Pattern Analogies
Learning to Perform Programmatic Image Edits by Analogy
Supplementary

Anonymous CVPR submission

Paper ID 1268

1. Introduction

In this document, we present additional details regarding our system. First, we provide a brief overview of the videos included in the supplemental material. Next, in section 3, we provide details of the proposed Domain Specific Language (DSL), SPLITWEAVE, including the design of the two pattern-style specific program samplers. Section 5 provides additional details regarding *Analogical Quartet Sampling*, detailing the *programmatic* pattern edits employed. This is followed by details of our test dataset and the three applications enabled by our approach in Section 6. Finally, Sections 7, 8, 9 presents additional experiments and results, including qualitative examples and failure cases. The code for our system — the DSL, program samplers, and model training — will be open sourced if and when the paper is acceptance.

2. Video Results

We provide the following videos in the supplemental materials:

1. A video titled `editing.mp4` which demonstrates the use of SPLITWEAVE for editing real-world patterns with simple pattern analogies.
2. A video titled `pattern_animation.mp4` which presents our results for pattern animation transfer. Please refer to section 6.2 for more details on transferring pattern animations.

3. A language for visual patterns

In the main paper, we introduced SPLITWEAVE, a DSL designed for creating visual patterns. As described previously, we use SPLITWEAVE to (a) generate a large dataset of high-quality synthetic patterns for training an analogical editor and (b) to define parametric analogy pairs (A, A') at test-time to guide transformation in target pattern B. Further, we constructed two custom SPLITWEAVE program samplers which aid the sampling of high-quality synthetic patterns in two domains, namely *Motif Tiling Patterns* (MTP), and *Split Filling Patterns* (SFP).

SPLITWEAVE is designed specifically for generating patterns that exhibit structured partitioning of a 2D canvas. Programs in SPLITWEAVE define a process to map each spatial location on the canvas to an RGBA value, resulting in a visual pattern. This process is achieved through two core mappings: (1) spatial locations (x, y) are first mapped to 2D UV coordinates and (2) UV coordinates are then mapped to outputs such as RGBA values or other signals. SPLITWEAVE provides operators to abstract and simplify these mappings.

3.1. `UVExpr` and `SExpr`

`UVExpr` and `SExpr` are the two key types of expressions used in SPLITWEAVE programs to define these mappings:

`UVExpr` A `UVExpr` defines a function

$$\texttt{UVExpr} : \mathbb{R}^2 \rightarrow \mathbb{R}^2,$$

which maps each spatial location (x, y) on the canvas to a corresponding UV coordinate (u, v). This provides a spatial framework for pattern generation, enabling operations such as distortions, tiling, or structured partitioning (e.g., `BrickSplit`, `HexagonalSplit`). Evaluating a `UVExpr` generates a UV grid which serves as the basis for further evaluating `SExprs`.

`SExpr` A `SExpr` defines a function

$$\texttt{SExpr} : \mathbb{R}^2 \rightarrow \mathbb{R}^N,$$

which maps each UV coordinate $(u, v) \in \mathbb{R}^2$ to an N-dimensional output. The value of N depends on the type of output being generated. `SExprs` that evaluate to 4-channel outputs ($N = 4$) are typically used to generate RGBA canvases. Alternatively, `SExprs` which evaluate to single-channel outputs($N = 1$) are used to generate single-channel

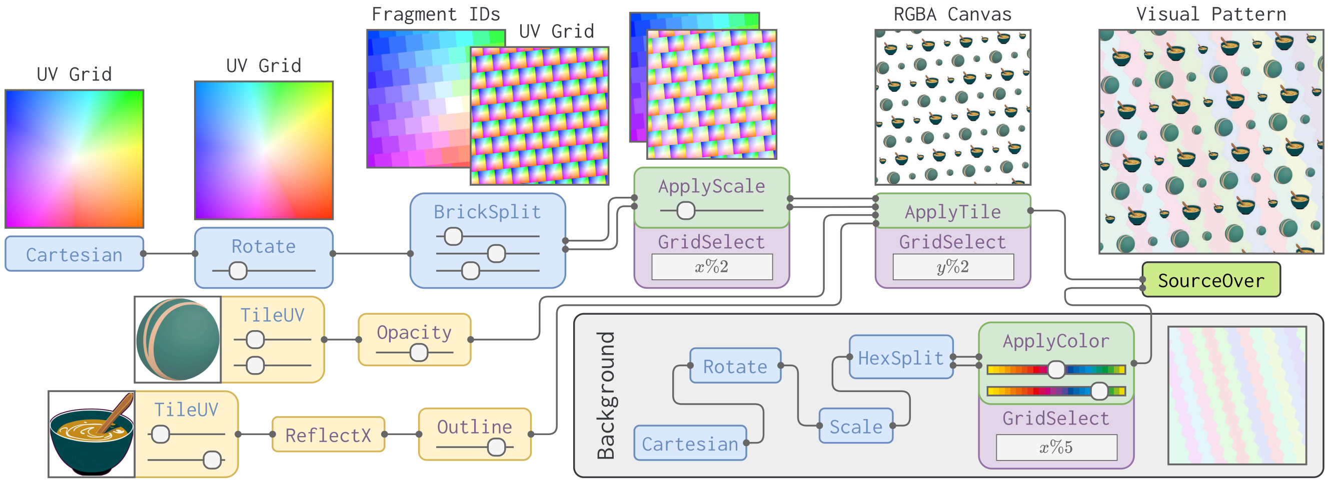

Figure 1. **Program evaluation** We illustrate the evaluation of a SPLITWEAVE program. SPLITWEAVE is used to create directed acyclic graphs representing data flow between different operators. The *UV Grid Operators* are used to define `UVExprs`, which map spatial coordinate to UV grids. *Signal Operators* are used to define `SExprs` which map UV-Grids to single or multi-channel spatial maps (such as RGBA canvases). *Spatially Varying Operators* take inputs such as UV-Grids and Fragment Ids to apply spatially varying transforms. Finally, *Utility Operators* perform tasks such as composing multiple RGBA canvases together.

buffers used to represent spatial masks, distortion fields, or other intermediate signals.

`UVExprs` are primarily used to generate structured partitions of the canvas through partitioning operators (e.g., *BrickSplit*). Evaluating these operators produce not only a corresponding UV grid, but also a **fragment ID buffer**, where each spatial location is assigned a fragment identifier corresponding to its partition. As operators are composed, the fragment ID buffers are updated and stacked, enabling hierarchical partitioning and fragment-aware transformations. This mechanism is critical for supporting *Spatially Varying Transformations*, used in *Motif Tiling Patterns* (MTP), where operations vary based on partitioning, and *Fragment Grouping*, essential for *Split-Filling Patterns* (SFP), where fragments are grouped together for applying color fills.

`SExprs` typically contain analytical functions defining SVG objects, such as 2D circles, Bezier curves etc, and Texuture-Mapping operators, which map UV coordinates to samples on pre-defined 2D maps. Texture mapping operators are primarily used for mapping RGBA tiles on UV grids. Evaluating `SExpr` on different UV-grids results in different outputs. These outputs are used to generate RGBA canvases or auxiliary data buffers for generating the visual pattern image.

3.2. Operator Categories

SPLITWEAVE provides four broad categories of operators to support the construction of `UVExprs`, `SExprs`, and their transformations:

1. $\sim$ *50 UV Grid Operators:* Used to define `UVExprs`.
2. $\sim$ *70 Signal Operators:* Used to define `SExprs`.
3. *10 Spatially Varying Operators:* Used to define transformations in a partition-aware manner using fragment IDs (e.g., resizing alternate rows or applying per-fragment coloring).
4. *Utility Operators:* Used for remaining purposes such as combining multiple canvases (*SourceOver*) or generating auxiliary spatial signals used in fragment-aware operations.

