for a neighboring patch of surface rather than at a single point. We evaluate $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ at 16 sample points within a screen-space disk centered on the user's mouse cursor and reduce them to their average $3 \times n$ matrix.

We then filter this patch-wise Jacobian matrix, noted $J(\mathbf{p}, \boldsymbol{\theta})$, by cancelling out procedural parameters that do not influence enough the position of the patch. The process of filtering the procedural parameters does not require any input from the user, and is automatically performed by extracting information from the Jacobian matrices themselves. We filter the i-th column of the Jacobian by estimating the influence of the i-th procedural parameter on the geometry. To be preserved, each Jacobian column $\boldsymbol{j}_i$ has to respect at least two of the three following conditions. **(a)** Its normalized magnitude $\frac{\|\boldsymbol{j}_i\|}{m_i}$ must be higher than an empirical threshold $\lambda_{mag} = 0.35$, aiming to keep dimensions with large impact on the shape geometry. **(b)** The standard deviation of $\boldsymbol{j}_i$ across all selected points must be lower than an empirical threshold $\lambda_{std} = 0.2$, aiming to keep dimensions that behaves similarly across the patch. **(c)** The angular distance $d_v \cdot \frac{\boldsymbol{j}_i}{\|\boldsymbol{j}_i\|}$, where d_v denotes the view direction, must be lower than an empirical threshold $\lambda_{view} = 0.4$ to foster dimensions whose impact is orthogonal to the view direction.

5.6 Solving

Thanks to the evaluation of the filtered and reduced Jacobian $J(\mathbf{p}, \boldsymbol{\theta})$ of $\mathbf{p}$ with respect to $\boldsymbol{\theta}$, we can integrate the manipulated shape in generic continuous optimization frameworks. At each frame of the interaction, we use a few steps of gradient descent to minimize the following multi-point manipulation loss:

$$\mathcal{L} = \sum_i \mathcal{L}_i + \lambda \mathcal{L}_{reg} \tag{5.4}$$

where $\mathcal{L}_{reg} = \|\Delta\boldsymbol{\theta}\|_2$ is a regularization term that prevents sudden changes in procedural parameters and $\lambda = 0.2$. The gradient of $\mathcal{L}_i$ with respect to $\Delta\boldsymbol{\theta}$ is:

$$\nabla\mathcal{L}_i = J_{\text{Proj}} \cdot J(\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}, \boldsymbol{\theta} + \Delta\boldsymbol{\theta}) \tag{5.5}$$

When updating $\boldsymbol{\theta}$ at each step of the gradient descent, we divide coefficient-wise the gradient $\nabla\mathcal{L}$ by the vector m of normalization factors (Section 5.5.2). Since the surface is only implicit, the updated positions $\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ are estimated as part of the

optimization (Equation 5.8), and the confidence of this estimation is measured by the difference $\|\Delta c\| = \|\Delta a\|_2 + \delta_{\Delta pid,0}$ between the initial co-parameter of $\mathbf{p}_i^{\boldsymbol{\theta}}$ and the one evaluated at $\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$. When this error exceeds a threshold $e_c = 0.7$, we divide the gradient by $\|\Delta c\|$ to slow down the drift. The scaled gradient is then multiplied by the global learning rate η. We then update the estimate $\mathbf{p}_i$ of the 3D position $\mathbf{p}_i^{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ of the dragged point:

$$\Delta\boldsymbol{\theta} := -\eta\nabla\mathcal{L}/m \tag{5.6}$$

$$\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta} \tag{5.7}$$

$$\mathbf{p}_i \longleftarrow \mathbf{p}_i + J(\mathbf{p}_i, \boldsymbol{\theta}) \cdot \Delta\boldsymbol{\theta} \qquad \forall i \tag{5.8}$$

5.7 Results

We implemented our method as well as our implicit primitives and operators in C++/GLSL. We use libfive trees [Kee19] as target to the compilation of our scene graph representation (Algorithm 1), which provides us with symbolic expression optimization and numeric automatic differentiation. All models shown throughout this chapter (Figure 5.1, 5.6, 5.7, 5.10, 5.11) were rendered using sphere tracing [Har96] in a standalone application. Experiments were performed on a desktop computer equipped with AMD® Ryzen 5 clocked at 3.6 GHz with 32 GB of RAM, and an NVIDIA GTX 1050 graphics card. Statistics for the models shown throughout this chapter and performances for the different steps of the pipeline are reported in Table 5.1.

Performance Our method runs at interactive framerates (including manipulation and rendering), enabling direct manipulation by the user without the need to wait for any sort of loading. The most computationally intensive part is the solving (Table 5.1), where the maximum number of gradient descent iterations is set to 50, which we found to be a good trade off between quality and performance To achieve interactivity, we optimize the expression of $\bar{f}$ using libfive expression optimization feature at initialization. As a further improvement in speed, the Jacobians used during solving are updated every 4 frames.

Control. We illustrate our direct manipulation tool on a set of 11 procedural implicit surfaces with varying complexity and topology. Scene graph complexity spans from a minimum of 19 nodes (Figure 5.6) to 265 nodes for the roller model in Figure 5.1, with a number of procedural parameters ranging from 4 to 45. Figure 5.11 shows editing sessions with three successive edits to these models. A typical workflow in our framework involves fixing some parts of the implicit surface while dragging

Table 5.1. Performance for the different scenes, with the amount *#nodes* of nodes in the scene graph and $\#\boldsymbol{\theta}$ of procedural parameters. We report the execution time for the different steps of an edit, namely the co-parameter sampling time t_c, the Jacobian evaluation time t_j and the average solving time t_s for the 50 optimization iterations. All timings are in ms.

Scene	Fig.	$\#\boldsymbol{\theta}$	*#nodes*	t_c	t_j	t_s
Roller	5.1	8	265	1.524	2.365	9.903
Cup	5.2	5	39	2.523	0.151	0.548
Fridge	5.6	3	19	2.325	0.214	0.381
Sofa	5.7	13	77	2.319	0.678	1.652
Webcam	5.10	5	206	1.628	0.935	3.503
Cheese	5.11a	27	158	1.897	1.932	3.145
Robot Arm	5.11b	6	243	2.061	0.732	2.831
Pipes	5.11c	6	249	1.559	1.510	4.152
Toaster	5.11d	4	256	1.904	1.126	4.811
House	5.11e	20	244	2.137	0.999	2.313
Rabbit	5.11f	45	188	1.484	3.980	5.928

some other parts, as highlighted in Figure 5.1, 5.6, and 5.11. On top of our Jacobian filtering (Section 5.5.3), this helps the solver disambiguate the procedural parameter update and allows for more intuitive edits, even when warping and affine transformations are involved. For instance, the House model combines a bend and a twist, that can both be manipulated separately when the right fixed constraints are provided. Another example is the Robot arm, which involves chained rotations that can be controlled independently by fixing points on the different joints.

Topology changes Our framework is also resilient to changes in topology induced by CSG operators (union, intersection, and difference) when some procedural parameters are altered. This is an important feature since the ease with which one can create varying topology is one of the key strengths of implicit modeling, and it emerges from the presence of a unique path index (*pid*) in the co-parameter that identifies the dragged points. We highlight the resiliency to topology changes in Figure 5.8 as well as in the edits performed on the pipe and cheese models (Figure 5.11)

Comparison with other techniques Our method is focused on the direct manipulation of analytic implicit surfaces defined as procedural scene graphs. We support the classical operators of implicit modeling, such as CSG operators that changes

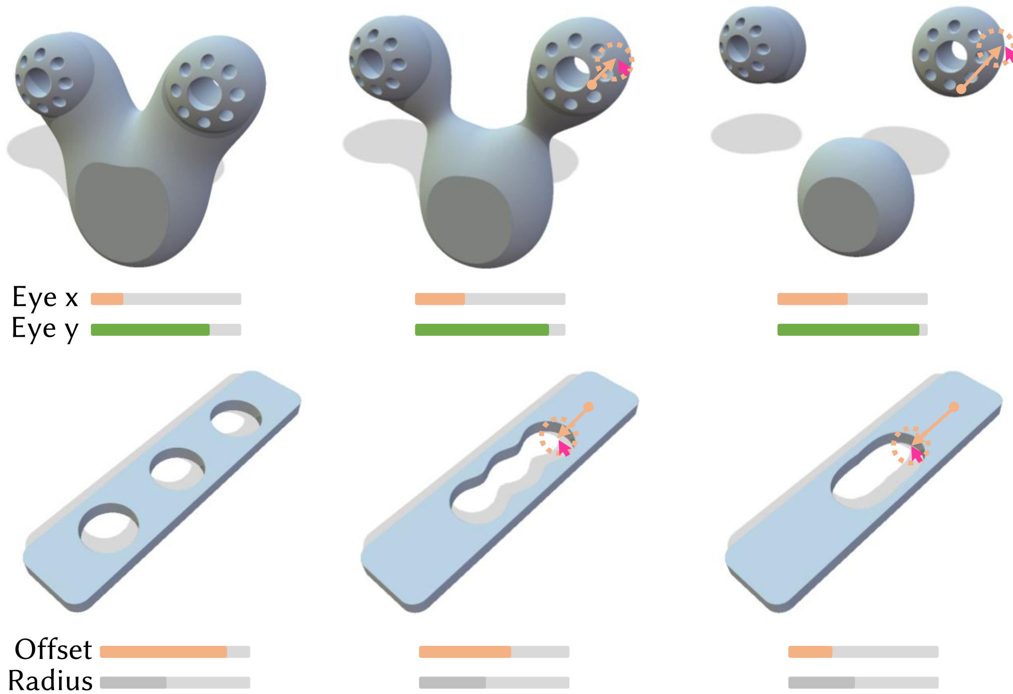


Figure 5.8. Examples featuring changes of topology. Our method naturally inherits from the ability of procedural implicit surfaces to represent objects of varying topology, be it through additive of subtractive, smooth or hard boolean operations.

the topology of the model, morphological operators (e.g., dilatation) and smooth Booleans (see Figure 5.9), bringing the power of direct manipulation techniques performed on meshes [MB21; Gai+22; Cas+22] to procedural implicit surfaces.

Close to our method is the direct manipulation solution exposed in libfive [Kee19], which allows manipulation of analytic implicit surfaces. Their solver tries to find a procedural parameter update such that *some* part of the surface passes by the new mouse position, back-projected into the 3D space along the normal of the base position on the model. However, it has no way to identify *which* part of the shape the user intends to modify. A failure case of their method is shown on Figure 5.10, where the user tries to modify the lens position of the camera. Although libfive's heuristic behaves well for procedural parameters that move elements of surface along their normal, it struggles with tangential movements, due to its lack of awareness of a point's identity. In contrast, our solution appropriately evaluates the local influence of a procedural parameter in all directions equally.

User Study We evaluated the effectiveness of our method via a user study involving 20 subjects, whose backgrounds in 3D editing ranged from novice to proficiency. After a hands-on session where the user could try both slider-based and direct

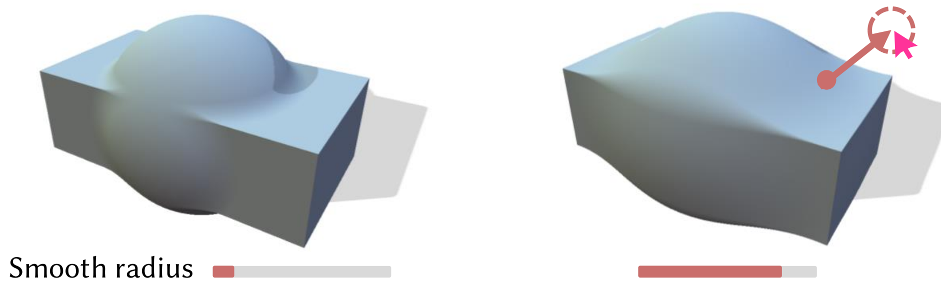


Figure 5.9. Smooth Booleans are key operators of implicit shape modeling, as they blend shapes together in a more organic way than hard Booleans (left). Our framework naturally supports updating the smoothness parameter, by dragging a point from the smooth junction between the two shapes (right).

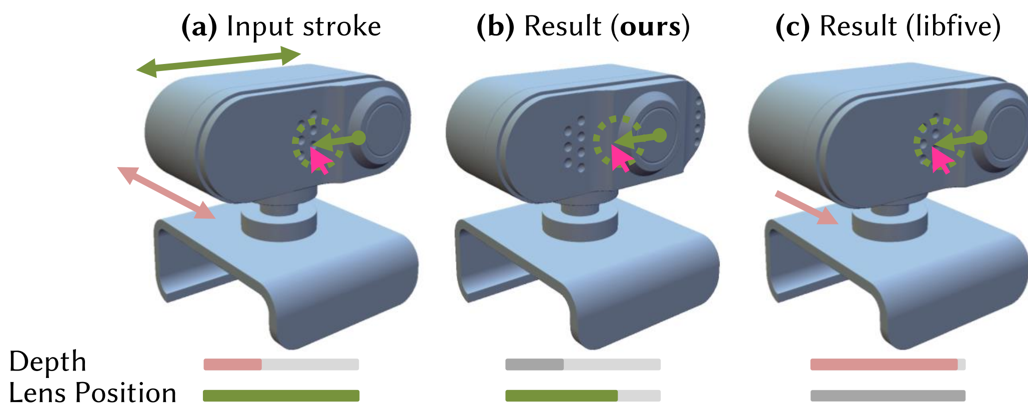


Figure 5.10. When the user intends to drag points in a direction significantly different than the local surface normal **(a)**, our direct manipulation approach keeps track of the dragged point **(b)** while libfive's solver only constrains that the overall surface passes by the new mouse position **(c)**. In this very example, libfive is never able to affect the lens position parameter.

