Then Try This • Algorithmic Pattern Salon

Parameterizing Patterns for Generative Art and Live Coding

Jessica Stringham

Then Try This

Published on: Nov 11, 2023

URL: https://alpaca.pubpub.org/pub/dpdnf8lw

Abstract

Patterns can provide an interesting backbone to generative art and live-coded visuals. Coupling patterns with comprehensive parameterization enables interactive exploration of generative art states to find interesting pieces or to tour a range of parameters in a real-time performance. I'd like to share an overview of the software I've been developing in Rust to support my creative code work, including how it uses parameterization to create and explore patterns.

I’d like to share an overview of the software I’ve been developing in Rust to support my creative code work, including how I use parameterization to make patterns. This is my first time writing about the software in any detail, and it’ll also include a demo of writing a simple system that can be used to live code visual patterns.

Before I get too into the weeds, here are a few examples created by similar code. First is a 48 by 48 grid of squares, where the pattern in the visibility of a square is determined by a bitwise operation, and the size of a square is determined by a formula.

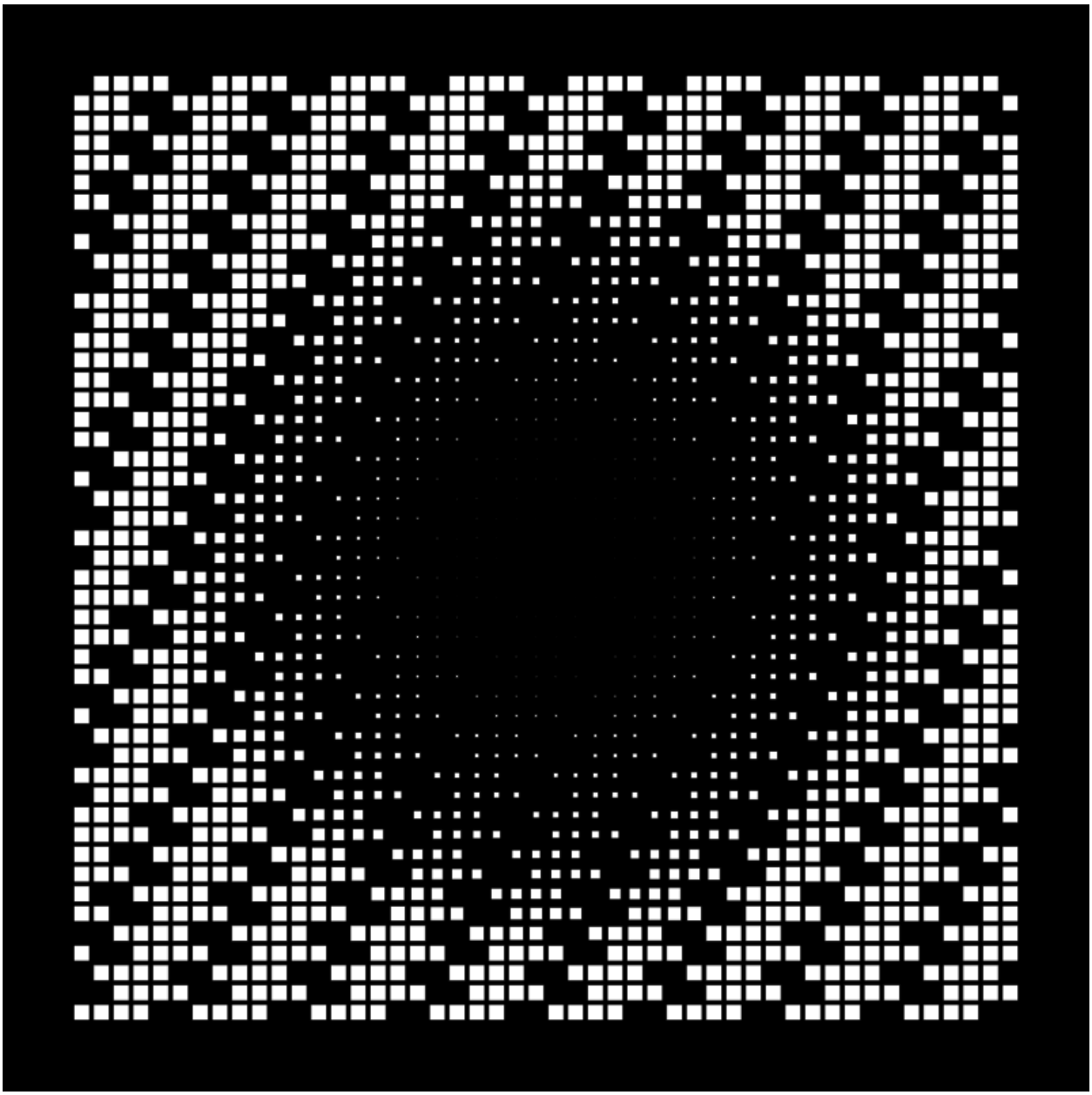

Figure 1
The original pattern.

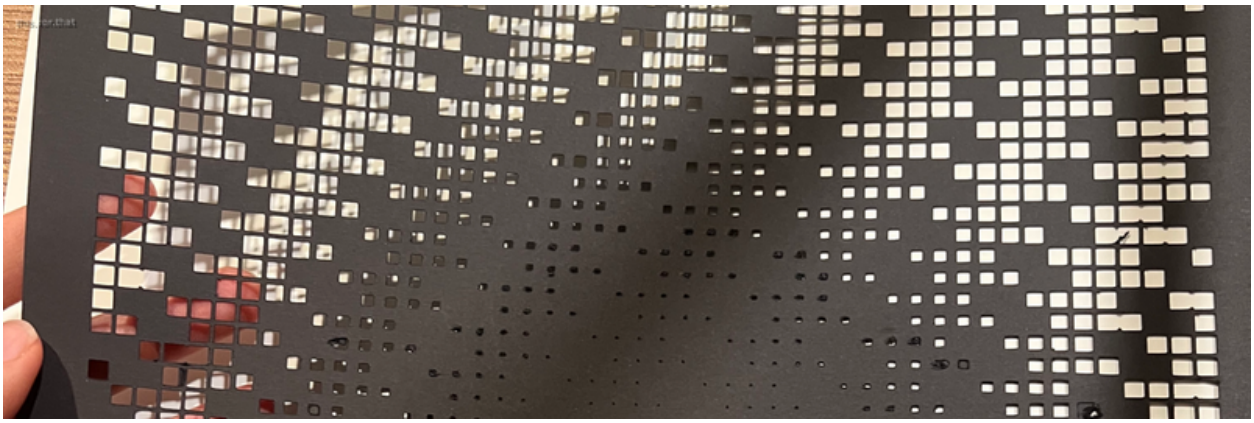

Figure 2
The pattern after going through a vinyl cutter.

Using the same software packages, I can create another system that feeds the grid through shaders to apply effects like distortion, colors, and feedback.