In Figure 1, we illustrate the evaluation of a SPLITWEAVE program used to create a MTP pattern. This program uses all the four different types of operators, each associated with a separate color. To create the pattern, we separately create a background canvas and a foreground canvas. To create the foreground canvas, we first convert the pixel-space canvas to a UV cartesian grid ($\in [-1, 1]^2$) using `Cartesian`. This grid is subsequently rotated using the `Rotate` operator. Next, by using the `BrickSplit` operator, we create two outputs, a transformed UV-grid, which now consists of brick-style spatial partitions, and a 2D fragment-ID buffer containing integers that corresponds to fragment IDs. Using the fragment-ID buffer, we apply spatially-varying scaling to decrease the size of tiles in alternate columns. This is followed by a `ApplyTile` operator to create the foreground canvas. Internally, `ApplyTile` evaluates the `SExprs` corresponding to each tile on the transformed UV-grid, and merges alternate rows of the resulting two RGBA canvases using the fragment-ids from `BrickSplit`. A similar process is followed for the background to obtain the background canvas.

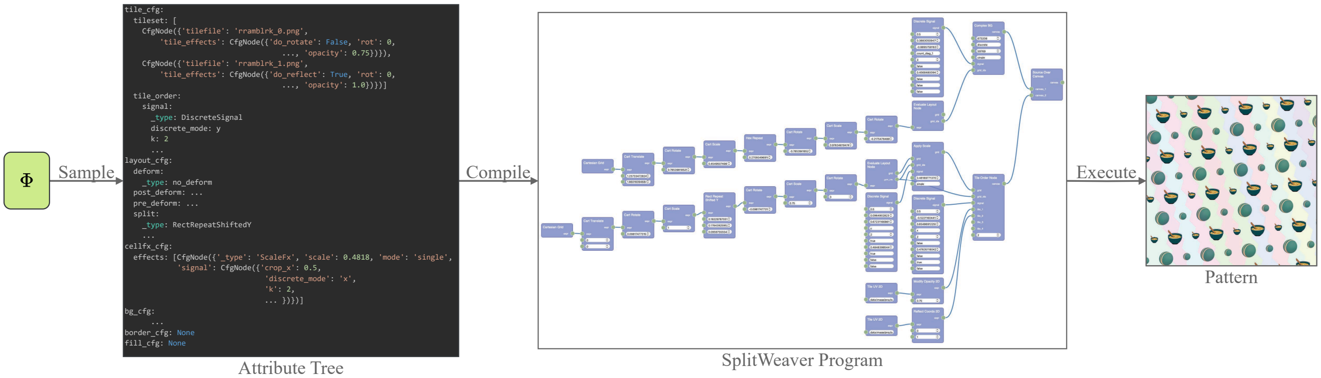

Figure 2. Our Custom program samplers Φ generates attribute trees AT, a hierarchical data structure that encodes patterns structure specification. The attribute trees are then compiled into SPLITWEAVE programs. Finally, we generate visual patterns by evaluating SPLITWEAVE program. The use of Φ and AT help generate high-quality synthetic patterns.

Finally, we combine the background and foreground with the `SourceOver` operator to obtain the final MTP pattern.

3.3. Implementation

SPLITWEAVE is implemented in Python, making it accessible to a wide range of users, including those with limited programming experience. This lowers the learning curve for novice users and facilitates integration with emerging tools, such as large language models (LLMs), for programmatic generation and manipulation of visual patterns. The core operators in SPLITWEAVE are implemented using PyTorch [9], which allows many of the operators to be automatically differentiable. This opens up exciting possibilities for future work in using automatic differentiation for visual program inference, enabling the recovery of programmatic structures directly from visual patterns.

We have also developed a front-end application using *Rete.js* [3] to support visual programming with SPLITWEAVE. This tool simplifies the creation and manipulation of SPLITWEAVE programs by providing an intuitive, node-based interface. Manipulation of SPLITWEAVE programs using this interface is demonstrated in the supplemental videos. Currently implemented as a proof of concept, it is primarily intended for inspecting SPLITWEAVE programs and performing parametric analogical edits on real-world patterns. Future work will focus on refining the application to make it more user-friendly and suitable for broader usage. We hope that SPLITWEAVE serves as a stepping stone for further research in visual pattern generation and manipulation, inspiring new methodologies and applications in this domain.

4. Custom Program Samplers

As discussed in the main paper, random sampling of the SPLITWEAVE grammar often produces poor-quality patterns that are incoherent or irrelevant for training. To address this limitation, we construct custom program samplers designed to generate high-quality SPLITWEAVE programs through a structured, hierarchical process.

The custom program samplers work by generating an *attribute tree*, a hierarchical data structure that encodes the specification for a pattern. This attribute tree is then compiled into a valid SPLITWEAVE program, which, when executed, produces the final visual pattern. The pipeline can be formalized as:

$$\Phi \xrightarrow{\text{Sample}} AT \xrightarrow{\text{Compile}} P_{SW} \xrightarrow{\text{Execute}} \text{Pattern},$$

where Φ is a high-level process specification that defines the abstract structure of the pattern, AT is the attribute tree that instantiates this structure with specific parameters, and the resulting SPLITWEAVE program, represented as P_{SW}, defines the procedural steps to produce the pattern. Figure 2 illustrates this workflow with an example, showing the attribute tree, its compilation into a SPLITWEAVE program, and the resulting visual pattern.

The attribute tree AT is constructed by first designing an abstract process specification Φ that represents the steps involved in creating a pattern. For example, in Motif Tiling Patterns (MTP), Φ includes stages such as sampling tiles, sampling layout parameters, and sampling effects like background elements. Each stage in Φ corresponds to a node or sub-tree in AT, where the nodes represent specific components, and the edges encode relationships or contextual parameters. To populate AT, we implement domain-specific random samplers for each node in the tree. These samplers generate valid and diverse configurations for their respective components. At the top level, a hierarchical sampler integrates these components to form a complete attribute tree. For instance, the MTP sampler samples specification for canvas partitioning, tiles and their transformations and spatially varying effects, combining them into a unified representation.