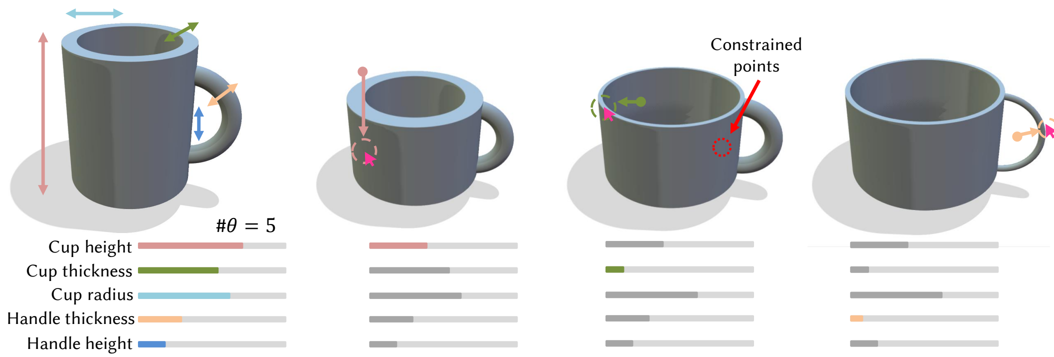
manipulation interactions, a more comprehensive editing session of five tasks is performed. Each task requires the user to reach a target shape configuration shown in a provided reference image, using slider-only interaction in the first task and direct manipulation for the remaining ones. It emerged that 95% (all but 1 subject) found the direct manipulation tool to be reactive to their inputs, and the 55% assessed that they rarely or never find themselves preferring to use sliders over direct manipulation. Some of them experienced a little frustration and provided feedback like *"It is sometimes difficult to isolate the exact parameter you want to tune"*. This opens up possibilities for improvements, such as presenting hints on areas of the model regarding the parameters that have the greatest impact on them. Overall, almost the totality of users leaned towards the use of a direct manipulation tool if provided by 3D editing software.

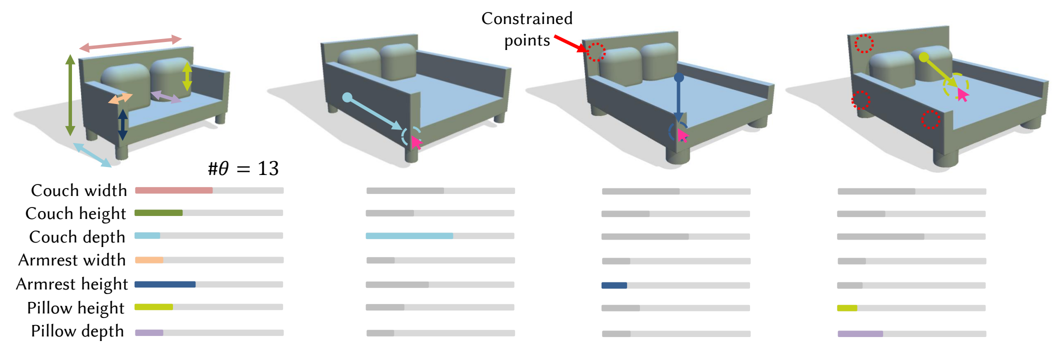
Limitations and Future Work Our method allows the direct manipulation of analytic implicit surfaces, but does not come without limitations. First, an explicit co-parameter must be derived for primitives and operators that we aim at editing, which may not be trivial. While our framework supports many different primitives and operators, some remain to be integrated. For instance, we do not support domain repetition operators, which requires to discriminate points belonging to different instances by producing a different *pid* for each one. Our definition of co-parameter does not ensure consistent interpolation, thus morphing operators can lead to unintuitive manipulation. Another limitation of our co-parameterization is that it must remain injective, which is not fulfilled in some edge cases, e.g., when the size of a box becomes 0. In this case, multiple points end up at the same position.

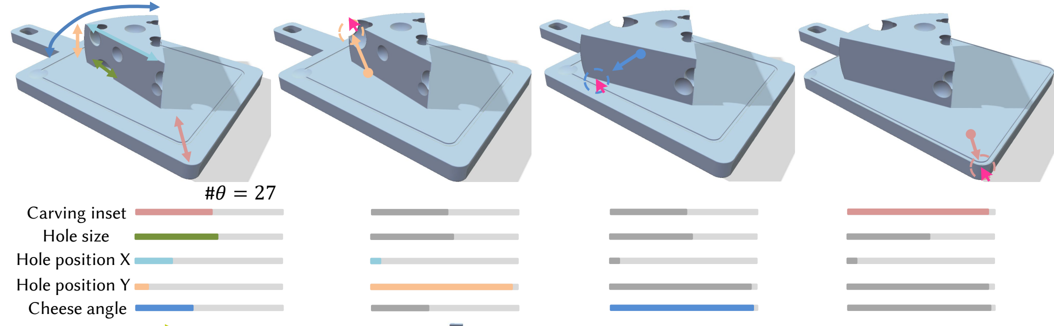
We found that our Jacobian filtering (Section 5.5.3) does not perform as well for models with too many procedural parameters influencing the same patch. This is partially solved by using multiple fixed constraints for editing, but future work may investigate more advanced filtering and reduction strategies. Finally, our current gradient descent optimization (Section 5.6) could benefit from more advanced techniques, such as ADAM optimization, or even Natural Gradient Descent as our total number of parameters remains small.

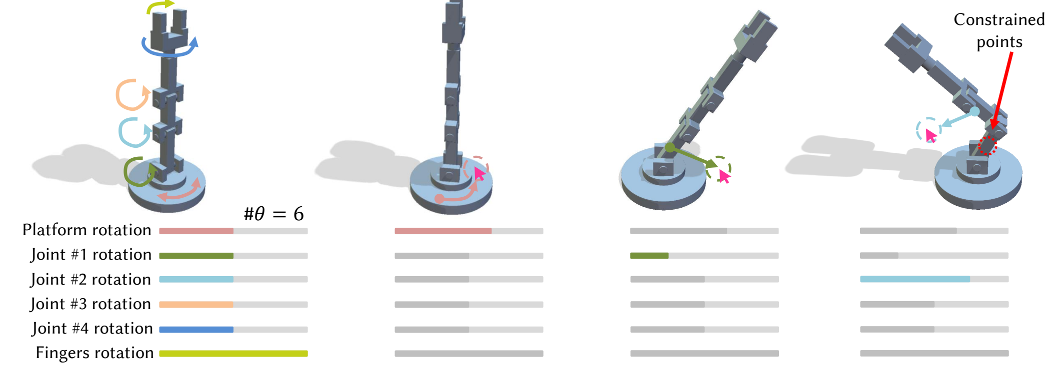
5.8 Conclusion

In this work, we proposed a direct manipulation approach for procedural implicit surfaces. We automatically augment the implicit function to output a *co-parameter*, allowing to robustly track the same point location throughout an edit. We leverage this to enable users to directly drag parts of the shape in the viewport, as opposed to tediously edit sliders, and more generally open the opportunity to evaluate the local influence of each procedural parameter on individual points of a shape. Our framework supports the direct manipulation of implicits made of a large set of primitives and complex operators, including warping and affine transformations.


Constrained points
$\#\theta = 5$
Cup height
Cup thickness
Cup radius
Handle thickness
Handle height


Constrained points
$\#\theta = 13$
Couch width
Couch height
Couch depth
Armrest width
Armrest height
Pillow height
Pillow depth


$\#\theta = 27$
Carving inset
Hole size
Hole position X
Hole position Y
Cheese angle


Constrained points
$\#\theta = 6$
Platform rotation
Joint #1 rotation
Joint #2 rotation
Joint #3 rotation
Joint #4 rotation
Fingers rotation

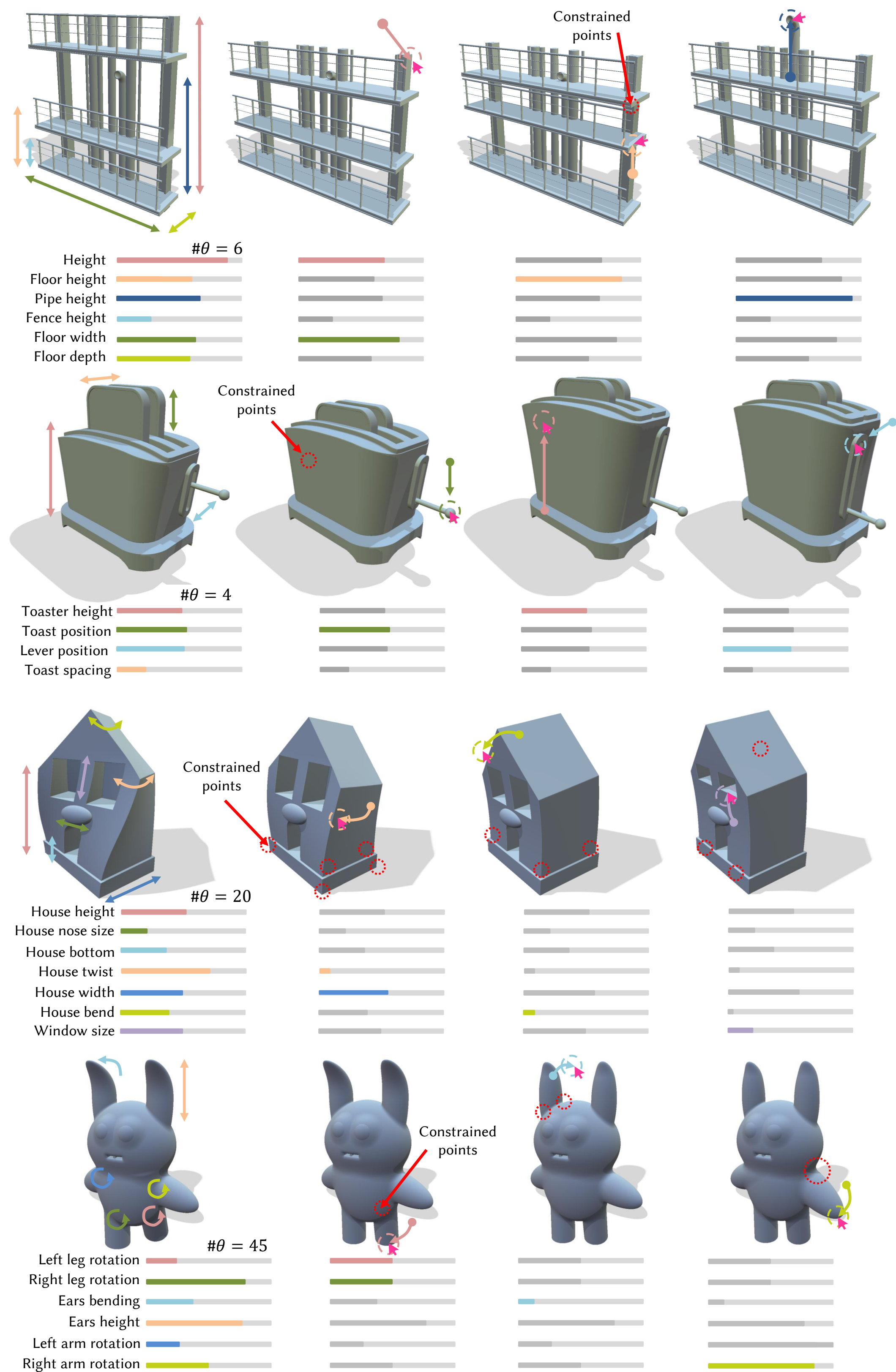


Figure 5.11. Editing sessions performed using our framework. The first image represents the original procedural implicit shape with a simplified semantical representation of its procedural parameters. The remaining images show three consecutive edits that are performed on the implicit shape, including both constrained and unconstrained manipulations. For each edit, we report the selected points and the mouse trajectory, highlighting the procedural parameter update in the underlying sliders.

Part II

Controlling Non-Procedural Assets

Chapter 6

Structured Pattern Expansion with Diffusion Models

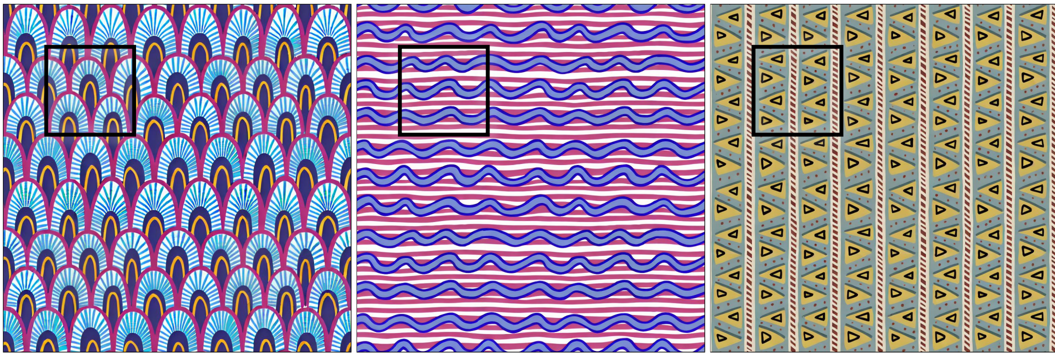


Figure 6.1. We propose a diffusion-based model for structured pattern expansion. Our approach enables the generation of large-scale, high-quality tileable patterns by extending a user-drawn input, shown within the black boxes, to an arbitrarily-sized canvas. Our method extends the input pattern while faithfully following the user input and producing coherently-structured yet non-repetitive images.

