Demo: Kaleidogen

Joachim Breitner
DFINITY Foundation
Germany
joachim@dfinity.org

Abstract

Kaleidogen lets you breed abstract circular patterns. You can crossbreed two and add their offspring to your stock. The game has no end, no score, no time pressure, the only goal is to please your personal sense of aesthetics.

The mechanisms behind Kaleidogen imitate genetic inheritance. It is written in Haskell, compiled to JavaScript, runs in the browser and generates GL shader programs on the fly.

CCS Concepts • **Software and its engineering → Functional languages**; • **Applied computing → Computer games**; **Media arts**.

Keywords Haskell, generative art

ACM Reference Format:
Joachim Breitner. 2019. Demo: Kaleidogen. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design (FARM '19), August 23, 2019, Berlin, Germany.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3331543.3342581

1 Inception

After a day of intensive mental work, your brain is still running at full speed. What kind of computer game would help you to gracefully calm down? It's not so obvious: If the game is too simple or passive, then your brain might still wander off and continue to think about work. A fast-paced intensive game however, like a shooter, would capture your attention, but will likely be more stressful than relaxing. Can a game be captivating and relaxing at the same time?

These considerations were at the beginning of a process that led to *Kaleidogen*, a small game that may not yet fit the bill, but still turned out to be rather nice.

2 User Experience

When you open http://kaleidogen.nomeata.de/, you will see a screen as in Figure 1. The bottom row is your current population of patterns. If you select two, either by clicking on two, or by dragging one on top of another, the patterns will be crossbred, i.e. mixed, and their offspring is shown on top. If you like what you see, you can add it to your population, and use it to breed further, more complicated patterns.

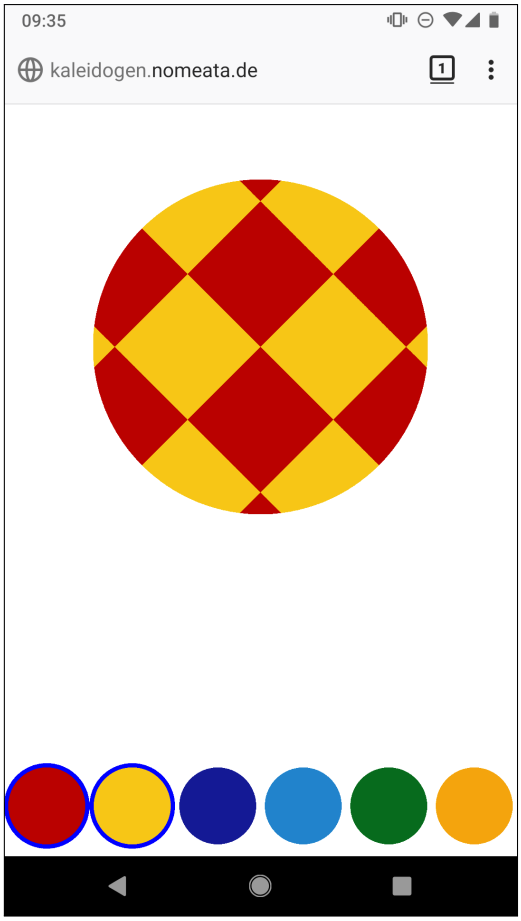

Figure 1. Kaleidogen's initial screen

When you crossbreed two patterns, the new pattern might inherit the colors of one or both parents, or aspects of the shape of the parents. Sometimes an aspect is dropped, and sometimes a completely new element is added. The process is mostly unpredictable, and you cannot breed a specific pattern in a targeted way; this is intentional. Every new breed should be partly plausible and partly surprising. After

FARM '19, August 23, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6811-7/19/08.
https://doi.org/10.1145/3331543.3342581