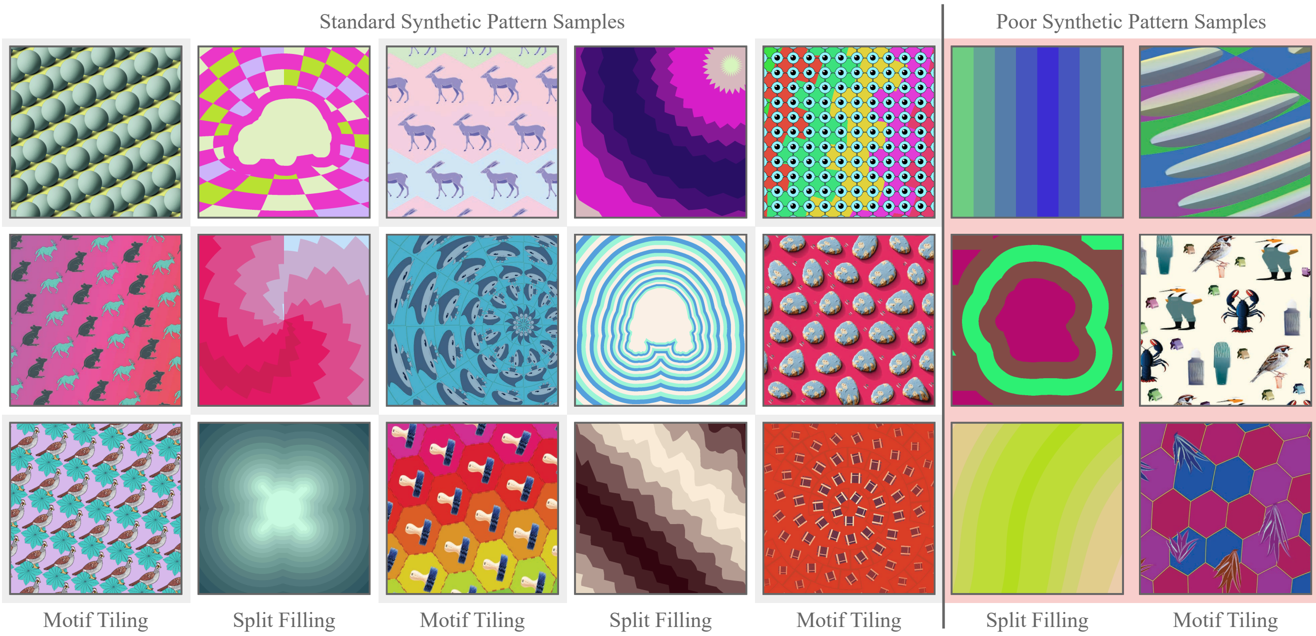

Figure 3. We present synthetic samples generated by our custom program samplers for two pattern styles, namely, Motif Tiling Patterns (MTP) and Split Filling Pattern (SFP). The custom program sampler can still produce poor quality patterns as depicted in the rightmost two columns.

The hierarchical nature of the attribute tree allows modular control over each component, enabling flexibility and extensibility. By sampling each node independently, the custom samplers ensure that the resulting patterns are both diverse and semantically meaningful, addressing the challenges of random grammar sampling. Once the attribute tree AT is constructed, it is compiled into a SPLITWEAVE program. This compilation step translates the hierarchical structure and parameters encoded in AT into valid SPLITWEAVE code, adhering to the syntax and semantics of the DSL. Executing the compiled SPLITWEAVE program produces the final visual pattern. This structured workflow provides a controlled and flexible framework for generating patterns. The combination of a process-driven attribute tree design and creation of pattern style-specific samplers ensures the generation of high-quality visual patterns.

In figure 3, we present synthetic samples of both MTP and SFP styles generated by this process. We also show failure cases in the two right-most columns. The custom sampler for MTP patterns sometimes generates samples with a high amount of stretching, too much visual complexity, or sparse tiling. Similarly, SFP pattern sampler can fail due to trivial grid partitioning, over-zooming, or poor random color section.

To create the MTP patterns we also generate a large dataset of $100,000$ RGBA tiles. Earlier experiments with fewer tiles showed that having a diverse and large set of tiles is essential to generalize to 'in-the-wild' real-world patterns. To create tiles on a large variety of subjects, we first extract a subset of nouns from wordnet-synset [8]. First, we prune the nouns by type (avoiding types such as 'event', 'process'), followed by rejection based on keyword match (to avoid different forms of 'bacteria', 'virus' etc.). Finally, we use SigLIP [14] text-encoding of prompts in the form of `` `A photo of a/an $item' `` to cluster the

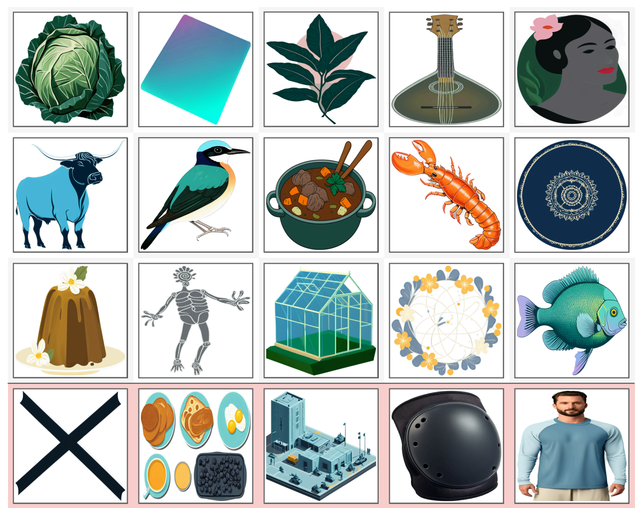

Figure 4. We generate tiles for MTP patterns using LayerDiffuse [15]. We present both good quality tiles (top 3 rows) and poor quality tiles (bottom row).

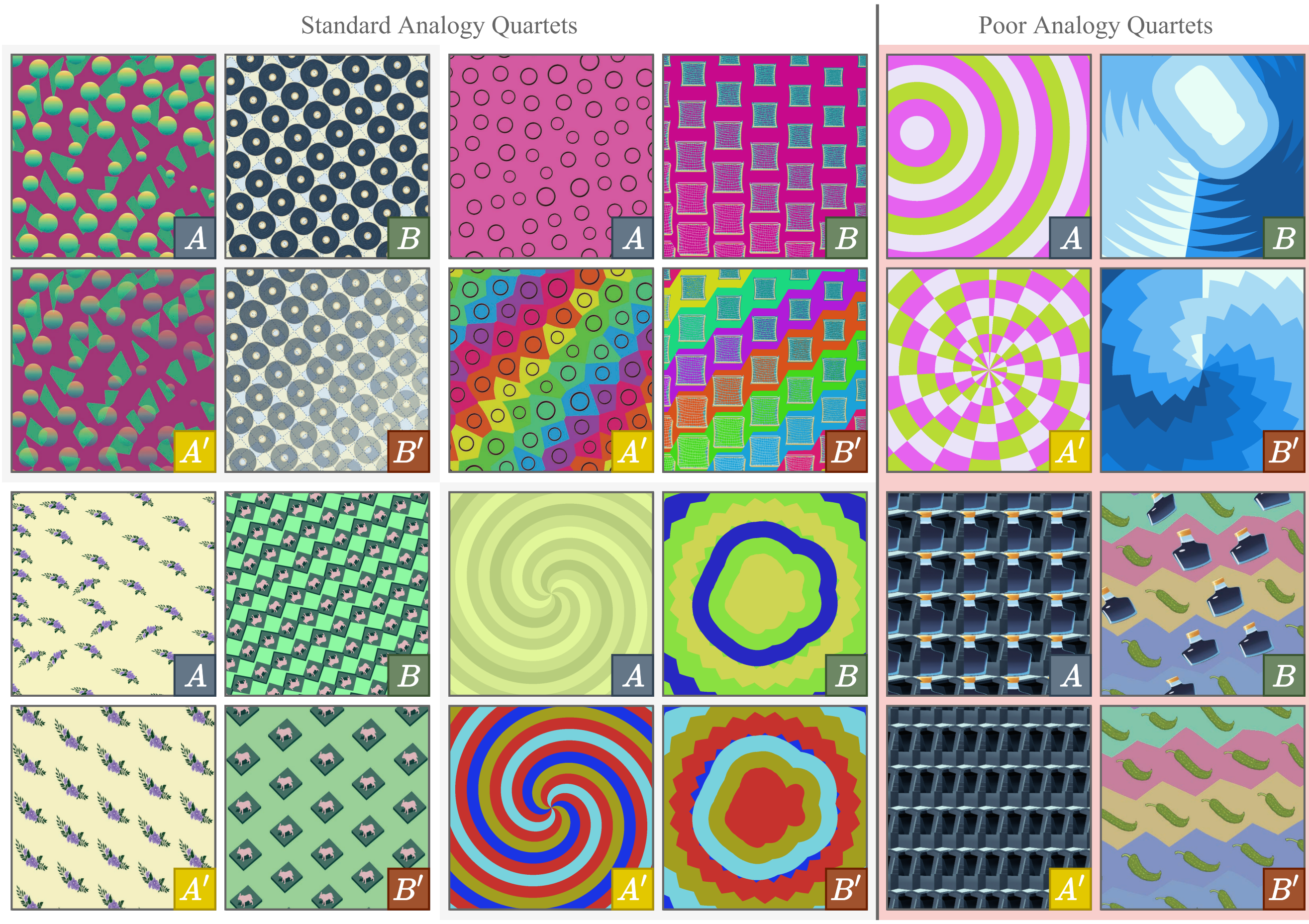

Figure 5. We present analogical quartets created using our approach. While many analogical quarters are of good quality, our synthetic sampling process can also result in poor quality quartets, as shown in the right-most column.