After digging in detail regarding procedural asset editing, we now focus on enhancing controllability in non-procedural asset ones. Leaving aside standard texture synthesis techniques, we focus on recent diffusion model applications, which facilitate the synthesis of materials, textures, and even 3D shapes. By providing a conditioning, usually in the form of a text or an image, users can guide content generation reducing the time needed for creating digital assets. In this chapter, we focus on the synthesis of structured, stationary, patterns, where diffusion models are less reliable and, more importantly, less controllable.

Our approach, namely *pAff*, leverages the generative capabilities of a stable diffusion model and adapts it to the pattern domain. We give users direct control on

the synthesis by expanding a partially hand-drawn pattern into a larger pattern while preserving the structure and details present in the input. We improve the quality of the synthesized pattern by incorporating a Low-Rank Adaptation (LoRA), to tune the model on structured patterns, a noise-rolling technique, to ensure tileability, and a patch-based approach, for the generation of large-scaled assets.

We demonstrate the effectiveness of our method through a comprehensive set of experiments, outperforming existing models in generating diverse, consistent patterns, that are directly controlled by user input.

6.1 Introduction

Hand-drawn structured patterns play a central role in computer graphics, finding applications across a variety of fields in design and digital art. The creation of these patterns remains a complex task, which requires both expertise and significant time investment. AI-assisted content creation, as shown, promize to simplify asset creation. For example, learning-based image-synthesis methods have shown impressive generation capabilities for natural images [Rom+22; BDS18; Kar+17; Kar+20; Pod+23]. The application of these methods to pattern-like synthesis focuses on mostly unstructured realistic materials [He+23; Vec+24; Vec+23], leaving the creation of structured patterns an under-explored task.

Given this scenario, we focus on structured patterns with a hand-drawn appearance, formed by the repetitions of hand-drawn shapes painted with solid colors and crisp edges. More formally, our structured patterns are stationary repetitions of recognizable shapes, with per-shape variations, and drawn with piece-wise constant colors. Examples of these patterns are visible throughout this chapter, with Fig. 6.2 also showing images of textures outside of our scope. We focus on these patterns for their importance both in design applications and because no current learning-based method addresses their synthesis directly.

In *pAff*, we focus on Latent Diffusion Models [Rom+22] as the base synthesis method. While these methods have made significant strides in natural image synthesis, they are not suited for generating structured patterns. Their first limitation is that the quality of the synthesized patterns is poor. This is because these models are trained to generate photo-realistic images characterized by unstructured, more chaotic, textures with high-frequency, stochastic, color variations. When applied to our domain, the output of these models does not maintain the patterns' structure, nor their sharpness, and coherent visual style.

Furthermore, in many design applications, users want to specify patterns precisely. Most diffusion models focus on text-to-image synthesis that does not allow for a precise specification of the desired pattern, since it is hard to describe in words the

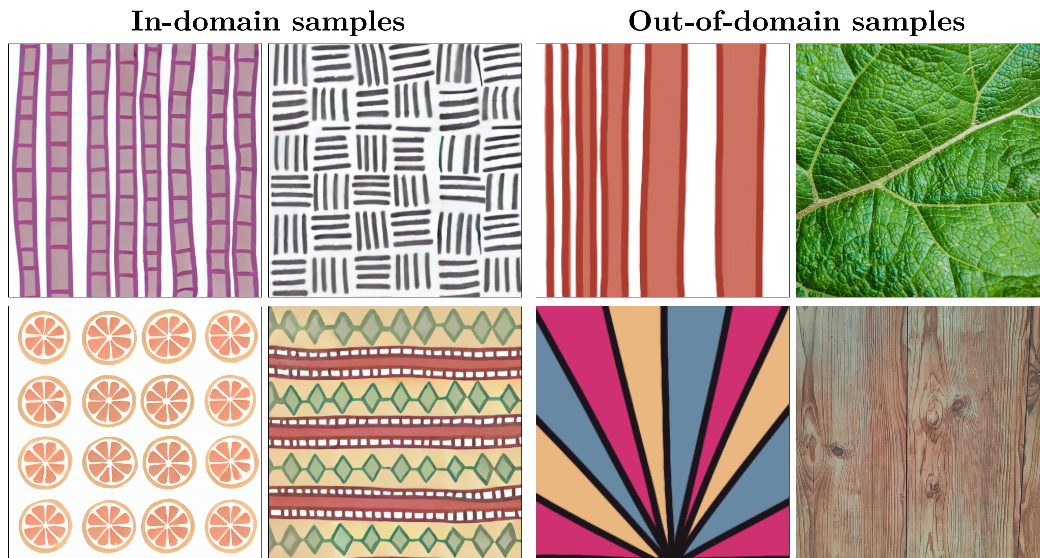


Figure 6.2. Pattern Category. We focus on structured, stationary, patterns in a hand-drawn style, characterized by repeated recognizable shaped drawn in flat colors (left). Unstructured or aperiodic patterns, as well as photorealistic textures, fall outside the scope of out method (right).

structure and appearance of the pattern. Even when using images as guides, we found current methods to perform unreliably in our domain.

To close this gap in literature and provide artists with simpler, but controllable, means for content creation, we propose a diffusion-based model specifically designed for the synthesis and expansion of structured, stationary patterns. In particular, we leverage the extensive knowledge already available in large-scale models, such as Stable Diffusion [Rom+22; Pod+23], and adapt it to the patterns domain by training a "small" LoRA [Hu+21a]. This approach not only allows us to reduce the computational and data requirements of training a diffusion model from scratch but helps to retain the expressivity of models trained on large-scale datasets like LAION [Sch+21], while adapting it to our specific domain. To that end, we collect a dataset of procedurally designed patterns that we use to train our LoRA.

We base our architecture on an inpainting pipeline, which supports the expansion of a partial, hand-drawn input sketch into a larger pattern while preserving its structural integrity and details. During inference, we leverage noise rolling and patch-based synthesis to produce large-scale, tileable patterns, at high quality in a reliable way. These design choices allow us to generate large-scale, tileable patterns that accurately follow the input sketch while adding a limited degree of variation avoiding visible repetitions.

We evaluate our approach by demonstrating its effectiveness in expanding patterns

from a large set of input sketches, and by qualitatively showing the improvement over previous state-of-the-art approaches in texture synthesis. We analyze our architecture to demonstrate the benefits of its design choices, by conducting a comprehensive set of experiments and ablation studies. The results show that our approach generates a wide range of structure patterns while maintaining a consistent structure and appearance found in the input sketches. In summary, the contributions of our work are as follows:

- we present a new diffusion-based approach for structured pattern synthesis and expansion;
- we introduce a new medium-scale dataset for fine-tuning generative models on the pattern domain;
- we demonstrate the generation capabilities of our model for different types of structured patterns and show its ability to control the generation precisely from input sketches;
- we validate the improvements over other generative methods, non-specifically trained for patterns, underlying the need for a specifically trained model.

6.2 Method

The work proposed in this chapter is based on the Latent Diffusion (LDM) architecture [Rom+22] adapted to synthesize high-quality stationary, structured patterns with a vector-like appearance. Given a hand-drawn sketch, which serves as the seed for expansion, we extend it to an arbitrary-sized canvas, introducing variations while keeping the overall structure and appearance unchanged. In particular, the initial sketch is centrally placed within a larger canvas, and our model extends the design outward in a process similar to "outpainting" effectively filling the entire frame.

Our approach finetunes an LDM model pre-trained for image generation by training a Low-Rank Adaptation (LoRA) on a dataset of procedurally generated patterns. To improve generation fidelity, we employ an IP-Adapter [Ye+23] for image prompting, ensuring that the generated extension remains true to the original design. To scale patterns to arbitrarily large sizes while maintaining coherence, we employ a combination of noise rolling for tileable pattern generation and latent replication. After a predefined number of diffusion steps, the latent space is manipulated to introduce variations while maintaining structural integrity.

In the following part of this chapter, we first provide an overview of the latent diffusion architecture for image generation and the approaches to combine text and

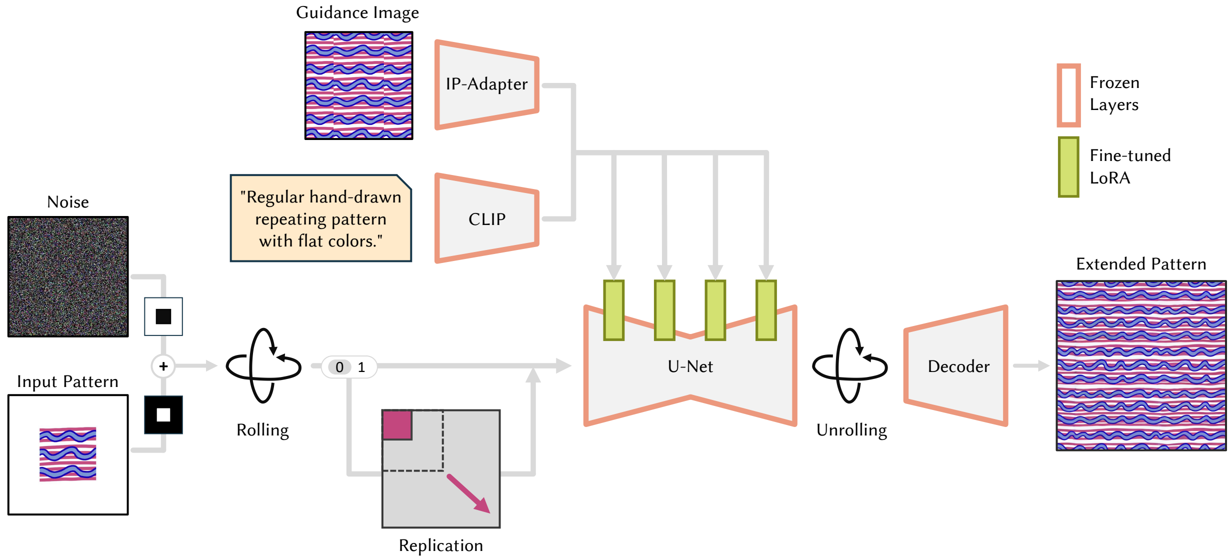


Figure 6.3. Architecture. Given a hand-drawn input pattern, we extend it to an arbitrary-sized canvas, introducing variations while keeping the overall structure and appearance unchanged. Our approach combines text and image conditioning to guide a finetuned Latent Diffusion Model [Rom+22] to generate high-quality consistent patterns. At inference, the input pattern is centrally placed within a larger canvas, with our model extending the design outward in a process similar to "outpainting" and effectively filling the entire frame. To ensure tileability of the output we *roll* the input tensor at each diffusion step and *unroll* afterward. To further extend the pattern beyond the initial canvas scale, we replicate the latent after 60% diffusion steps.

image conditioning, to then detail our approach and architectural choices specific to the structured pattern domain. The general architecture of our work is shown in Fig. 6.3. We ablate our design choices and architectural component in Sec. 6.3.4, demonstrating the benefits of our approach.

6.2.1 Guided Image Generation

Latent Diffusion Model. We leverage the Latent Diffusion architecture, consisting of a Variational Autoencoder (VAE) [KW13] and a diffusion U-Net [Rom+22]. The encoder $\mathcal{E}$, compresses an image $\mathbf{x} \in R^{H\times W\times 3}$ into a latent representation $z = \mathcal{E}(\mathbf{x})$, where $z \in R^{h\times w\times c}$, and c is the dimensionality of the encoded image, capturing the essential features in a lower-dimensional space. The decoder, $\mathcal{D}$, reconstructs the image from this latent space, effectively projecting it back to the pixel space.

The diffusion process involves a series of transformations that gradually denoise a latent vector, guided by a time-conditional U-Net. During training, noised latent vectors are generated, following the strategy defined in [HJA20], through a

deterministic forward diffusion process $q(z_t|z_{t-1})$, transforming the encoding of an input image into an isotropic Gaussian distribution. The diffusion network ϵ_θ is then trained to perform the backward diffusion process $q(z_{t-1}|z_t)$, effectively learning to "denoise" the latent vector and reconstruct its original content.

Text conditioning. Latent Diffusion models can typically be globally conditioned with high-level text prompts via cross-attention [Vas+17] between each convolutional block of the denoising U-Net and the embedding of the condition y, extracted by an encoder τ_θ, with the attention defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \tag{6.1}$$

where $Q = W_Q^i \cdot \tau_\theta(y), K = W_K^i \cdot \varphi_i(z_t), V = W_V^i \varphi_i(z_t)$. Here, $\varphi_i(z_t) \in R^{N \times d_\epsilon^i}$ is the flattened output of the previous convolution block of ϵ_θ, and $W_Q^i \in R^{d \times d_\tau^i}$, $W_K^i \in R^{d \times d_\epsilon^i}$, $W_V^i \in R^{d \times d_\epsilon^i}$, are learnable projection matrices.

The training objective in the conditional setting becomes

$$L_{LDM} := E_{\mathcal{E}(M), y, \epsilon \mathcal{N}(0,1), t}\left[\|\epsilon - \epsilon_\theta(z_t, t, \tau(y))\|_2^2\right] \tag{6.2}$$

Openly available LDM implementations use a pre-trained CLIP [Rad+21] model as feature extractor τ to encode the text condition. In particular, we use the same CLIP encoder as Stable Diffusion v1.5, relying on a ViT model with a patch size of 14×14.