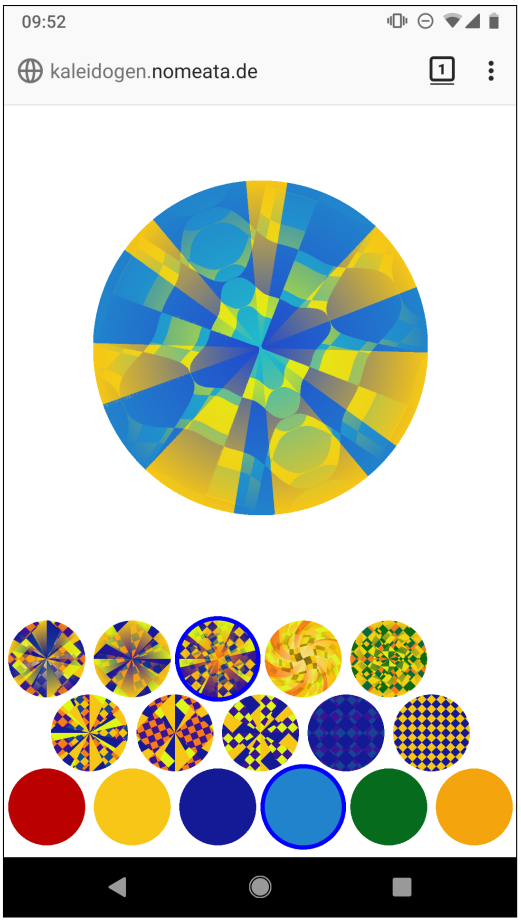

Figure 2. After a few steps

a few steps, the patterns become more intricate and more interesting, as you can see in Figure 2.

You can also remove patterns from your population, and save individual patterns as images. That is all there is: the game has no end (as your population grows, the circles become smaller to fit the screen), no score, no winning condition. You cannot even save the state of the game: your creations are ephemeral.

3 Under the Hood

Despite this simplicity, there are a few interesting things happening under the hood.

3.1 Pattern Genetics

The goal is that newly generated patterns are plausible descendants of their parents, also but unpredictable (and hence pleasantly surprising). To that end, each pattern is represented by sequence of bytes, which is suggestively called the DNA. For example, the pattern in Figure 2 is represented by `0x812B73A74D1C4D874E`. These genomes can be rendered as a pattern, or crossbred to form a new one:

```
data RNA
    = Solid (RGB Double)
    | Rotate Double  RNA
    | Invert         RNA
    | Swirl Double   RNA
    | Dilated Int    RNA
    | Blend Double   RNA RNA
    | Checker Double RNA RNA
    | Rays Int       RNA RNA
    | Gradient       RNA RNA
    | Ontop Double   RNA RNA
```

Figure 3. Pattern expressions

3.1.1 Gene Expression

When interpreting the DNA as a pattern, it is turned into an expression tree, suggestively called the RNA: The leaves are circles filled with a plain color, and the nodes are (unary) transformations or binary compositions of patterns. Figure 3 describes the possible operations as a Haskell data type.

It turned out that too much uncontrolled variation of color is a problem, therefore color is encoded in the DNA separately from the shape. The first two bytes simply select two of the six initial base colors, which were chosen manually. From each base color, a seven shades (unmodified, darker, brighter, more intense, less intense, hue shifted by ±20°) are created, shuffled, interleaved, and cycled to form an infinite list. This list is used for the leaves of the RNA tree.

The remaining bytes encode this tree: The first 4 bits select one of the 16 operations in Figure 3. Some operation have a parameter, such as the size of the squares, or the angle of rotation; the next 4 bits are used for that. For the unary operations, all following bytes are parsed as the subexpression; for the binary operations, the remaining bytes are first evently split into two sublists. When no bytes are left, we have reached a leaf of the tree, and take the next shade from the list of colors.

One nice effect of this encoding is that the trees tend to be relatively balanced.

3.1.2 Crossbreeding

To crossbreed two patterns, we again treat the first two color-encoding bytes differently: From each parent, we randomly pick one color. This ensures that offspring does not suddenly have base colors that were not present before, which turned out to be too surprising.