nouns and extract $\sim 10,000$ distinct nouns. These nouns are then used to create text-prompts using a template of the form ``` ``A minimal $style $second_term of a $noun $minimalism on a $color_scheme background.'' ``` where the variables such as `$style` and `$second_term` are filled with random samples from list of keywords. Then, we generate RGBA images for each prompt using LayerDiffuse [15], which generates images with alpha maps. Finally, tiles are created by extracting a tight bounding box subset of the generated image, with simple thresholds to reject samples in case of too high and too low complexity (measured using JPEG [13] compression). Figure 4 presents a few samples of tiles generated by this process. We note a few recurring failure cases: a) Extremely simple tiles, b) tiles with multiple objects, c) Tiles with poor cropping, and d) realistic rendering effects on tiles. Despite these flaws, a majority of the tiles seem to be useful, particularly to help to model avoid overfitting to training tiles.

5. Sampling Analogical Quartet

In the main paper, we introduced analogical quartets (A, A', B, B') that are used to our train analogical editing model. These quartets are grounded in Structure Mapping Theory [6], which defines analogies as mappings of relational structure from a base to a target domain. The relationship R between program pairs $(z_A, z_{A'})$ and $(z_B, z_{B'})$ remains consistent:

$$R(z_A, z_{A'}) = R(z_B, z_{B'}). \tag{1}$$

Here, we provide additional details on how edits are defined, sampled, and applied to construct these quartets, along with examples and a discussion of failure cases.

Edits in our framework operate directly on the attribute tree AT, rather than on SplitWeave programs. This approach ensures semantic validity and allows for efficient resampling of components. Each edit targets a node just below the root of the tree, corresponding to high-level components in the pattern creation process. For Motif Tiling

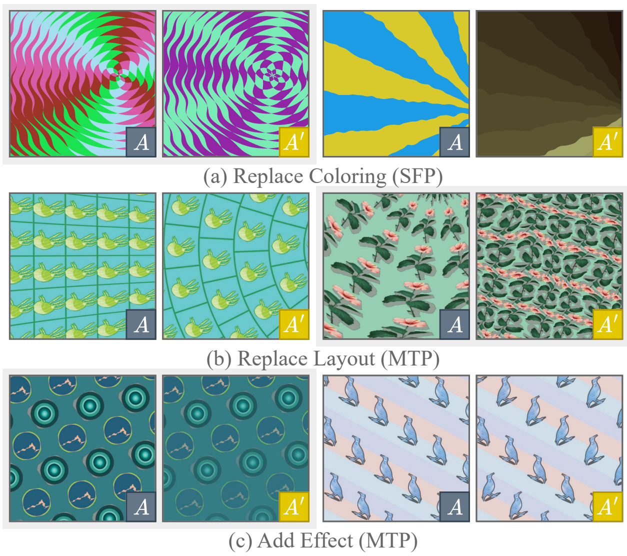

Figure 6. We present examples of editing synthetic patterns A with different edits to generate edited pattern A'.

Figure 7. Our model enables users to mix aspects of different patterns to create novel patterns. In this example, The layout of X is mixed with the tiles of Y to generate the pattern Y'.

Patterns (MTP), the editable components include:

1. *Tiles*: Add, remove, or replace tiles in the pattern.
2. *Layout*: Replace the layout structure.
3. *Cell Effects*: Add or remove specific spatially varying effects applied to cells.
4. *Background and Border*: Replace background or border styles.

For Split-Filling Patterns (SFP), the editable components include:

1. *Foreground Layout and Background Layout*: Replace the layout for either foreground, background or both elements.
2. *Fill Specifications*: Replace the specifications for filling regions.

Edits are applied by resampling or modifying nodes in the attribute tree. To perform a *replace* edit, the target node is resampled to produce a new specification, such as a new layout or tile configuration, and this new specification is used to create both A' and B'. To perform a *add* edit, a new node is created and inserted into the appropriate list (e.g., adding a tile or effect). Finally, to perform a *remove* edit, a node is added, and the order of the quartet is flipped (e.g., swapping $A \leftrightarrow A'$ and $B \leftrightarrow B'$). Applying random edits to randomly sampled pattern sets (A, B) can generate invalid new pattern (A', B'). Therefore, we instead first sample an edit e and, perform rejection sampling of (A, B) pairs to generate valid analogical quartets.

In Figure 6, we present some examples of pattern pairs generated by editing a pattern A to create A', of both MTP and SFP styles. Figure 5 provides examples of generated analogical quartets, demonstrating consistent transformations between (A, A') and (B, B'). Despite its robustness, our approach can encounter failure cases. For instance, despite the pattern programs satisfying equation 1, visually salient relation between (A, A') and (B, B') may not be analogical. Furthermore, sometimes (A, A') pair may not clearly demonstrate the desired change.

6. Additional Details

We now provide additional details regarding our test set consisting of real-world patterns, and

6.1. Test Set Creation

To evaluate our method, we created a test set by collecting 116 patterns from Adobe Stock. Based on visual inspection, we annotated desirable edits for 50 patterns in text. For each annotated edit, we manually constructed input analogies using SPLITWEAVE. These analogies were not always designed to be simple, as we aimed to test the model's ability to interpret non-trivial analogies effectively. The test set, along with annotated edits, is included in the supplementary material.

6.2. Application: Pattern Mixing

The goal of pattern mixing is to transfer aspects of one real-world pattern X to another real-world pattern Y. This approach makes it easier to create novel variations of patterns and to transfer specific aspects of patterns that may not be present in our synthetic dataset. To achieve this, we construct an analogy pair (X, X'), which is used as input to edit Y. This sequential process, referred to as "chaining," allows edits to build upon the outputs of previous steps.

Our model's ability to use real-world patterns as analogy inputs enables chaining, which is critical for pattern mixing. This capability is attributed to the scale and diversity of the synthetic dataset used during training. Figure 7 illustrates this process for a pair (X, Y) where we mix the layout of X with the tiles of Y.

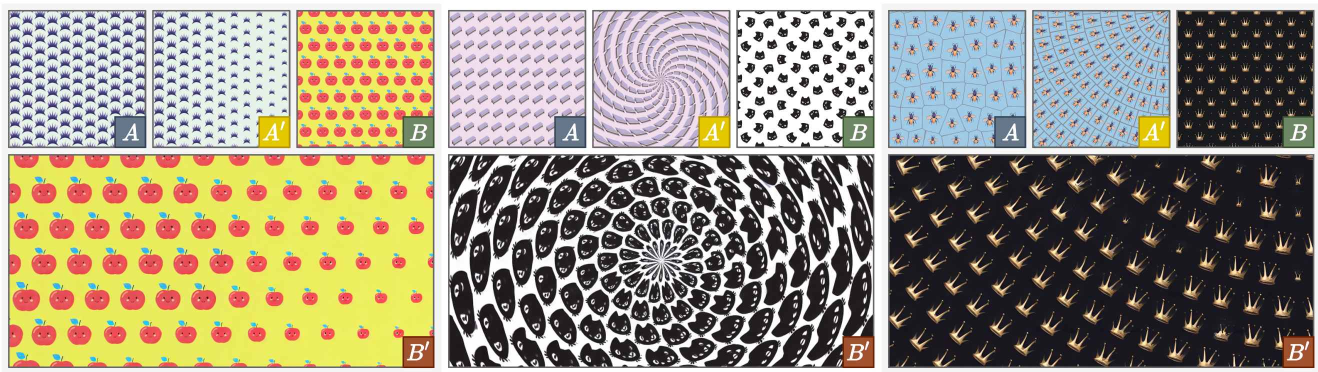

Figure 8. Our model can also be used to create wide canvases of non-stationary patterns by adapting MultiDiffusion [2] for spatially-conditioned generation. In these examples, we generate patterns of size 1536×1536 pixels and show a vertically centered crop.