Image conditioning. Despite the expressive capabilities of text, which has shown remarkable results in the context of natural image synthesis, accurately describing a pattern structure with text is challenging since it would require precise definitions of the pattern shapes, their positions and symmetries in relation to the other, and their appearance features.

To provide better control of the synthesized pattern, we propose to combine a high-level text prompt, generally valid for all our patterns, with image conditioning via an IP-Adapter [Ye+23] model. This lightweight adapter achieves image prompting capability, for pretrained text-to-image diffusion models, through a decoupled cross-attention mechanism that separates cross-attention layers for text features and image features. In particular, the adapter computes separate attention for the text and image embeddings, which are then summed before being fed to the next U-Net layer. The output of the new cross-attention is computed as:

$$\text{Atten.}(Q, K_t, V_t, K_i, V_i) = \text{softmax}\left(\frac{QK_t^T}{\sqrt{d}}\right)V_t + \text{softmax}\left(\frac{QK_i^T}{\sqrt{d}}\right)V_i \tag{6.3}$$

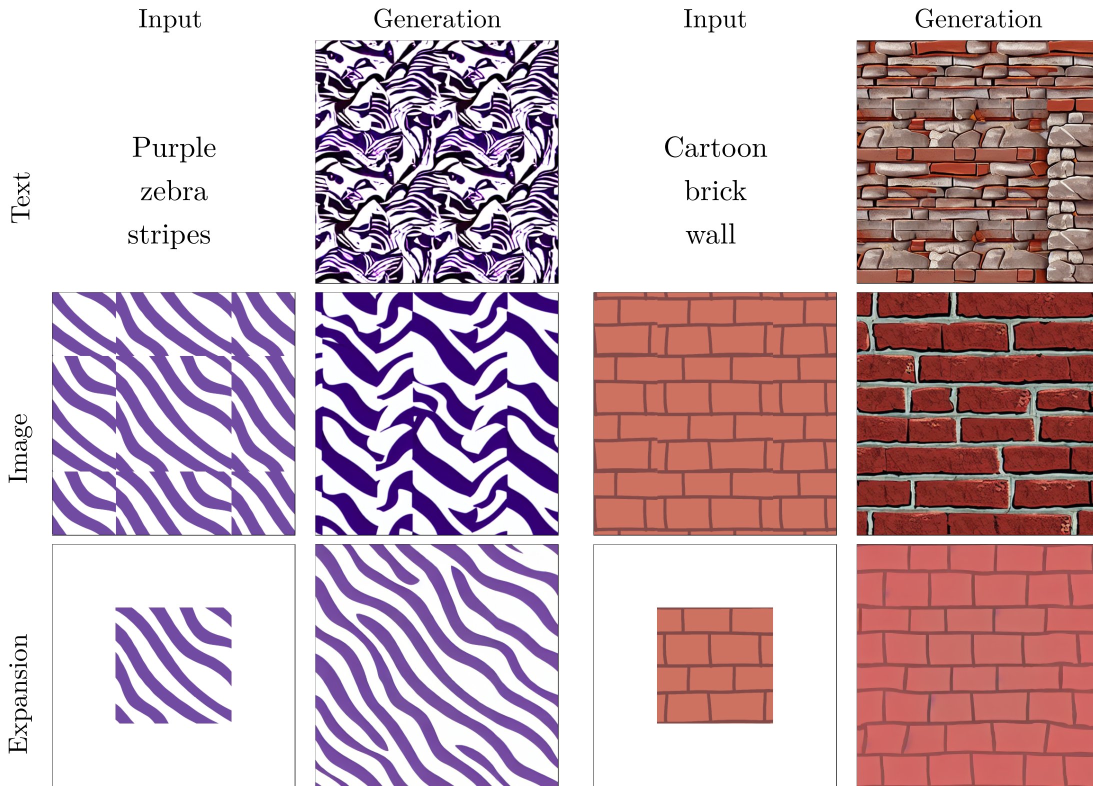


Figure 6.4. Comparison of the generation modalities. We show the different generation modalities that our base architecture support. While global conditioning via text or image only (first two rows) is an option, it struggles in generating a consistent pattern and lack fine-grained user control. In contrast, our expansion approach (last row) enables generation of arbitrarly large patterns ensuring consistency with the provided input.

with K_t, V_t, K_i, V_i being respectively the keys and values for the text and image embeddings. During the training of the IP-Adapter, only the image cross-attention layers are trained, while the rest of the diffusion model is kept frozen.

This approach has shown remarkable performances in controlling the generation process with image prompts, allowing it to closely follow the reference image.

6.2.2 Stable Diffusion Finetuning

To achieve visually coherent pattern synthesis and expansion, we fine-tune the Stable Diffusion 1.5 model [Rom+22] to our specific pattern domain. In particular, we leverage a Low-Rank Adaptation (LoRA) technique [Hu+21a] for efficient fine-tuning of a large, pre-trained model, limiting the number of training parameters, while avoiding catastrophic forgetting.

In particular, we train low-rank matrices into the transformer layers of the base

Latent Diffusion Model (LDM):

$$\theta' = \theta + \Delta\theta, \tag{6.4}$$

where θ represents the original weights of the transformer in the LDM, and $\Delta\theta$ is the low-rank update, computed as:

$$\Delta\theta = U \cdot V^T, \tag{6.5}$$

with $U \in R^{r \times d}$ being the trainable matrices, and r much smaller than d, the dimensionality of the layer's parameters.

This fine-tuning step focuses the generation on the pattern domain and is mostly responsible for the model's ability to maintain stylistic consistency and detailed coherence specific to the target patterns. By introducing these low-rank updates, we ensure that the model adapts efficiently to the specific feature of the pattern domain, without losing its expressive capabilities from the training on the image domain.

6.2.3 Patterns expansion

To expand patterns to an arbitrary size, while ensuring their aesthetic coherence and tileability, we base our architecture on the Stable Diffusion 1.5 model specifically trained for image inpainting [Rom+22]. We leverage its ability to understand the context of partial images and generate coherent completions and combine it with latent replication and *noise rolling* [Vec+23], to produce high-quality, tileable expansions. This ensures that the expanded patterns remain consistent with the original input, preserving the overall visual appearance as shown in Fig. 6.4 and Fig. 6.5. In particular, we place our input pattern at the center of the canvas and use the inpainting capabilities of SD to fill the missing area in an outpainting fashion. However, while inpainting alone can reconstruct the missing part, it tends to lose long-term dependencies inside the image, leading to inconsistencies in the global structure. This limitation arises because inpainting primarily focuses on local continuity without adequately preserving the broader context and relationships within the image. To mitigate this issue, while also enabling tileable generation, we employ *noise rolling*. At inference time, the latent representation z_t of the pattern at a particular timestep t is cyclically shifted spatially. In particular, for each diffusion step, we compute:

$$z'_t = \text{roll}(z_t, \Delta x, \Delta y), \tag{6.6}$$

where $\text{roll}(\cdot, \Delta x, \Delta y)$ denotes the cyclic shift operation along the image's width (Δx) and height (Δy). After rolling, the model estimates the noise component and performs a denoising step, computing z'_{t-1}. Subsequently, the latent space is unrolled

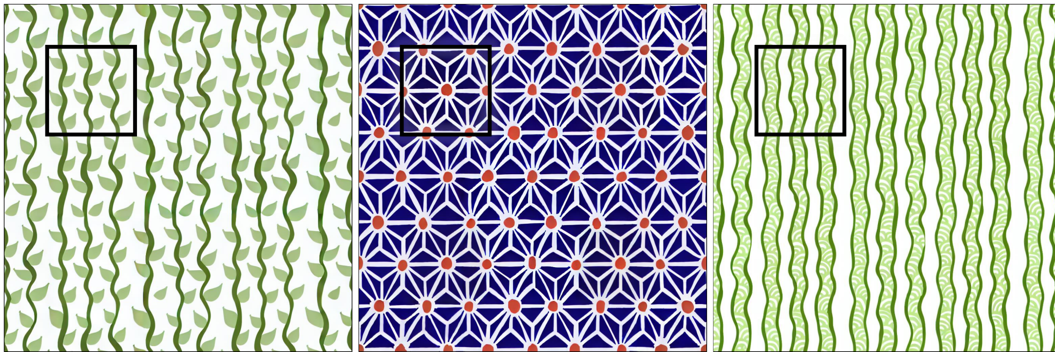


Figure 6.5. Pattern expansion. The figure illustrates the pattern expansion capabilities of our model, where the input patterns are contained within the black boxes. The left and right sample showcase organic patterns, demonstrating the model's ability to extrapolate intricate designs while maintaining a natural flow. The central panel presents a more structured geometric pattern which is also successfully expanded to cover a larger area. This example highlights the model's versatility in handling both organic and geometric patterns.

back to its original configuration to maintain the integrity of the global pattern structure:

$$z_{t-1} = \text{roll}(z'_{t-1}, -\Delta x, -\Delta y). \tag{6.7}$$

By manipulating the latent space in this manner, the model effectively treats the pattern's edges as interconnected, removing any visible border.

Finally, to be able to cover an arbitrarily large canvas, we start the denoising process at the model's native resolution of 512×512, placing the input at the center of the canvas, and filling the remaining area. Then, after N denoising steps, we replicate the latent with the minimum factor to cover the target size and denoise using patched diffusion for the remaining steps. In our experiments, we set N to the 60% of the inference steps, resulting in the best compromise between pattern consistency and variation. By combining latent replication and noise rolling, we support a larger expansion while guaranteeing the quality of the generated patterns, as demonstrated in Sec. 6.3 and in the ablation study in Sec. 6.3.4.

6.3 Experimental results

6.3.1 Datasets

Due to the lack of publicly available pattern datasets, we created a custom dataset consisting of 4000 patterns of 8 classes, with some of them showcased in Fig. 6.6. Such classes were specifically designed to expose strong geometrical structures, like checkered patterns, stripes, grids, and polka dot arrangements, to help the LoRA to

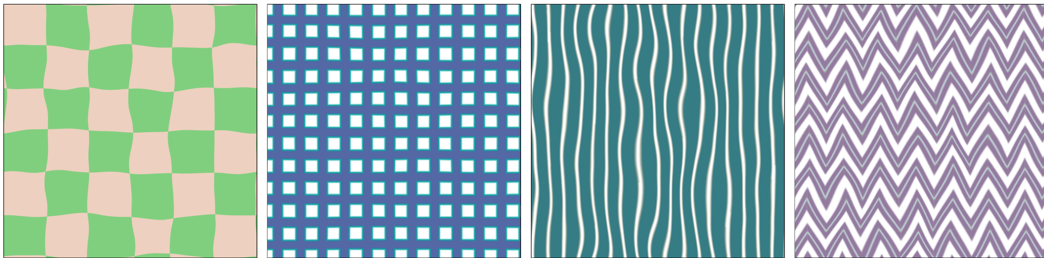


Figure 6.6. Pattern samples from the dataset. Samples extracted from four pattern classes, namely *grid*, *checker*, *stripes* and *zigzag*. For each pattern, we use the generating procedural parameters to define a caption matching its design details, thus creating a pattern-text pair used while training the LoRA.

learn the features that characterize the structured pattern domain the most. For each class, we defined an ad-hoc procedural program capable of creating a set of samples with a great variety in both design and colors. To enhance the sketched style, we also combined our patterns with different scales of Perlin noise [Per85], thus introducing the irregularities that are commonly found in hand-drawn designs.

Our dataset consists of procedurally generated pattern-text captions pairs. We generated each pattern by randomly sampling a value in the proper range for each exposed procedural parameter, including exposed colors. For each pattern, the values used in the procedural program are also employed to produce a caption highlighting some of its details. For each class, we designed a base caption structure that is filled with details drawn from the procedural parameter values. As an example, the caption matching the checkered pattern in Fig. 6.6 (left) is generated from the base caption of *"A hand-drawn checkered pattern. Checkers are colored in* `<even_color>` *and* `<odd_color>`*, and their size is* `<checker_size>`*. Checkers are surrounded on all four sides by a checker of a different color. Colors are flat and without shading."*, where the free variables are completed by "light green", "wheat" and "big" respectively.

For each class, we sample 500 different parameter sets and generated the corresponding pattern-text pair for training.

6.3.2 Technical details

Training. We train our LoRA with a mini-batch gradient descent, using the Adam [KB14] optimizer with a learning rate set to 10^{-4} and a batch size of 8. The training is carried out for 5000 iterations on a single NVIDIA RTX3090 GPU with 24GB of VRAM, using the pre-trained inpainting Stable Diffusion 1.5 checkpoint from [Rom+22].

Inference. Generation is performed by denoising a latent random noise for 50 steps, using the DDIM sampler [SME20] with a fixed seed. Pattern expansion takes about 2 seconds at 512×512 and 4 seconds at 1024×1024 and 6GB of VRAM, about 12 seconds at 2048×2048 and 8GB VRAM. Memory and requirements can be further reduced by processing fewer patches in parallel, albeit at the cost of increased computation time.

6.3.3 Results and comparisons

We evaluate *pAff* generation capabilities when conditioning using either text or image. Despite not being the main focus of this work, we show that our architecture is able to generate pattern-like images when being globally conditioned. However, as shown in Fig. 6.4, generation style tends to significantly diverge from the guidance image, highlighting the need for stronger constraining for a specific pattern expansion that closely follows the input sample.