The remaining bytes from each parent are concatenated, and mutated as follows: With probability $p = \min(1, e^{-\frac{l-2}{20}})$, where l is the number of bytes in the concatenation, we add a fresh random byte to the beginning, and remove each byte in the list with probability $(1 - p)$. This way patterns grow quickly initially, but not without bound.

We seed the random number generator with the two input DNAs; this turns this "random" process into a deterministic process, and prevents the user from cross-breeding the same two patterns over and over.

3.1.3 Abstraction Mismatch

Observe that cross-breeding operates on the raw bytes, and not on the expression trees. This mismatch of abstraction is intentional: This way, the tree may be broken up and re-assembled differently. Nevertheless, the algorithms are co-designed. For example, by adding new random bytes to the *beginning* of the list of bytes, the new operation will be the root of the tree and hence very visible.

Coming up with these algorithms is an art, not a science. In no sense are they finished or optimal, and we tweak the continuously.

3.2 Technology Stack

Figure 3 already gives it away: Kaleidogen is implemented in Haskell. To make it usable as a web application, we compile it to JavaScript using `ghcjs`. The user interface itself is essentially one big HTML Canvas that we draw on.

It was quickly clear that actually generating the pictures pixel by pixel in Haskell is going to be far too slow. Therefore, the Haskell code renders a pattern of type `RNA` not directly, but generates a GL shader program from it – a task where Haskell excells at. The whole user interface is rendered using WebGL. Despite drawing each pattern completely anew in each frame, the user interface is snappy on a Desktop browser. On mobile devices it becomes sluggish with many patterns, and further profiling and performance turning is necessary.

We can cross-build and package Kaleidogen as an app for Android using the infrastructure from `reflex-dom`, `jsaddle` and Nix: It still uses the web view for the UI, but the actual code runs naively (and thus more performing).

Building almost exclusively on GL made it very simple to also build Kaleidogen as a native Desktop application, using the SDL library, for more direct access to the graphics system. We hope to use this back-end also for the Android app, but are currently facing problems cross-building Haskell SDL applications.

3.3 User Interface

An earlier version of Kaleidogen used functional reactive programming (FRP), using the proven `reflex-dom` library, to model the user interface. We found that this was great as long as the state of the UI is mostly derived from the abstract, logical state of the game, but it turned out to be problematic once we added animations and drag-n-drop interactions: Now the state of the UI is heavily dependent not only on the current state of game, or the recent states of the game, but also on *why* the game changed its state – for example, did the user abort the drag operation, or did they complete it, but created a pattern that was already there, which we want to animate differently?

Therefore, the code is now a layer of state machines where state transitions not only define a new state, but also an abstract notion of how to get from the old state to the new one, which is interpreted by the outer layer. The inner state machine deals with abstract positions for patterns (the single "big one" and multiple, indexed "small ones"), the next one turns these positions into actual coordinates, taking the window size into account, animating the changes if necessary. The last layer implements mouse interaction including dragging; this isolates the lower layers from the temporary displacement of a circle during dragging, and the animation when the circle moves back when dragging ends.

Now that we have found a factoring of concerns that works, it might be possible to express this design again using FRP abstractions, for additional robustness; this is ongoing work.

4 Conclusion

Kaleidogen is an interesting experiment on many layers: Is it a good game? How to make the user interface intuitive and pleasant to use? Which colors and patterns to use, and how to best combine them? How to build a performing cross-platform app using Haskell? How to structure the program architecture to disentangle abstract logic, UI layout, animation and interaction? We hope that our ideas can be an inspiration, and at the same time we are looking forward to your input[1].

Acknowledgments

I would like to thank Christina Zeller, Rebecca Schwerdt and Juliane Jastram for playtesting and valuable input, especially concerning the arcane art of picking colors that go well together.

[1] or even contributions at https://github.com/nomeata/kaleidogen