6.3. Application: Pattern Animation

This application allows users to transfer an animations created using simple synthetic pattern A to real-world patterns B. Traditionally, such transfers require inferring the program for B and applying the animation to it. In contrast, with our method, users can automatically create analogy pairs from A's animation sequence to generate corresponding variations in B. The user provides as input $(A, \mathbf{A}')$, where $\mathbf{A}'$ represents the frames of the animation, and a real-world pattern B. We then employ TRIFUSER to generate variations of B that correspond to analogy pairs created for each frame as follows:

$$\mathbf{B}' = \{B' = M(A, A', B) | A' \in \mathbf{A}'\}. \tag{2}$$

To ensure temporal consistency, for each frame, we fix the initial latent noise, generate $n = 5$ samples and select the one with the lowest PSNR relative to the preceding frame. This approach avoids program inference and enables automated animation transfer. A demonstration video is provided in the supplementary material. In future, we hope to enforce stronger priors to improve temporal consistency.

6.4. Application: Wide Non-stationary Canvas

Visual patterns often need to adapt to varying resolutions, such as for use in presentations or posters. This is commonly achieved for stationary patterns by making the pattern image seamlessly tile-able. In fact, images generated using convolution-based diffusion models can also be made seamlessly tile-able by employing circular padding in the convolution layers. However, no such solution exists for non-stationary patterns. We provide a novel solution by adapting Multi-Diffusion [2] to our settings.

Multi-Diffusion solves the task of generating large images with diffusion models. This is achieved by first generating model predictions on tiled crops of the canvas and using the average predicted noise across overlapping image crops at each denoising step. Applying this naively to our method fails as our model strongly depend on the conditioning input (A, A^*, B) for generating B', i.e, they have strong dependence on the spatial orientation of the conditioning embeddings. To circumvent this issue, we adapt multi-diffusion for our model by performing consistent cropping across analogy inputs (A, A^*, B) and the latent code of B^* during generation. This adaptation enables the generation of wide, non-stationary canvases. Figure 8 illustrates three examples generated using this method, where we generated wide canvases which are 1536×1536 pixels in size.

	Analogy data	DSIM (↓)	DISTS (↓)	LPIPS (↓)	SSIM (↑)
LatentMod CATEGORICAL	✗	**0.242**	**0.320**	0.613	0.502
LatentMod TOKENWISE	✗	0.307	0.333	**0.581**	0.500
LatentMod ANALOGICAL	✓	0.273	0.330	0.620	**0.525**

Table 1. We compare different variations of *LatentMod* baselines. We observe that none of the variations are suitable for performing precise *programmatic* edits, indicating the unsuitability of latent-arithmetic based analogical editing for precise structure editing.

7. Quantitative Results

We now describe additional experiments conducted to further validate our system. First we discuss quantitative evaluations in this section, followed by qualitative results in section 8.

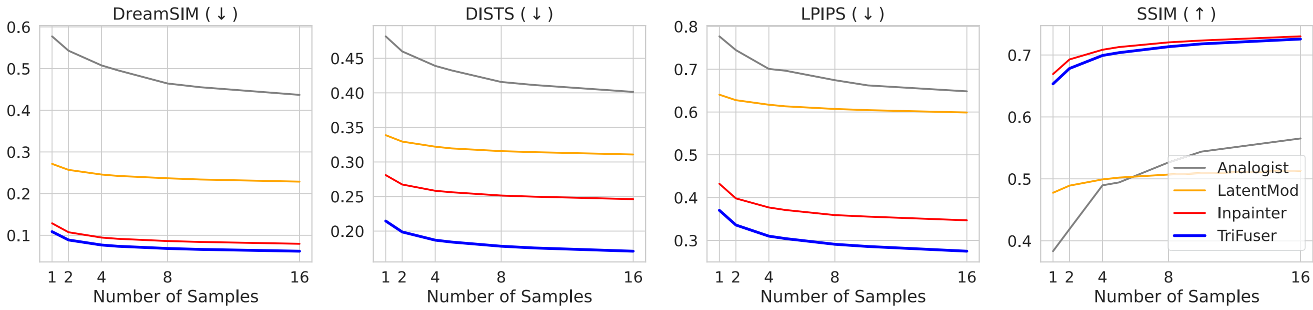

Figure 9. We compare our method, TRIFUSER, against the three baselines with four different metrics. The x-axis of each plot corresponds to the number of samples used for evaluation, demonstration that TRIFUSER remains superior to the baselines across sample count.

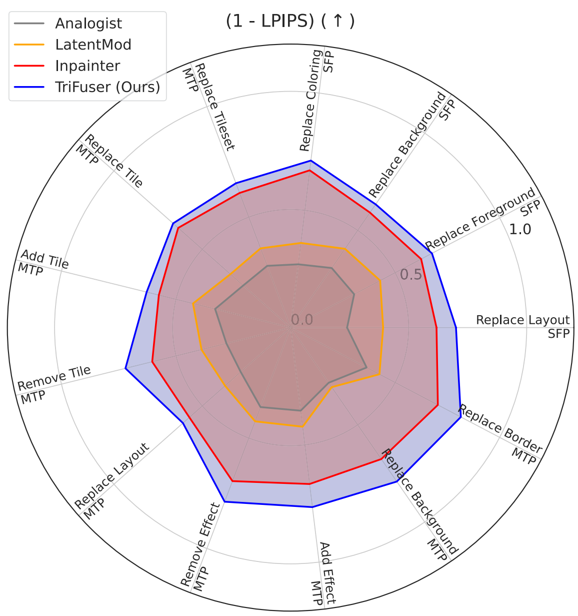

Figure 10. We compare our model against the baselines on a per-edit type basis. We observe that our model obtains higher perceptual similarity to the ground truth target across the edit types.

7.1. *LatentMod* Ablation

An important baseline we considered is *LatentMod*, where first we train a model to learn a latent space for representing patterns, followed by deploying *latent-arithmetic* [12] to create analogical patterns. Specifically, we first train a Image Variation Latent Diffusion Model (LDM) on our pattern dataset (i.e. condition on tokens extracted from a pattern image to denoise the same pattern image). Then, during test-time, given patterns (A, A', B) we infer the analogically edited pattern B' by using the LDM to denoise a Gaussian-initialized latent conditioned on the latent arithmetic tokens $(E(B) + E(A') - E(A))$. In this section we explore different variations of this model, demonstrating the superiority of the baseline used in the main paper over its alternatives.

First, we consider two architectures for the Image variation model. The first, referred to as CATEGORICAL, uses only a single pooled token (i.e. a 1×768 size embedding) as the conditioning input $E(A)$. The second, referred to as TOKENWISE, uses all the 257 image tokens generated by the token extracted (i.e. a 257×768 size embedding) as the conditioning input $E(A)$. Finally, we also consider an alternative of CATEGORICAL, as introduced in DeepVisualAnalogy [10]. This variation, referred to as ANALOGICAL, has the same architecture as CATEGORICAL, but has an alternate loss formulation which resembles the test-time usage. Essentially, this model is trained to denoise B' while being conditioned on $E(B)+E(A')-E(A)$ explicitly. Note that CATEGORICAL and TOKENWISE only require a dataset of training patterns, whereas ANALOGICAL requires analogical quartets (A, A', B, B') during training as well (similar to conditional generative approaches like ImageBrush [11] and our approach, TRIFUSER).