Expansion results. We evaluate our model generation capabilities for pattern expansion (Fig. 6.5 and Fig. 6.10). The results demonstrate our method to closely follow the input prompt, producing high-quality, coherent pattern expansion.

Fig. 6.5 highlights the pattern expansion capabilities of our model. The input patterns are contained within the black boxes, while the surrounding patterns are generated by the model. The model successfully extends the input pattern, maintaining coherence and preserving the structural integrity of the original designs. Each generated pattern flows naturally from the input, ensuring that there are no abrupt transitions or noticeable repetitions. The model keeps color consistent in the generated area, matching the original input. Due to the adoption of the *noise rolling* technique, all results are tileable, thus allowing seamless repetition of each generation. We show more large-scale expansions in Fig. 6.10. All examples use an expansion factor of 2 in both width and height dimensions.

Comparison. We compare our method, against a series of established techniques including [Hei+21], GCD [Zho+23], MatFuse [Vec+24], and [Zho+24] as depicted in Fig. 6.7. For each method, we use the official code and weights released by the authors. As MatFuse [Vec+24] is trained to generate PBR materials, we provide the pattern as the diffuse component of the material, initializing the other properties at a default value.

pAff notably improves with respect to previous approaches in preserving the structural integrity and visual fidelity of patterns. Both [Hei+21] and GCD, while capturing the visual features of the patterns, tend to break the overall structure, introducing unnatural distortions that result in a loss of pattern detail and clarity.

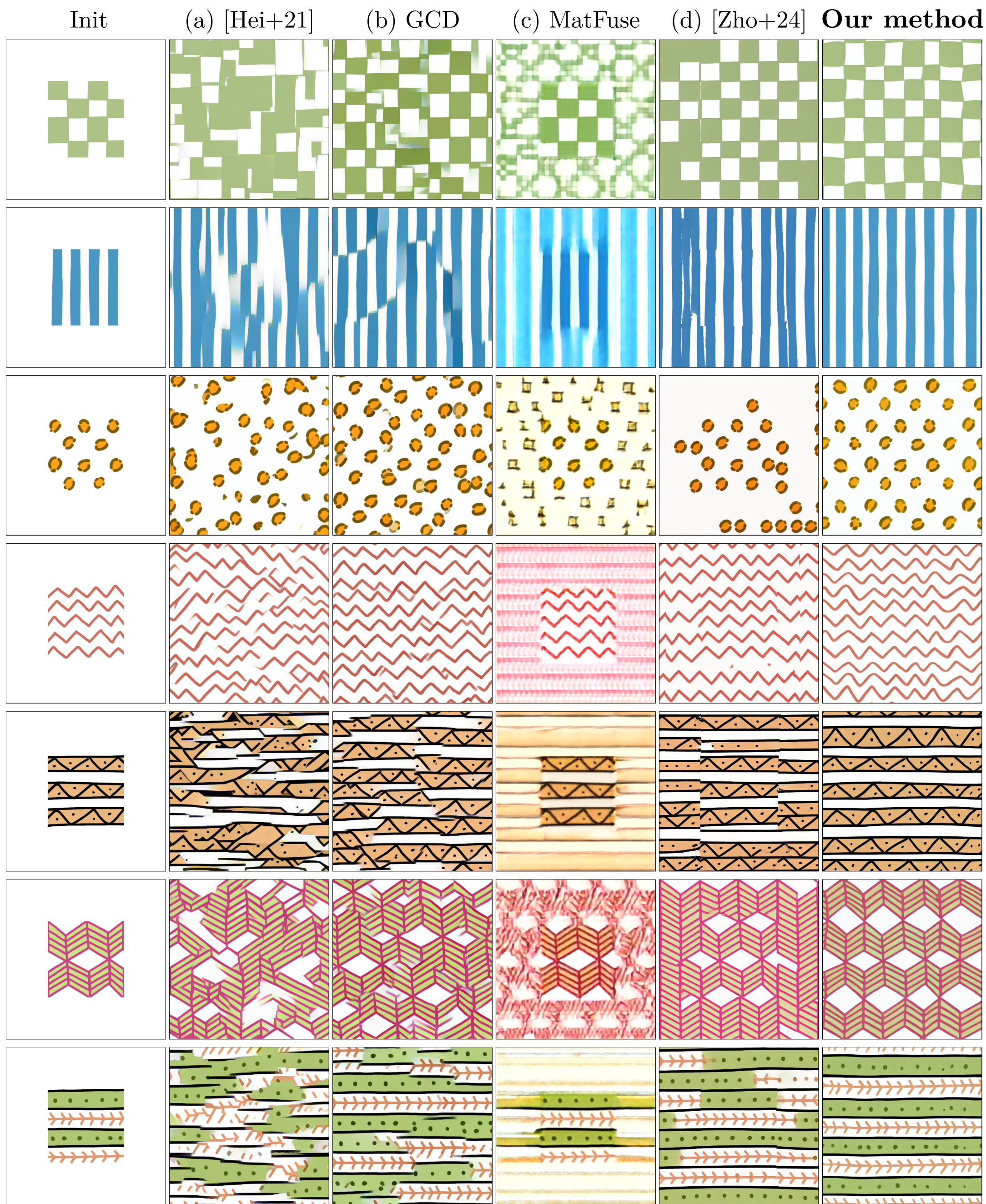


Figure 6.7. We compared our method with established exture and material synthesis techniques. As previous work tends to break the overall structure (a, b, d) or fails at reconstructing the pattern appearance (c), our method consistently expands the input by preserving structural integrity and input coherency.

MatFuse fails to capture the appearance of the pattern, mostly due to the training on natural textures, being only able to reproduce the colors and general shape of the pattern but lacking any sharpness or detail. [Zho+24], in contrast, is generally

able to reproduce the sharp visual appearance of the pattern, and capture the main features; however, due to its main focus on non-stationary textures, it tends to break the overall structure, resulting in sharp discontinuity edges inside the image and transitioning between different parts of the pattern. Additionally, it struggles with very sparse patterns (e.g., third row in Fig. 6.7), and introduces a color shift on the original input. None of the other methods produces tileable results. Compared to the other approaches, our work can capture the visual features of the pattern and extend it seamlessly, introducing slight variations without altering the overall structure. All results are tileable, thanks to the use of noise rolling at inference time.

6.3.4 Ablation Study

We evaluate our design choices starting from the baseline solution and gradually introducing the different proposed architectural components and diffusion elements –IP-Adapter, LoRA finetuning, noise-rolling–. To systematically assess the impact of each component, we test the different configurations on a series of example patterns. We provide qualitative results of the ablation study in Fig. 6.8.

We firstly evaluate the Stable Diffusion base model performing a text-guided inpainting task. This sets a performance baseline without being influenced by any of the design choices that characterize our method. Although the model is able to fill in the missing areas, it tends to diverge from the input condition and break the overall structure. Even for simple examples, the text-guided approach is not a natural mean to express pattern structures such as shapes and arrangements, and moreover, it is not versatile enough to perfectly describe the design of the partial input pattern.

To provide control in a more natural way, we include an IP-Adapter [Ye+23] that introduces an image prompt as further guidance for the inpainting process. The guidance image is constructed by simply repeating the image prompt multiple times to fill a 512×512 canvas. As described in Sec. 6.3.2 this helps the CLIP encoder better capture the visual features and sharpness of the guidance, due to the high sensitivity of CLIP to image resolution [WCL23]. As shown in our results, visual guidance allows the model to better follow the input, while still presenting some visual inconsistencies and limitations, mostly due to the training on natural images. In fact, the model is capable of better catching the style and colors provided by the guidance image, but it still fails at reconstructing its geometrical details in both shape features or pattern scale and often provides natural-looking results.

Since the model is more exposed to photorealistic, natural, and unstructured data during training, we perform a fine-tuning on the structured patterns to better adapt it to this new domain and task. To do so, we trained a LoRA module on our

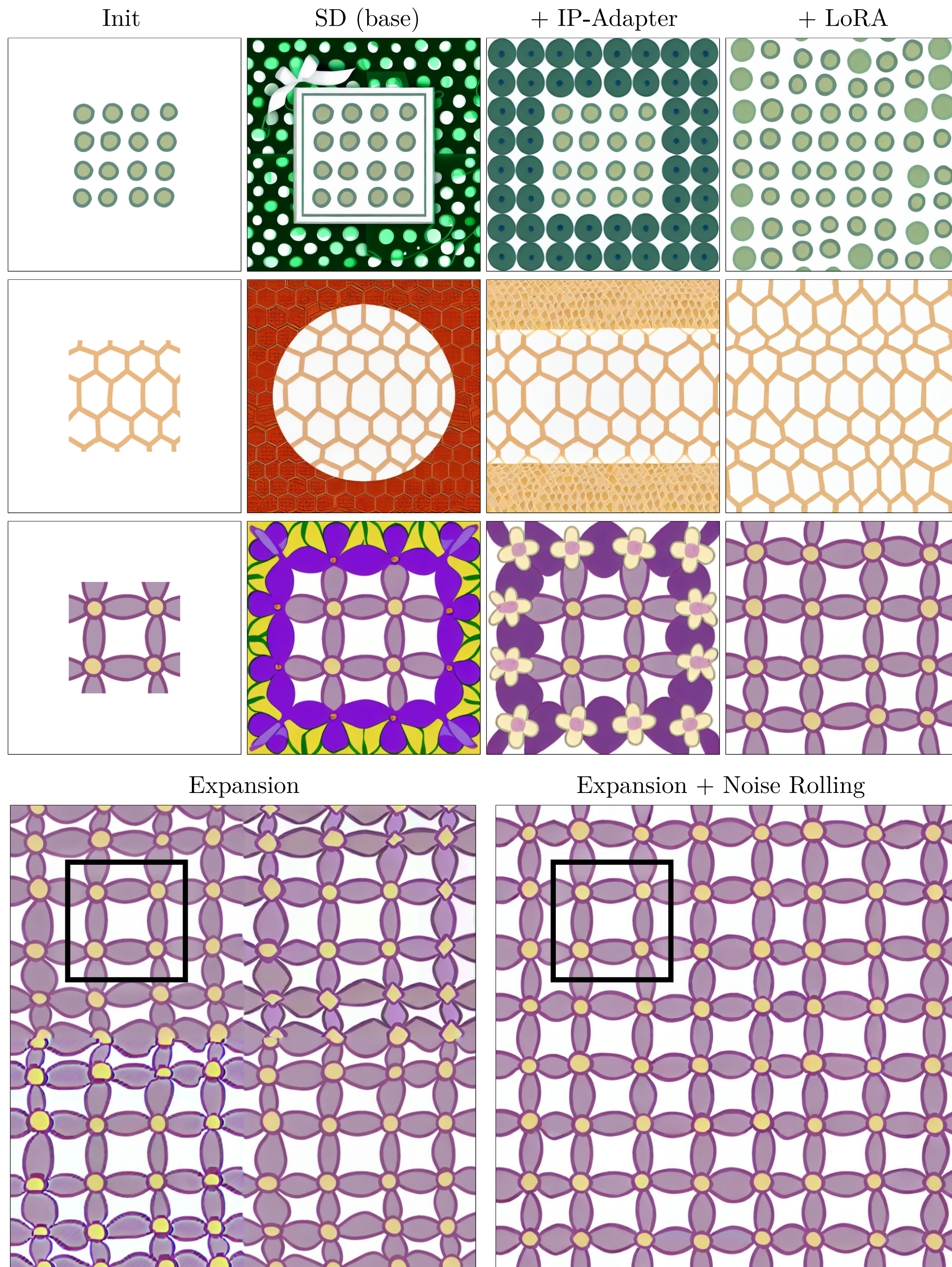


Figure 6.8. Ablation study. We show the performance improvements with the introduction of each design choice. The base inpainting model is unable to expand the pattern while keeping visual coherence. The introduction of image conditioning via the IP-Adapter improves the generation consistency with the prompt. The LoRA finetuning, on a small dataset, greately enhances generation quality, ensuring visual consistency over the entire generation. Finally, the introduction of noise rolling enables tileable generation and removes repetition seams and visual artifacts.

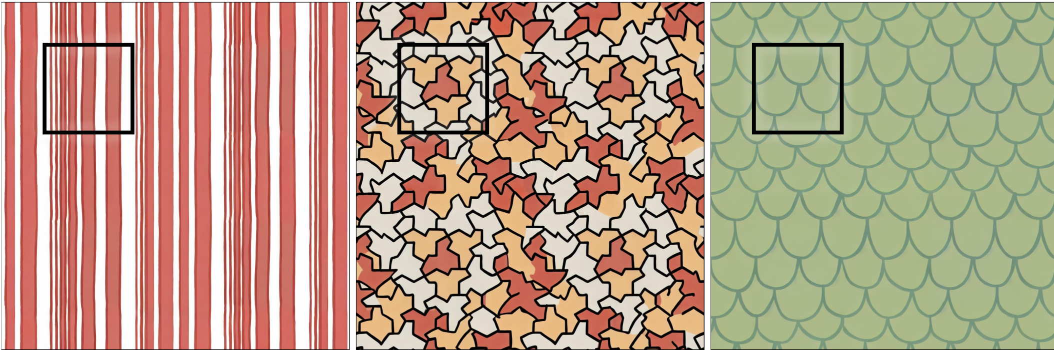


Figure 6.9. Limitations. Our model presents some limitations, which can be categorized into domain-specific and architectural limitations. First, it is by design unable to expand non-repeating patterns, either non-stationary or aperiodic [Smi+23] (left and center). Additionally, it can fail to generate very structured patterns at a consistent scale while ensuring tilebility, thus squeezing or distorting the pattern to make it fit the canvas (right).

crafted pattern dataset. By combining the LoRA domain knowledge with the Stable Diffusion Model backbone, we noticed that the overall result quality and consistency are significantly improved, thanks to the new adaptation to the pattern domain. In particular, results preserve the same style as the provided input and reconstruct geometrical details and arrangements in a more resilient way.

