Pattern Analogies
Learning to Perform Programmatic Image Edits by Analogy
Supplementary

Aditya Ganeshan*
Brown University

Thibault Groueix
Adobe Research

Paul Guerrero
Adobe Research

Radomír Měch
Adobe Research

Matthew Fisher
Adobe Research

Daniel Ritchie
Brown University

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 accepted.

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

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

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 `SExpr`s.

*Work performed during an internship at Adobe Research

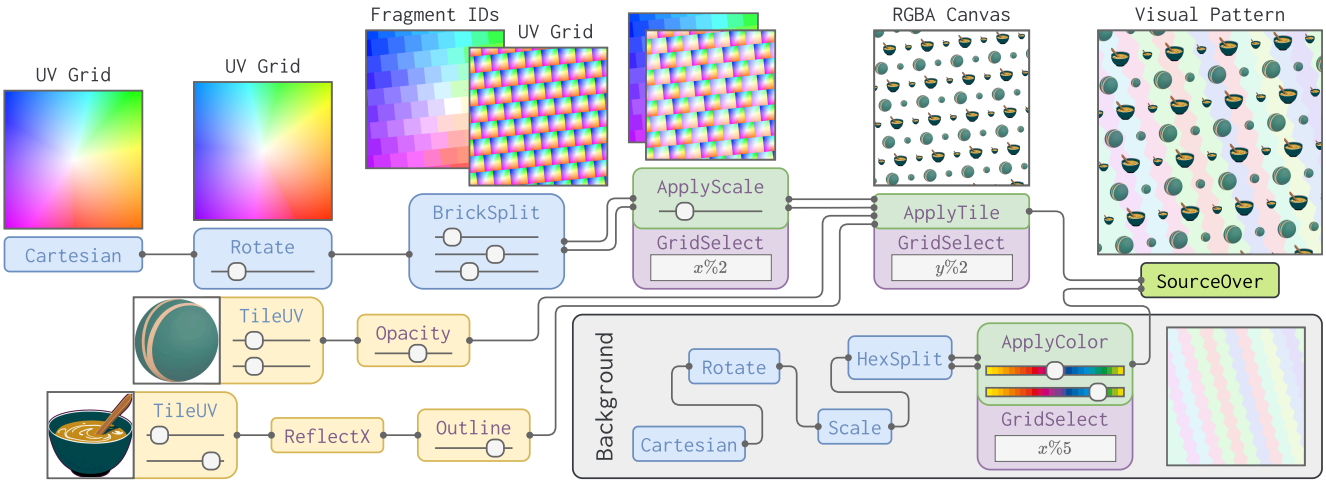

Figure 1. **Program evaluation** We illustrate the evaluation of a SPLITWEAVE program. SPLITWEAVE is used to create directed acyclic graphs representing data flow between different operators. The *UV Grid Operators* are used to define `UVExprs`, which map spatial coordinate to UV grids. *Signal Operators* 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.

`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 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

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

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 SExpr s 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. 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 [10], 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:

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

where 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 that represents the steps involved in creating a pattern. For example, in Motif Tiling Patterns (MTP), includes stages such as sampling tiles, sampling layout parameters, and sampling effects like background elements. Each stage in corresponds to a node or sub-tree in AT, where the nodes represent specific compo-