Table 1 compares these approaches on our synthetic validation set, reporting perceptual metrics—DSim [5], DIST [4] and LPIPS [16]—along with SSIM to capture pixel-level structural similarity. We first note that TOKENWISE shows worse results than CATEGORICAL. Since TOKENWISE is conditioned on a large embedding of size 257×768, the latent embedding fails to aid analogical reasoning (i.e. compression is essential for learning a latent space capable of analogical latent arithmetic). Secondly, we notice a surprising result that ANALOGICAL, despite being trained explicitly trained for analogical editing, is infact slightly weaker than CATEGORICAL. Visual inspection reveals that although ANALOGICAL and CATEGORICAL generate similar results, CATEGORICAL often tends to retain more aspects of the input pattern B compared to ANALOGICAL, which consequently sometimes results in a higher perceptual similarity to the target B'.

Finally, we remark that all these variations remain significantly weaker than the conditional analogical editors. This indicates that Latent Arithmetic is perhaps not suit-

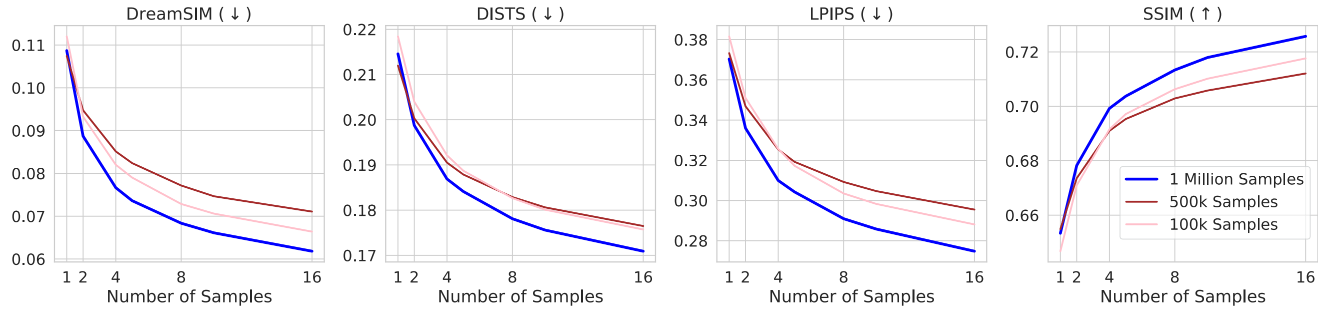

Figure 11. Training TRIFUSER with more analogical quartet samples improves its performance.

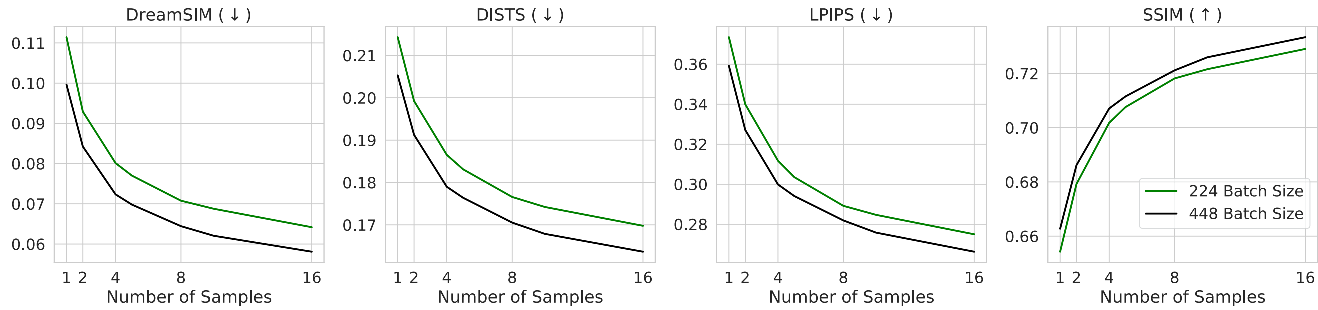

Figure 12. Training TRIFUSER with a larger batch size improves its performance.

able for *precise* editing as there is a inherent tussle between (a) representing sufficient details of patterns in the latent space to recreate them with high fidelity and (b) having sufficient compression of the latent space to achieve analogical reasoning via latent arithmetic. Consequently, most image-editing methods in the diffusion-era have turned towards alternate strategies such as manipulation of attention map [1] and latent noise inversion [7] for enabling *precise* editing.

7.2. TRIFUSER Ablations

As described in the main paper, analogies can have multiple valid interpretations, and even a single interpretation may yield several visually-related variations. To account for this multiplicity, we generate k output patterns for each input set (A, A', B) and select the one that maximizes each metric. We first elucidate the relation between the number of generated sample k and the different metrics in Figure 9. We show four plots, one for each metric. Each plot has the number of samples k as the x-axis, and the metric (e.g. LPIPS, SSIM) on the y-axis. These plots reveal that for perceptual similarity metrics, TRIFUSER triumphs over the baselines across all values of k. Furthermore, as we increase k, TRIFUSER significantly closes the gap between itself and *Inpainter* when measuring SSIM. More importantly, these plot reveal that using a smaller number of samples ($k = 5$ as used in the main paper) is sufficient, and performance does not drastically decrease as k is decreased from 16.

We also evaluate all the models separately for each type of edit in the synthetic validation set. We measure the average ($1 - \mathit{LPIPS}$ with $k = 5$) (so that higher value indicates better performance) for each type of edit and visualize the results a radar plot as show in Figure 10. We observe that TRIFUSER surpasses all the baselines across the different types of edits. For more details regarding the edits, please refer to section 5.

7.3. TRIFUSER Scalability

Recent research has shown that scaling neural approaches, in terms of computational complexity and dataset size, is fundamental for achieving compelling results. Consequently, it is critical to investigate the *scalability* of novel models/methods. In this section, we study the scalability of TRIFUSER with respect to its training dataset size and its training compute budget.

First, we perform ablations to elicit the relation between training dataset size and TRIFUSER performance. We train three variations of TRIFUSER each with a dataset size of $100,000$ samples, $500,000$ samples and 1 Million samples respectively. The performance of these three methods is then compared on the held-out synthetic validation set. The resulting metrics are visualized as line-plots in Figure 11. Here, we provide 4 plots, one for each metric, similar to the format in Figure 9. The x-axis corresponds to the number of samples (k), and the y-axis corresponds to the respec-