Despite good results that could be achieved on small expansions, we notice a deterioration of the output for higher expansion factors. As reported in the right-most images of Fig.6.8, the expansion tends to produce a degraded output that influences the style and the structure, in terms of color artifacts and discontinuities in the pattern respectively. The introduction of the *noise rolling* technique enables us to produce results that correctly integrate the provided image by maintaining both the visual and geometrical aspects. In particular, this addition has a twofold effect: it makes the generated pattern tileable, removing edge discontinuities, and it helps in better capturing long-range dependencies inside the image, thus allowing us to increase the expansion factor without losing quality.

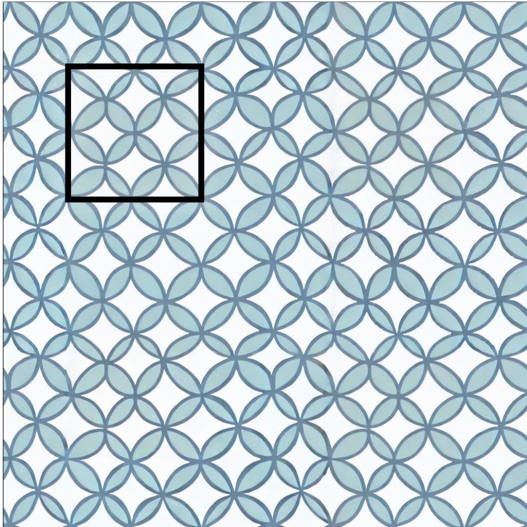
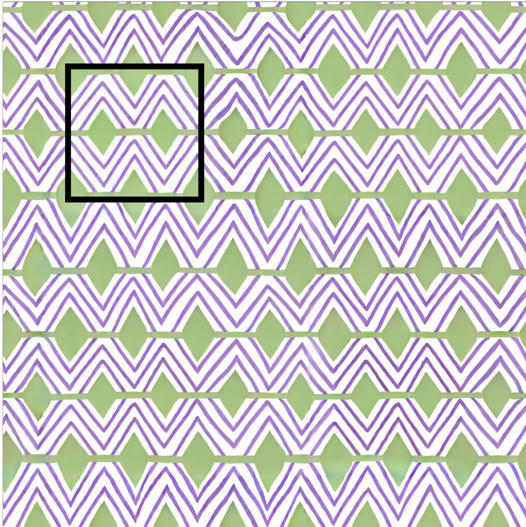
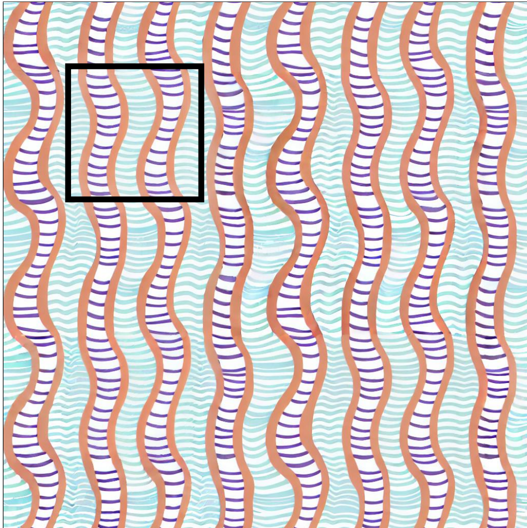
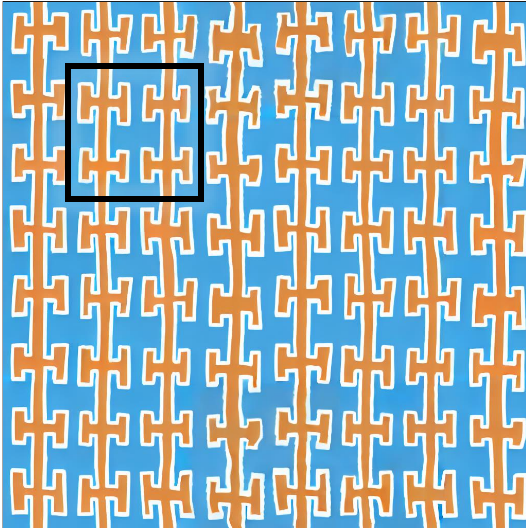
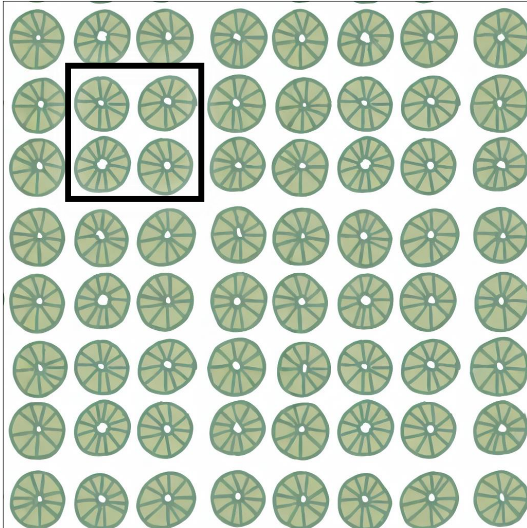
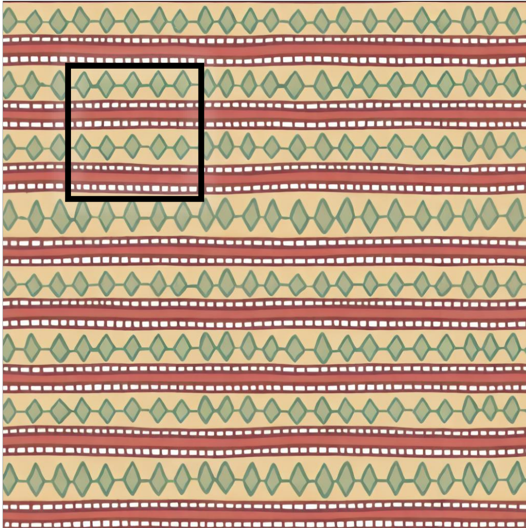
6.4 Limitations and Future Work

Our method comes with some limitations that we can divide between architectural limitations and domain limitations. Examples of failure cases or unexpected behavior are presented in Fig. 6.9. As discussed in previous sections, our method cannot faithfully expand non-repeating patterns, either non-stationary (Fig. 6.9 left), or aperiodic [Smi+23] (Fig. 6.9 center). This limitation comes from the design choices of our approach, which focus on repeating patterns. While both expansions present

plausible patterns, they don't necessarily follow the expected behavior, where the lines in the first figure should keep growing, while the tiles should not present a predictable pattern. Future work could focus on tackling non-repeating patterns by injecting, into the generation, additional information, in the form of conditioning about the pattern' repetitiveness. The last failure case (Fig. 6.9 right), on the other hand, shows the design limitations of our approach, which can fail to generate very structured patterns at a consistent scale in the presence of tilebility. This is related to the noise rolling which enforces tileability on the border of the image, thus squeezing the border shingles to make them fit the canvas. Possible improvements could involve an automated solution to find the optimal crop of the pattern [Rod+24] before beginning the expansion.

6.5 Conclusion

We presented *pAff*, a diffusion-based architecture for structured pattern expansion, with a focus on controllability of the generated pattern. We demonstrated the expansion of several patterns in the class with distinctly different structures, symmetries, and appearance. Our results show the robustness of the proposed architecture and its controllability, while the comparison with prior work shows that our method is significant over the state of the art.

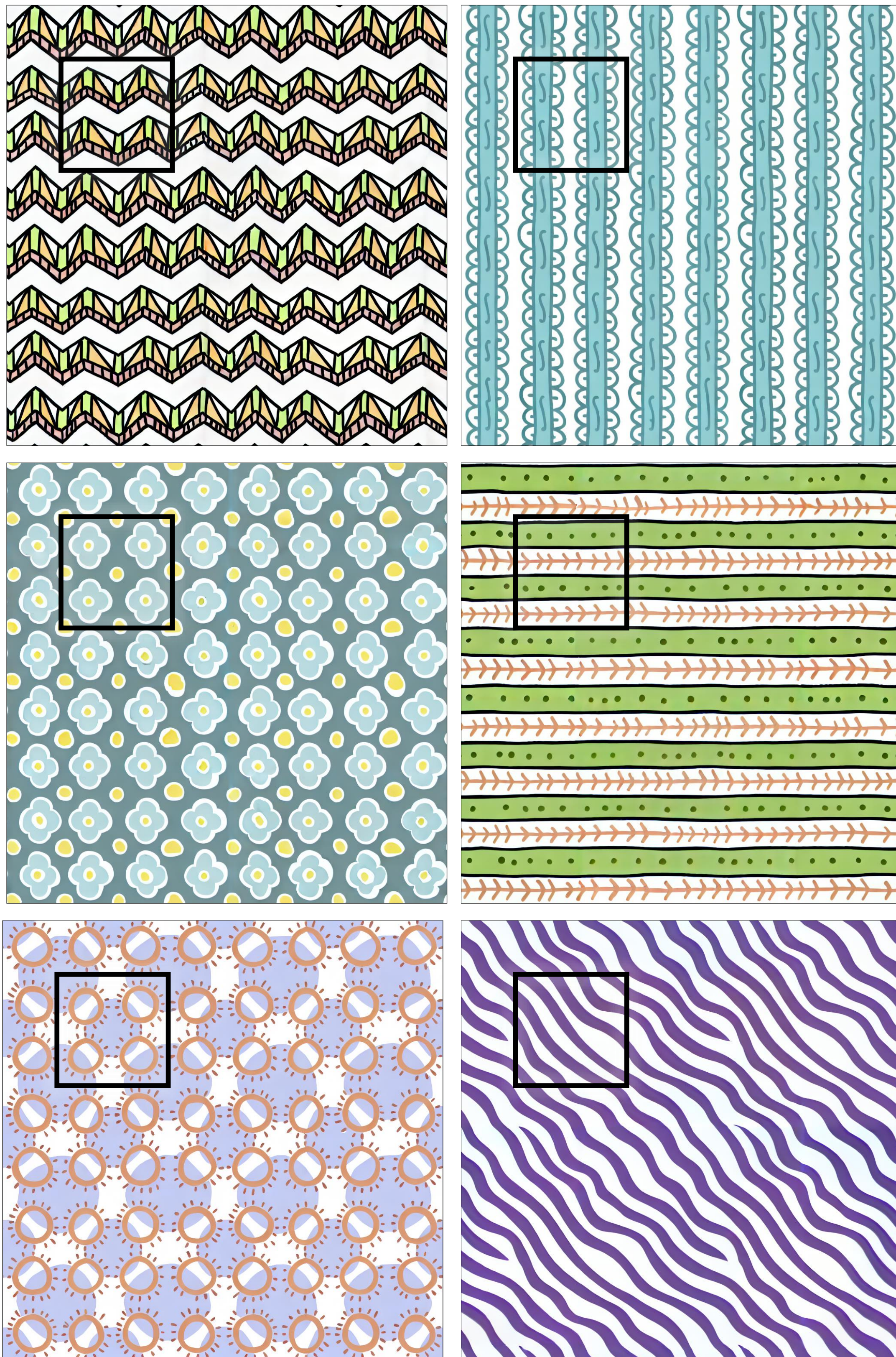


Figure 6.10. Our diffusion-based pattern expansion method enables the generation of large-scale, high-quality and tileable patterns from a small user-drawn input, reported in the black boxes. By being fine-tuned on domain-specific data, it adapts to different structured arrangements of solid-colored shapes, consistently extending the input design features to a larger-scale result.

Chapter 7

Conclusion

In this thesis, we explored procedural and non-procedural asset editing, with the aim of enhancing its controllability to ease the design process for both novice and experienced users.

Procedural assets are used in computer graphics applications since variations can be obtained by changing the parameters of the procedural programs, thus increasing variability without having to model each asset from scratch separately. However, as the number of parameters increases, editing becomes a cumbersome operation 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, direct manipulation approaches let users manipulate the output asset interactively, while the system determines the procedural parameters that match the performed modification. In this thesis, we mainly focused on 2D structured textures, namely procedural vector patterns, and on 3D models expressed as procedural implicit surfaces.