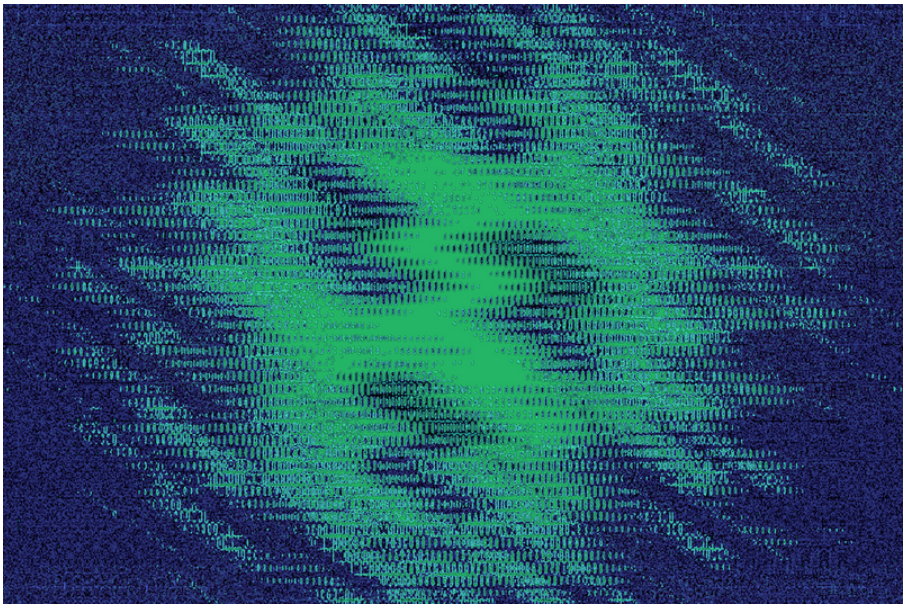

Figure 3
Similar grid pattern (but inverted) is still visible after going through a lot of filters.

I perform live coded visuals, where I control real-time graphics to go with an audio performance. When live coding visuals, additional patterns can be introduced along the time dimension to complement audio's temporal patterns.

Figure 4
The grid pattern is still visible while the author makes a funny face, performing at a livecode.nyc event with Hardcore Software on audio. (Photo credit Doug Linse).

Overview

I have been writing several connected libraries in Rust that enable my live coded visuals, generative art, and many other creative coding endeavors. The paradigm works like this: I create a new *system* for a given performance, project, or doodle and import the libraries. A given system might have dozens of *parameters*, like the color and position of an object, or constants for a physics simulation or an L-system grammar. I can then explore the system interactively by adjusting numerical and categorical parameters by setting the value directly or using midi controllers, audio-reactivity, time normalized by the tempo, or an expression combining all of those. I have a lot of fun with this. It encourages me to crank up parameters beyond what I intuitively thought was interesting, and that's where I find the coolest outputs.

Related Libraries

I make use of nannou, a package written for Rust that is very similar to other creative coding libraries like Processing, p5js, and OpenFrameworks. These packages generally provide things like a way to draw and transform shapes, handle colors, and take care of the event loop to display things on a screen. They are open-

ended and flexible, so you can pull in additional packages from their language's environment, and that's where my libraries sit.

I also draw inspiration from other software used for live coding. One is Tidal Cycles, a live coding environment commonly used for audio with an emphasis on patterns over time. And while I have a different approach than Hydra, a live coding video synth environment, I am inspired by what people can do with it and other visual libraries. Unlike these other live coding environments, I do need to write and compile the Rust code ahead of time that defines the system and its parameters.

Demo

For this demo, we will once again draw a grid of squares through the configuration. Depending on the goals of the performance, one way to parameterize would be to specify how the squares are arranged on the screen, the number of squares, the squares' positions, sizes, rotations, and colors/designs. Different systems are capable of producing the the same results as a grid of squares; for example, one could parameterize based on the ratio of a parallelograms' width/height and its angle, or using the number of sides of a regular polygon. The choice depends on the performance. For the following example, we'll restrict the program to produce squares.

A starting example

Let's work up to the grid. Here's how I could start setting up the parameters and using that to draw the system: I manually specify the location, size, and the color for three squares.

```
# square.yaml

# draw three squares with different colors
squares:
 - loc: [-200.0, 0.0]
   size: 50.0
   color: [0.1, 1.0, 0.8, 1.0] # hsva
 - loc: [0.0, 0.0]
   size: 100.0
   color: [0.5, 1.0, 0.8, 1.0]
 - loc: [200.0, 0.0]
   size: 50.0
   color: [0.8, 1.0, 0.8, 1.0]
```

Below is the Rust code that I use. If you're not familiar with the Rust code, it's okay to skim this code! In Rust, the `#[derive(…)]` is a custom proc-macro, a way to write code to write code. The Livecode library I've written provides the proc-macro which creates code that can parse the configuration, fill in midi/time/audio, and compute expressions. So in Rust, I only need to write the code to draw the shape given the already-computed values.

```
# squares.rs
#[derive(Livecode)]
struct Square {
  size: f32,
  loc: Vec2, // yaml is a list of two numbers
  color: LinSrga // a color, the corresponding yaml is a list of 4 items representing HSVA
}
impl Square {
```