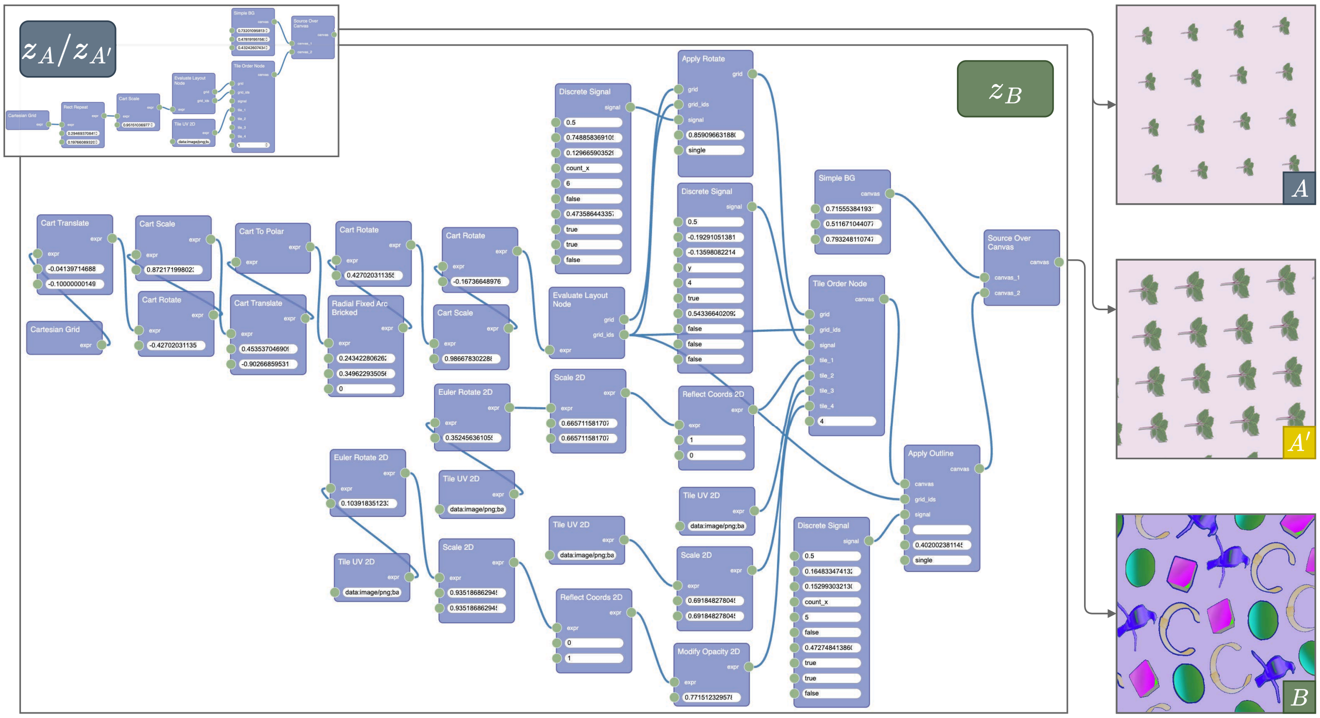

Figure 13. We show an example of a complex synthetic pattern B which has a SPLITWEAVE program z_B with 31 nodes. Inferring such programs automatically, i.e. VPI, is infeasible. Our approach, in contrast, allows to use to construct simple program z_A, and create analogical patterns (A, A') to parametrically edit B, *without* inferring z_B. The task of constructing z_A is significantly easier (in this example, z_A contains 8 nodes, only $\sim 25\%$ of z_B's size).

tive metrics. We notice a meaningful increase in the performance across the different metrics, as we increase the scale of the training dataset. This indicates training the method in future with larger datasets containing even more pattern styles may result in further improvements.

Similarly, we study the effect of training compute budget on model performance. All our models are typically trained on 8 A100-40GB GPUs with a batch size of 224. To explore the relation between training budget and model performance, we train a variation of TRIFUSER on 8 A100-80GB GPUs with a batch size of 448. We report a comparison between these two models in Figure 12. As shown in this figure, increasing the batch-size results in further improvements to the model, indicating a positive correlation w.r.t. the training budget. In future, training TRIFUSER with a larger training budgets may lead to further improvements in the model's performance.

8. Qualitative Results

We now present qualitative results to emphasize the utility and impressive capabilities of our method. As discussed earlier, a primary motivator for our approach is that Visual Program Inference attempts to infer the a program that fully replicates the input pattern, which not only is a hard task, but also results in a tedious editing experience as the user often has to fiddle with various parameters to ascertain *which* parts of the program must be edited to attain the desired edit. In contrast, with our approach, the user only has to construct the program for (A, A') which demonstrate *which* property to edit and *how* to edit it. Particularly, A does not need to even be similar to B, making the task of constructing the programs $(z_A, z_{A'})$ considerably simpler.

In Figure 13, we compare the program of a complex target pattern B, marked as z_B, with the simple program, z_A constructed to create a analogy pair (A, A') for editing the layout of B. While z_B contains 31 operator nodes, z_A contains only 8, which is $\sim 25\%$ of the size of z_B. We make the following notes: (a) The task users need to perform — that of creating z_A — is significantly easier than the task of inferring z_B, (b) Using the analogical editor inducing parametric control over pattern B based on the program z_A. Consequently, to perform simple edits of pattern B, the user only needs to specify a simple program z_A.

As mentioned previously, analogies can have multiple valid interpretations, and even a single interpretation may yield several visually-related variations. Consequently, a analogical editor must also be capable of producing multiple interpretations for any given input analogy pairs. Having such a one-to-many mapping, as our model has, is more

Figure 14. TRIFUSER generates multiple *equally-valid* yet different edited images B'.

suitable for editing as the user can select the edited pattern that matches their edit intent from multiple generations. In contrast, restricting to a singular interpretation may more easily lead to scenarios where the system's and user's interpretation of the input analogy differ.

In Figure 14, we present analogical edits performed on real-world patterns by our method, highlighting the generation of different *equally valid* analogy interpretations. The first row corresponds to an edit for removing a random coloring variation effect on the input pattern B. TRIFUSER produces two outcomes, both reasonable as pattern B does not make it clear what the tile's original color is. The example presented in the second row corresponds to an edit to modify the background of the input pattern. However, its unclear if the muted ellipses behind the lion tiles are part of the tile, or part of the background. Consequently, some generations keep these ellipses updating their color accordingly, while other generations eschew them to provide a uniform colored background as shown in A'. Finally, the third example corresponds to an edit on the layout of the input pattern. We show two equally reasonable outputs generated by our model as the underlying orientation of the bone tile is ambiguous.

Finally, in figure 15, we demonstrate the ability of our model to reasonably edit patterns in styles unseen during training. Additionally, we present additional qualitative results comparing our method to the other baselines in Figure 17. Images comparing the four methods across the entire test set is also provided in the supplemental material.

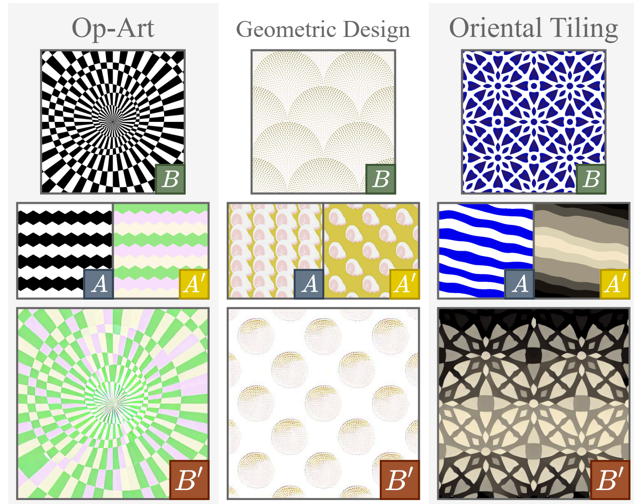

Figure 15. We present additional examples that show that TRIFUSER effectively edits patterns from novel pattern styles not present in the training dataset.

9. Failure Cases

We present and discuss some recurring failure cases for our method. Figure 16 provides 6 exmaples from our test set of real-world patterns where our method fails to generate a reasonable analogical edit. When editing the layout of patterns, our model still sometimes struggles to retain the fine-details of the input pattern's tile, particularly when they contain text — this is demonstrated in example (b) and (d). Another mode of failure is when the edit does not *fully* perform the intended edit, as visible in example (c) and (e). In (c) though the model adds a color change effect on B as intended, it produces color variations that do not match the color variations shown in A, A'. This is due to the usage of *relative* color shifts (with respect to a hue-wheel) in our synthetic patterns. Similarly, in (e), while the model correctly removes the tile scaling effect, it replaces the fish tile with the cat tile. Finally, a few failure cases also emerge due to the model failing to understand the input analogy pair, as show in examples (a) and (f).