Concerning the former, we propose **pOp** and **pEt**, two complementary approaches whose union uplifts the process of editing procedural vector patterns under the assumption of differentiability with reference to the exposed procedural parameters. In particular, *pOp* is an example-based editing tool that estimates procedural parameter values that better match the target appearance provided by users in terms of images like renders or even sketches. Instead of solving an inverse rendering problem, *pOp* relies on an inverse signed distance field approach, thus optimizing for the procedural parameters by minimizing a function computing a per-color pattern signed distance field loss. On the other hand, *pEt* proposes a direct manipulation tool for differentiable vector patterns, enabling users to interact with them using a click-and-drag interaction schema. By defining an identifier for each selected point, *pEt* solves for the procedural parameters in a gradient descent-based optimization loop, thus computing an update in the procedural parameter values that is coherent

with the edit expressed by the user directly in the viewport. Using these approaches in a hybrid mode, artists can firstly identify a good initial assignment for each parameter value by providing a sketch target, and refine their values by directly manipulating the procedural vector pattern to get a more precise result. Focusing on a different procedural asset for 3D modeling, we propose a **direct manipulation tool** specifically designed for **implicit surfaces**. By defining a way of uniquely identifying points on an implicit surface via a co-parameterization function, our system enabled the selection of patches of the surface that can be arbitrarily moved around in the viewport. The underlying gradient descent-based optimization loop estimates a parameter update that coherently expresses the change suggested by the user at interactive time rates, enabling users to visualize the outcome of their edit as the dragging movement is performed. Our framework correctly handles regular primitives and modeling operations, such as affine transformations. It also supports operations that are widely adopted in implicit surface modeling like CSG Booleans and Smooth Booleans, being resilient to changes in topology, as well as warping operations.

Lastly, concerning non-procedural asset editing, this thesis proposes **pAff**, a model that leverages novel generative applications and extends them to the domain of structured patterns, oppositely to widespread natural and non-stochastic content creation. This model takes as input a hand-drawn user sketch depicting a desired pattern and extends it to produce a larger-scale, high-quality, and tileable result, without having the user design the entire canvas and still keeping a high fidelity with the provided input.

7.1 Future Directions

In this work of thesis, we explored how to enhance user control in common design workflows for both inverse procedural asset modeling and non-procedural asset synthesis. As regards the former task, this thesis mainly focuses on parameter estimation, but it does not explore procedural program estimation, which is still a challenging and yet complex task in the wider inverse procedural modeling field. So, future work may explore this direction, providing tools that are capable of estimating the structure of a procedural generator in terms of programs or graphs, in an example-based scenario as well as through direct manipulation occurring on target assets.

Similarly, we can also assess the limitations of the methods proposed in this thesis as possible directions for future explorations. As an example, new formulations can be investigated to handle properties, such as opacity, in *pOp* framework as well as domain repetition nodes in the procedural implicit surface direct modeling tool.

Optimization can also be strengthened, providing a suitable definition of the co-parameterization function, assessing injectivity in corner cases, and a more consistent interpolation. The gradient descent optimization algorithm may be improved using other optimization techniques like ADAM or momentum and more emphasis could be given to extracting information from the points or patches selected by users, as they could hypothetically suggest a possible update in certain parameters without having to update all of them simultaneously.

As regards generative-based pattern expansion, further training and fine-tuning can be performed to handle unexpected or unfaithful behaviour in generation, increasing the families of structured data correctly supported by our expansion method. Improving guidance by identifying a good repeatable tile in the input proposed by the user may help in generating patterns in a more reliable way, possibly supporting spatially varying features, a more refined tileability constraint may be adopted to reduce scale artifacts and additional information may be injected into the generation, thus improving conditioning mechanisms to support non-repeating patterns as well.

In conclusion, this thesis firstly assesses challenges in procedural and non-procedural asset editing, proposing novel methods for simplifying 2D texture modeling as well as 3D implicit surface design and representing a step forward in the simplification of such time-consuming workflows. The methods proposed in this thesis could be extended to other assets and integrated into common 2D or 3D editing software, easing the design process.

References

[ADO] ADOBE. *Adobe Substance 3D Designer*. https://www.adobe.com/products/substance3d-designer.html.

[Ali+16] Daniel Aliaga, Ilke Demir, Bedrich Benes, and Michael Wand. "Inverse procedural modeling of 3D models for virtual worlds". In: *ACM SIGGRAPH Courses*. July 2016, pp. 1–316.

[Ang+17] Baptiste Angles, Marco Tarini, Brian Wyvill, Loıc Barthe, and Andrea Tagliasacchi. "Sketch-Based Implicit Blending". In: *ACM Trans. Graph.* 36.6 (2017).

[AL10] Ken-ichi Anjyo and John Lewis. "Direct Manipulation Blendshapes". In: *IEEE Computer Graphics and Applications* 30.04 (July 2010), pp. 42–50.

[Ari+18] Andreas Aristidou, Joan Lasenby, Yiorgos Chrysanthou, and Ariel Shamir. "Inverse Kinematics Techniques in Computer Graphics: A Survey". In: *Computer Graphics Forum* 37 (2018).

[ACB17] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks". In: *International conference on machine learning*. PMLR. 2017, pp. 214–223.

[Bar+23] Omer Bar-Tal, Lior Yariv, Yaron Lipman, and Tali Dekel. "MultiDiffusion: Fusing Diffusion Paths for Controlled Image Generation". In: *arXiv preprint arXiv:2302.08113* 2 (2023).

[BGA05] Aurélien Barbier, Eric Galin, and Samir Akkouche. "A framework for modeling, animating, and morphing textured implicit models". In: *Graphical Models* 67.3 (2005), pp. 166–188.

[Bar+06] Pascal Barla, Simon Breslav, Joëlle Thollot, François X. Sillion, and Lee Markosian. "Stroke Pattern Analysis and Synthesis". In: *Proc. of Eurographics 2006 : Computer Graphics Forum*. Vol. 25. 663-671. ACM, 2006.

[BB89] Richard H. Bartels and John C. Beatty. “A Technique for Direct Manipulation of Spline Curves”. In: *Proceedings of Graphics Interface ’89*. GI ’89. 1989, pp. 33–39.

[BWS10] Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. “A connection between partial symmetry and inverse procedural modeling”. In: *ACM SIGGRAPH 2010 papers* (2010).

[BM96] Ronan Boulic and Ramon Mas. “Hierarchical Kinematic Behaviors for Complex Articulated Figures”. In: *Interactive Computer Animation*. USA: Prentice-Hall, Inc., 1996, pp. 40–70. ISBN: 013518309X.

[BDS18] Andrew Brock, Jeff Donahue, and Karen Simonyan. “Large scale GAN training for high fidelity natural image synthesis”. In: *arXiv preprint arXiv:1809.11096* (2018).

[Cas+22] Dan Cascaval, Mira Shalah, Philip. Quinn, Rastislav. Bodik, Maneesh. Agrawala, and Adriana Schulz. “Differentiable 3D CAD Programs for Bidirectional Editing”. In: *Computer Graphics Forum* 41.2 (2022), pp. 309–323.

[DN21] Prafulla Dhariwal and Alexander Nichol. “Diffusion models beat gans on image synthesis”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 8780–8794.

[DB15] P. Kingma Diederik and Jimmy L. Ba. “Adam: A Method for Stochastic Optimization”. In: *ICLR 2015*. 2015.

[Du+18] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. “InverseCSG: Automatic conversion of 3d models to CSG trees”. In: *ACM Transactions on Graphics (TOG)* 37.6 (2018), pp. 1–16. DOI: 10.1145/3272127.3275006.

[EF01] Alexei A. Efros and William T. Freeman. “Image Quilting for Texture Synthesis and Transfer”. In: *CVPR*. 2001, pp. 341–346.

[EL99] Alexei A. Efros and Thomas K. Leung. “Texture synthesis by non-parametric sampling”. In: *CVPR*. Vol. 2. 1999, pp. 1033–1038.

[For96] Stephanie Forrest. “Genetic algorithms”. In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 77–80.

[For+12] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. “DEAP: Evolutionary Algorithms Made Easy”. In: *Journal of Machine Learning Research* 13 (2012), pp. 2171–2175.

[Gai+22] Mathieu Gaillard, Vojtech Krs, Giorgio Gori, Radomir Mech, and Bedrich Benes. “Automatic Differentiable Procedural Modeling”. In: *Computer Graphics Forum* 41 (May 2022), pp. 289–307.

[GEB15] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. “Texture Synthesis Using Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 28. 2015.

[GEB16] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. “Image Style Transfer Using Convolutional Neural Networks”. In: *Proc. of CVPR*. June 2016.

[Gie+21] Lena Gieseke, Paul Asente, Radomír Měch, Bedrich Benes, and Martin Fuchs. “A Survey of Control Mechanisms for Creative Pattern Generation”. In: *Computer Graphics Forum* 40.2 (2021), pp. 585–609.

[Gle94] Michael Lee Gleicher. *A differential approach to graphical interaction*. Carnegie Mellon University, 1994.

[Goo+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. “Generative Adversarial Nets”. In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger. Vol. 27. Curran Associates, Inc., 2014.

[Gue+16] Paul Guerrero, Gilbert Bernstein, Wilmot Li, and Niloy J. Mitra. “PATEX: Exploring Pattern Variations”. In: *ACM Trans. Graph.* 35.4 (2016).

[Gue+22] Paul Guerrero, Milos Hasan, Kalyan Sunkavalli, Radomir Mech, Tamy Boubekeur, and Niloy J. Mitra. “MatFormer: A Generative Model for Procedural Materials”. In: *ACM Trans. Graph.* 41.4 (2022).

[Gul+17] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. “Improved training of wasserstein gans”. In: *Advances in neural information processing systems* 30 (2017).

[Guo+20] Jianwei Guo, Haiyong Jiang, Bedrich Benes, Oliver Deussen, Xiaopeng Zhang, Dani Lischinski, and Hui Huang. “Inverse Procedural Modeling of Branching Structures by Inferring L-Systems”. In: *ACM Transactions on Graphics* 39.5 (2020), 155:1–155:13.

[Guo+19] Yu Guo, Milos Hasan, Ling-Qi Yan, and Shuang Zhao. “A Bayesian Inference Framework for Procedural Material Parameter Estimation”. In: *Computer Graphics Forum* 39 (2019).

[Har96] John C. Hart. "Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces". In: *The Visual Computer* 12.10 (1996), pp. 527–545.

[He+23] Zhen He, Jie Guo, Yan Zhang, Qinghao Tu, Mufan Chen, Yanwen Guo, Pengyu Wang, and Wei Dai. "Text2Mat: Generating Materials from Text". In: *Pacific Graphics Short Papers and Posters*. Ed. by Raphaëlle Chaine, Zhigang Deng, and Min H. Kim. The Eurographics Association, 2023. ISBN: 978-3-03868-234-9. DOI: 10.2312/pg.20231275.

[Hei+21] Eric Heitz, Kenneth Vanhoey, Thomas Chambon, and Laurent Belcour. "A sliced wasserstein loss for neural texture synthesis". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 9412–9420.

[HLC19] Brian Hempel, Justin Lubin, and Ravi Chugh. "Sketch-n-Sketch: Output-Directed Programming for SVG". In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (2019).

[HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. "Denoising diffusion probabilistic models". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 6840–6851.

[HS22] Jonathan Ho and Tim Salimans. "Classifier-free diffusion guidance". In: *arXiv preprint arXiv:2207.12598* (2022).

[Hu+21a] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. "Lora: Low-rank adaptation of large language models". In: *arXiv preprint arXiv:2106.09685* (2021).

[HDR19] Yiwei Hu, Julie Dorsey, and Holly E. Rushmeier. "A novel framework for inverse procedural texture modeling". In: *ACM Trans. Graph. (TOG)* 38 (2019), pp. 1–14.

[Hu+22a] Yiwei Hu, Paul Guerrero, Milos Hasan, Holly Rushmeier, and Valentin Deschaintre. "Node Graph Optimization Using Differentiable Proxies". In: *ACM SIGGRAPH 2022 Conference Proceedings*. SIGGRAPH '22. 2022.

[Hu+22b] Yiwei Hu, Miloš Hašan, Paul Guerrero, Holly Rushmeier, and Valentin Deschaintre. "Controlling Material Appearance by Examples". In: *Computer Graphics Forum* 41.4 (2022), pp. 117–128.

[Hu+21b] Yiwei Hu, Chen He, Valentin Deschaintre, Julie Dorsey, and Holly E. Rushmeier. "An Inverse Procedural Modeling Pipeline for SVBRDF Maps". In: *ACM Transactions on Graphics (TOG)* 41 (2021), pp. 1–17.

[Hur+09] Thomas Hurtut, Pierre-Edouard Landes, Joëlle Thollot, Yann Gousseau, Rémy Drouilhet, and Jean-François Coeurjolly. "Appearance-guided Synthesis of Element Arrangements by Example". In: *Proc. of the 7th International Symposium on Non-photorealistic Animation and Rendering (NPAR'09).* ACM Press, 2009, pp. 51–60.

[IMH05] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. "As-Rigid-as-Possible Shape Manipulation". In: *ACM Trans. Graph.* 24.3 (2005).

[Iji+08] Takashi Ijiri, Radomir Mech, Takeo Igarashi, and Gavin Miller. "An Example-based Procedural System for Element Arrangement". In: *Computer Graphics Forum* (2008). ISSN: 1467-8659.

[Jac+11] Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. "Bounded Biharmonic Weights for Real-Time Deformation". In: *ACM Trans. Graph.* 30.4 (2011).

[JQ14] Pol Jeremias and Inigo Quilez. *ShaderToy.* https://www.shadertoy.com/. 2014.

[Jim23] Álvaro Barbero Jiménez. "Mixture of Diffusers for scene composition and high resolution image generation". In: *arXiv preprint arXiv:2302.02412* (2023).

[Kar+17] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. "Progressive growing of gans for improved quality, stability, and variation". In: *arXiv preprint arXiv:1710.10196* (2017).

[Kar+20] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. "Analyzing and improving the image quality of stylegan". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition.* 2020, pp. 8110–8119.

[Kee19] Matt Keeter. *libfive: Infrastructure for solid modeling.* https://libfive.com/. 2019.

[KB14] Diederik P. Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[KW13] Diederik P. Kingma and Max Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[Krs+20] Vojtech Krs, Radomir Mech, Mathieu Gaillard, Nathan Carr, and Bedrich Benes. "PICO: Procedural Iterative Constrained Optimizer for Geometric Modeling". In: (2020), pp. 1–1. DOI: 10.1109/TVCG.2020.2995556.

[Kwa+05] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. "Texture Optimization for Example-Based Synthesis". In: 24.3 (2005), pp. 795–802.

[Li+20] Tzu-Mao Li, Michal Lukáč, Michaël Gharbi, and Jonathan Ragan-Kelley. "Differentiable Vector Graphics Rasterization for Editing and Learning". In: *ACM Trans. Graph.* 39.6 (2020).

[Lip+19] Markus Lipp, Matthias Specht, Cheryl Lau, Peter Wonka, and Pascal Müller. "Local editing of procedural models". In: *Comp. Graph. Forum.* Vol. 38. 2. Wiley Online Library. 2019, pp. 13–25. DOI: 10.1111/cgf.13615.

[LMS03] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. "Iterated local search". In: *Handbook of metaheuristics.* Springer, 2003, pp. 320–353.

[Ma+13] Chongyang Ma, Li-Yi Wei, Sylvain Lefebvre, and Xin Tong. "Dynamic Element Textures". In: 32.4 (2013). ISSN: 0730-0301.

[Mag22] MagicaCSG. *MagicaCSG.* http://ephtracy.github.io/index.html?page=magicacsg. 2022.

[Mes18] Lars Mescheder. "On the convergence properties of gan training". In: *arXiv preprint arXiv:1801.04406* 1 (2018), p. 16.

[Met+16] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. "Unrolled generative adversarial networks". In: *arXiv preprint arXiv:1611.02163* (2016).

[MB21] Élie Michel and Tamy Boubekeur. "DAG amendment for inverse control of parametric shapes". In: *ACM Transactions on Graphics* 40 (Aug. 2021), pp. 1–14.

[Mül+07] Pascal Müllerl, Gang Zeng, Peter Wonka, and Luc Van Gool. "Image-Based Procedural Modeling of Facades". In: *ACM Trans. Graph.* 26.3 (2007), 85–es. ISSN: 0730-0301.

[NM65] J. A. Nelder and R. Mead. "A Simplex Method for Function Minimization". In: *The Computer Journal* 7.4 (1965), pp. 308–313.

[NBA18] Gen Nishida, Adrien Bousseau, and Daniel G. Aliaga. "Procedural Modeling of a Building from a Single Image". In: *Computer Graphics Forum* 37 (2018).

[PLL11] Jong Pil Park, Kang Hoon Lee, and Jehee Lee. "Finding Syntactic Structures from Human Motion Data". In: *Computer Graphics Forum* 30 (2011).

[Pas+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[Pel10] Fabio Pellacini. "envyLight: an interface for editing natural illumination". In: *ACM Transactions on Graphics* 29 (July 2010), p. 1.

[PTG02] Fabio Pellacini, Parag Tole, and Donald P. Greenberg. "A User Interface for Interactive Cinematic Shadow Design". In: *ACM Trans. Graph.* 21.3 (2002), pp. 563–566.

[Per85] Ken Perlin. "An image synthesizer". In: *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '85. New York, NY, USA: Association for Computing Machinery, 1985, pp. 287–296. ISBN: 0897911660. DOI: `10.1145/325334.325247`. URL: `https://doi.org/10.1145/325334.325247`.

[Pod+23] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. "Sdxl: Improving latent diffusion models for high-resolution image synthesis". In: *arXiv preprint arXiv:2307.01952* (2023).

[Qui] Inigo Quilez. *Inigo Quilez*. `https://iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm`.

[Rad+21] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. "Learning transferable visual models from natural language supervision". In: *International conference on machine learning*. PMLR. 2021, pp. 8748–8763.

[Red+20] Pradyumna Reddy, Paul Guerrero, Matt Fisher, Wilmot Li, and Niloy J. Mitra. "Discovering Pattern Structure Using Differentiable Compositing". In: *ACM Trans. Graph.* 39.6 (2020).

[RMD11] Tim Reiner, Gregor Mückl, and Carsten Dachsbacher. "Interactive Modeling of Implicit Surfaces Using a Direct Visualization Approach with Signed Distance Functions". In: *Computer & Graphics, Proceedings of Shape Modeling International* 35.3 (2011), pp. 596–603. ISSN: 0097-8493.

[Rit+15] Daniel Ritchie, Ben Mildenhall, Noah D Goodman, and Pat Hanrahan. "Controlling procedural modeling programs with stochastically-ordered sequential monte carlo". In: *ACM Trans. Graph.* 34.4 (2015), pp. 1–11. DOI: 10.1145/2766895.

[Rod+24] Carlos Rodriguez-Pardo, Dan Casas, Elena Garces, and Jorge Lopez-Moreno. "TexTile: A Differentiable Metric for Texture Tileability". In: *arXiv preprint arXiv:2403.12961* (2024).

[Rom+22] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. "High-resolution image synthesis with latent diffusion models". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 10684–10695.

[Rov+15] Riccardo Roveri, A. Cengiz Öztireli, Sebastian Martin, Barbara Solenthaler, and Markus Gross. "Example Based Repetitive Structure Synthesis". In: Eurographics Association, 2015, pp. 39–52.

[Rui+23] Nataniel Ruiz, Yuanzhen Li, Varun Jampani, Yael Pritch, Michael Rubinstein, and Kfir Aberman. "Dreambooth: Fine tuning text-to-image diffusion models for subject-driven generation". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 22500–22510.

[Sau+23] Axel Sauer, Dominik Lorenz, Andreas Blattmann, and Robin Rombach. "Adversarial diffusion distillation". In: *arXiv preprint arXiv:2311.17042* (2023).

[Sch+06] Ryan Schmidt, Brian Wyvill, Mario Costa Sousa, and Joaquim. Armando Jorge. "ShapeShop: Sketch-Based Solid Modeling with BlobTrees". In: SIGGRAPH '06. 2006.

[Sch+14] Thorsten-Walther Schmidt, Fabio Pellacini, Derek Nowrouzezahrai, Wojciech Jarosz, and Carsten Dachsbacher. "State of the Art in Artistic Editing of Appearance, Lighting, and Material". In: *Computer Graphics Forum* 35 (Jan. 2014).

[Sch+21] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. "Laion-400m: Open dataset of clip-filtered 400 million image-text pairs". In: *arXiv preprint arXiv:2111.02114* (2021).

[Sha+18] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. "CSGnet: Neural shape parser for constructive solid geometry". In: *Proceedings of the IEEE Conference on Computer Vision*

and Pattern Recognition. 2018, pp. 5515–5523. DOI: 10.48550/arXiv.1712.08290.

[Shi+20] Liang Shi, Beichen Li, Miloš Hašan, Kalyan Sunkavalli, Tamy Boubekeur, Radomir Mech, and Wojciech Matusik. "MATch: Differentiable Material Graphs for Procedural Material Capture". In: *ACM Trans. Graph.* 39.6 (2020), pp. 1–15.

[Smi+20] Dmitriy Smirnov, Matthew Fisher, Vladimir G. Kim, Richard Zhang, and Justin Solomon. "Deep Parametric Shape Predictions using Distance Fields". In: *CVPR*. 2020.

[Smi+23] David Smith, Joseph Samuel Myers, Craig S. Kaplan, and Chaim Goodman-Strauss. *An aperiodic monotile*. 2023. arXiv: 2303.10798 [math.CO].

[Soh+15] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. "Deep unsupervised learning using nonequilibrium thermodynamics". In: *International Conference on Machine Learning*. PMLR. 2015, pp. 2256–2265.

[SME20] Jiaming Song, Chenlin Meng, and Stefano Ermon. "Denoising diffusion implicit models". In: *arXiv preprint arXiv:2010.02502* (2020).

[Son+23] Yang Song, Prafulla Dhariwal, Mark Chen, and Ilya Sutskever. "Consistency models". In: *arXiv preprint arXiv:2303.01469* (2023).

[Šta+10] Ondrej Št'ava, Bedrich Benes, Radomir Měch, Daniel G Aliaga, and Peter Krištof. "Inverse procedural modeling by automatic generation of L-systems". In: *Comp. Graph. Forum*. Vol. 29. 2. Wiley Online Library. 2010, pp. 665–674. DOI: 10.1111/j.1467-8659.2009.01636.x.

[Šta+14] Ondrej Št'ava, Sören Pirk, Julian Kratt, Baoquan Chen, Radomír Měch, Oliver Deussen, and Bedrich Benes. "Inverse procedural modelling of trees". In: *Comp. Graph. Forum*. Vol. 33. 6. Wiley Online Library. 2014, pp. 118–131. DOI: 10.1111/cgf.12282.

[Sug+08] Masamichi Sugihara, Erwin de Groot, Brian Wyvill, and Ryan Schmidt. "A Sketch-Based Method to Control Deformation in a Skeletal Implicit Surface Modeler". In: *Eurographics Workshop on Sketch-Based Interfaces and Modeling*. 2008.

[SWS10] Masamichi Sugihara, Brian Wyvill, and Ryan Schmidt. "WarpCurves: A tool for explicit manipulation of implicit surfaces". In: *Computers & Graphics* 34.3 (2010), pp. 282–291.

[Tru+19] E. Trunz, S. Merzbach, J. Klein, T. Schulze, M. Weinmann, and R. Klein. "Inverse Procedural Modeling of Knitwear". In: *CVPR* (2019), pp. 8622–8631.

[Tu+20] Peihan Tu, Li-Yi Wei, Koji Yatani, Takeo Igarashi, and Matthias Zwicker. "Continuous Curve Textures". In: *ACM Trans. Graph.* 39.6 (Nov. 2020).

[Van+12] Carlos A. Vanegas, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Paul Waddell. "Inverse design of urban procedural models". In: *ACM Trans. Graph.* 31.6 (2012), 168:1–168:11.

[Vas+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[Vec+23] Giuseppe Vecchio, Rosalie Martin, Arthur Roullier, Adrien Kaiser, Romain Rouffet, Valentin Deschaintre, and Tamy Boubekeur. "ControlMat: Controlled Generative Approach to Material Capture". In: *arXiv preprint arXiv:2309.01700* (2023).

[Vec+24] Giuseppe Vecchio, Renato Sortino, Simone Palazzo, and Concetto Spampinato. "MatFuse: Controllable Material Generation with Diffusion Models". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024.

[Vir+20] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, I. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272.

[WCL23] Jianyi Wang, Kelvin CK Chan, and Chen Change Loy. "Exploring clip for assessing the look and feel of images". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 2. 2023, pp. 2555–2563.

[Wei+09] Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. "State of the Art in Example-based Texture Synthesis". In: *Eurographics 2009 - State of the Art Reports*. The Eurographics Association, 2009.

[Wom22] Womp3D. *Womp 3D Inc.* https://womp.com/. 2022.

[Wu+13] Fuzhang Wu, Dong-Ming Yan, Weiming Dong, Xiaopeng Zhang, and Peter Wonka. "Inverse procedural modeling of facade layouts". In: *ACM Transactions on Graphics (TOG)* 33 (2013), pp. 1–10.

[WGG99] Brian Wyvill, Andrew Guy, and Éric Galin. "Extending the CSG Tree - Warping, Blending and Boolean Operations in an Implicit Surface Modeling System". In: *Computer Graphics Forum* 18.2 (1999), pp. 149–158.

[Ye+23] Hu Ye, Jun Zhang, Sibo Liu, Xiao Han, and Wei Yang. "Ip-adapter: Text compatible image prompt adapter for text-to-image diffusion models". In: *arXiv preprint arXiv:2308.06721* (2023).

[ZCG15] Cédric Zanni, Marie Paule Cani, and Michel Gleicher. "N-Ary Implicit Blends with Topology Control". In: *Comput. Graph.* 46 (2015), pp. 1–13.

[ZRA23] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. "Adding conditional control to text-to-image diffusion models". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision.* 2023, pp. 3836–3847.

[Zho+22] Xilong Zhou, Milos Hasan, Valentin Deschaintre, Paul Guerrero, Kalyan Sunkavalli, and Nima Khademi Kalantari. "TileGen: Tileable, Controllable Material Generation and Capture". In: *SIGGRAPH Asia Conference Papers.* 2022, 34:1–34:9.

[Zho+23] Yang Zhou, Kaijian Chen, Rongjun Xiao, and Hui Huang. "Neural Texture Synthesis with Guided Correspondence". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2023, pp. 18095–18104.

[Zho+24] Yang Zhou, Rongjun Xiao, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. "Generating Non-Stationary Textures using Self-Rectification". In: *arXiv preprint arXiv:2401.02847* (2024).

[Zho+18] Yang Zhou, Zhen Zhu, Xiang Bai, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. "Non-stationary Texture Synthesis by Adversarial Expansion". In: *ACM Trans. Graph. (Proc. SIGGRAPH)* 37.4 (2018), 49:1–49:13.

[Zhu+15] Huilong Zhuo, Shengchuan Zhou, Bedrich Benes, and David Whittinghill. "User-Assisted Inverse Procedural Facade Modeling and Compressed Image Rendering". In: *Advances in Visual Computing.* 2015, pp. 126–136. ISBN: 978-3-319-27863-6.