10. Limitations

While our method demonstrates robust performance and versatility, there are a few limitations that merit discussion.

The primary limitation lies in the reliance on a synthetic dataset of analogies. To extend this technique to other domains, users must construct a domain-specific language (DSL) and define editing functions. Furthermore, real-world applicability of our method depends on the coverage of the DSL and the editing functions. Although we

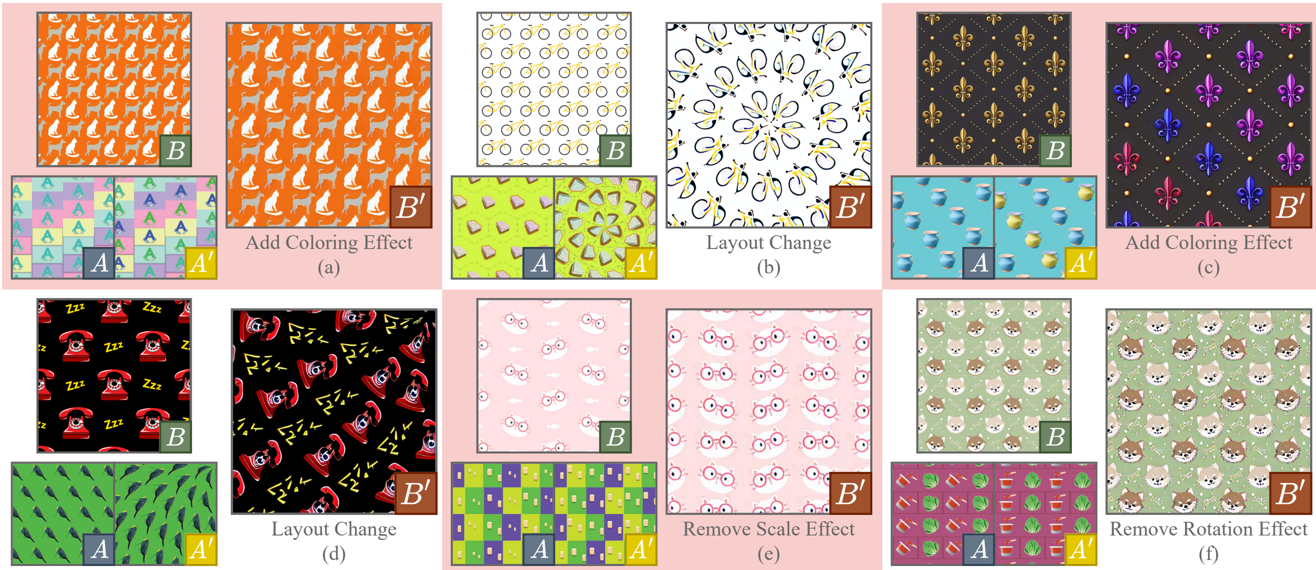

Figure 16. We present examples on the test-set where our method fails to produce a reasonable edit. Edits sometimes fail due to poor retention of tile-details ((b) and (d)), or imperfectly applying the edit demonstrated with (A, A') ((c) and (e)) or failing to understand the input analogy ((a) and (f)).

demonstrate generalization to related pattern styles, the current scale of the dataset limits the model's ability to handle entirely novel pattern styles or edits. However, this limitation may be addressed by automatic the construction of analogical data from multiple domains such as ShaderToy shader code. Such data could enable pretraining on a broader scope of analogical variations before fine-tuning for specific domains. Additionally, various visual domains such as Zentangle patterns, materials, Lego already contain well defined DSLs making it easier to extend our framework to other structured visual data domains.

A second drawback is that analogies, while universal in their ability to represent arbitaray edits, are not always the most efficient modality for conveying edit intent. For example, simple edits such as color changes might be more easily performed through direct recoloring of the canvas. Furthermore, the inherent flexibility of analogies, which allows multiple interpretations, can sometimes make it tedious to sample and select a desired output. This issue could be mitigated by coupling analogies with text-based guidance or other constraints to make the process more directed and user-friendly.

Finally, using the system requires constructing analogy pairs, which depends on the user's familiarity with SPLITWEAVE or node-based programming. While this could pose a barrier to some users, the increasing adoption of node-based tools in visual programming provides a promising path forward. Future research into improving user interaction for visual programming and analogical editing could further lower this barrier and make the system more accessible.

Despite these limitations, our work provides a flexible framework for analogical pattern editing and highlights several avenues for future research, including extending analogical datasets, improving edit specificity, and enhancing user interfaces.

Figure 17. Qualitative comparison between patterns generated by our model, TriFuser, and the baselines. TriFuser generates higher quality patterns with greater fidelity to the input analogy.

References

[1] Yuval Alaluf, Daniel Garibi, Or Patashnik, Hadar Averbuch-Elor, and Daniel Cohen-Or. Cross-image attention for zero-shot appearance transfer. In *ACM SIGGRAPH 2024 Conference Papers*, New York, NY, USA, 2024. Association for Computing Machinery. 9

[2] Omer Bar-Tal, Lior Yariv, Yaron Lipman, and Tali Dekel. Multidiffusion: Fusing diffusion paths for controlled image generation. *arXiv preprint arXiv:2302.08113*, 2023. 7

[3] Rete.js Contributors. Rete.js: Javascript framework for visual programming, 2023. Version 1.x, accessed November 20, 2024. 3

[4] Keyan Ding, Kede Ma, Shiqi Wang, and Eero P. Simoncelli. Image quality assessment: Unifying structure and texture similarity. *CoRR*, abs/2004.07728, 2020. 8

[5] Stephanie Fu, Netanel Tamir, Shobhita Sundaram, Lucy Chai, Richard Zhang, Tali Dekel, and Phillip Isola. Dreamsim: Learning new dimensions of human visual similarity using synthetic data. *Advances in Neural Information Processing Systems*, 36, 2024. 8

[6] Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983. 5

[7] Amir Hertz, Ron Mokady, Jay Tenenbaum, Kfir Aberman, Yael Pritch, and Daniel Cohen-Or. Prompt-to-prompt image editing with cross attention control. 2022. 9

[8] George A. Miller. WordNet: A lexical database for English. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*, 1994. 4

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach 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*. Curran Associates Inc., Red Hook, NY, USA, 2019. 3

[10] Scott Reed, Yi Zhang, Yuting Zhang, and Honglak Lee. Deep visual analogy-making. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, page 1252–1260, Cambridge, MA, USA, 2015. MIT Press. 8

[11] Yasheng Sun, Yifan Yang, Houwen Peng, Yifei Shen, Yuqing Yang, Han Hu, Lili Qiu, and Hideki Koike. Imagebrush: learning visual in-context instructions for exemplar-based image manipulation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2024. Curran Associates Inc. 8

[12] Yoad Tewel, Yoav Shalev, Idan Schwartz, and Lior Wolf. Zero-shot image-to-text generation for visual-semantic arithmetic. *arXiv preprint arXiv:2111.14447*, 2021. 8

[13] Gregory K. Wallace. The jpeg still picture compression standard. *Commun. ACM*, 34(4):30–44, 1991. 5

[14] Xiaohua Zhai, Basil Mustafa, Alexander Kolesnikov, and Lucas Beyer. Sigmoid loss for language image pre-training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 11975–11986, 2023. 4

[15] Lvmin Zhang and Maneesh Agrawala. Transparent image layer diffusion using latent transparency. *ACM Trans. Graph.*, 43(4), 2024. 4, 5

[16] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018. 8