Then Try This • Algorithmic Pattern Salon

The complete guide to live-coding visuals in Punctual

Joan Queralt Molina

Then Try This

Published on: Nov 11, 2023

URL: https://alpaca.pubpub.org/pub/lb3o0yti

Abstract

In this paper, the *complete guide to live-coding visuals in Punctual* is presented: a new documentation effort for the visuals aspect of the Punctual live-coding language. This guide aims at documenting every detail of the features offered by Punctual regarding the creation and modification of visual algorithmic pattern through live-coding, some of them known only to the most active and up-to-date users of the language.

Introduction

Punctual is a language suitable for live coding both audio and visuals. Although it has a big creative potential, it is a very little known and used language amongst the live-coding community.

One of the main reasons for this is the lack of a good, complete documentation of all the features the language provides. Even though its author, **David Ogborn**, keeps an up-to-date reference of the language, there is still a lack of explanations, examples, and ideas on how to use the different features it provides. In addition to this, there are some powerful features that aren't documented at all.

Here, the complete guide to live-coding visuals in Punctual is presented. This guide intents to make up for the missing official documentation, providing a new source of information regarding the visual part of the Punctual live-coding language, with plenty of examples and creative ideas, both for the beginner and the experienced live-coder.

Also, my aim in this Algorithmic Pattern Salon is to offer an on-site workshop in Barcelona centered around Punctual and its capabilities to create and evolve interesting patterns using the examples developed in the aforementioned guide, which is still under construction.

About Punctual

From the official Punctual repository:

> Punctual is a language for live coding audio and visuals. It allows you to build and change networks of signal processors (oscillators, filters, etc) on the fly. When definitions are changed, when and how they change can be explicitly indicated.

Punctual has huge visual capacities. Its strong points are:

- Runs in a browser, no installation needed.
- Fully integrated into the Estuary collaborative live-coding environment.
- Compact syntax allows fast pattern creation and further modification.
- Direct access to pixel coordinates allows the creation of patterns using any mathematical formula.
- Low-level geometry changing functions lead to a great flexibility.

- Flexible graphs arithmetic for even more creative possibilities.
- Simple but effective modulation functions. Modulate everything.
- Audio reactive visuals using frequency analysis, FFT (Fast Fourier Transform) and internal tempo.
- Feedback allows building complex patterns using the last frame as source.
- Capacity to use remote images and videos.
- Can use webcam as source.
- Easy to get help via the Estuary discord server.

Compared to Hydra (arguably the best known live-coding language for visuals), it has the following limitations:

- Lack of high-level effects (saturation, pixelation, etc.)
- Some mathematical skills are needed to build complex patterns using the low-level functions Punctual provides.
- Not easy to extend with your own functions.
- Not very well documented (until now, I hope).
- Not many people using it.

Punctual was designed to be used mainly inside Estuary, a platform for collaborative live-coding. When implementing such a web application, security is a major concern: the code a participant in a jam writes, is sent and executed on the other participants computers. If not done right, this allows a potential attacker to run arbitrary code there. Punctual avoids this issue by parsing all the code itself. Most live-coding languages are extensions or libraries that can be added to a base general programming language (for example, TidalCycles with Haskell, or Hydra with Javascript). On the contrary, Punctual is its own little, independent language, thus avoiding the mentioned security problem as it only allows to write sound and visual code.

Disclaimer

Punctual is a personal art project by **David Ogborn**. He likes to keep absolute freedom on how or when Punctual evolve, and that's the main reason why he doesn't usually accept contributions into the source code or the official documentation.

The *complete guide to live-coding visuals in Punctual* is an unofficial document and can be made obsolete by changes to Punctual at any time, even though I'll try to keep it up to date. This is specially true for any officially undocumented feature that may appear there.

The only official documentation is maintained by David himself on the Punctual git repository. Make sure to check the README.md and REFERENCE.md files for up-to-date, official information on the project.

Free software, live-coding, and documentation

There is an ongoing issue in the live coding community, and I would say that it extends to the whole free software movement, that is neglecting documentation, resulting in limited accessibility and knowledge preservation. In some cases, there are functionalities already implemented, but there may pass months or years before it is truly made available to the users because of lack of documentation.

As an example, TidalCycles 1.7 introduced a feature known as "control buses" which allows to pattern effects while a sound is playing. TidalCycles 1.7 was launched on January '21, and its author, **Alex McLean**, wrote a message explaining them in the TidalCycles forum. Control buses didn't reach the official documentation until October '22, that is more than one a half year later. This is not to be taken as a critique of the project's documentation, but rather as an example of the challenge involved in keeping documentation up-to-date in an open-source software project.

Reasons for the lack of user-friendly, comprehensive documentation are often constraints in time and energy, but also that developers usually enjoy writing code rather than documenting it.

Fortunately, there are also very good documentation examples in the live coding and free software community. For example, the free online book "*Learn You a Haskell for Great Good!*" is a quite good attempt at making a difficult subject like Haskell (which is used to implement several live coding languages) more accessible. The "*Hydra Book*" is another noteworthy effort in the live coding community (and one of the inspirations for making this guide to Punctual), and resources like this partially explain the popularity of the Hydra language amongst live coders. As a last example of good documentation in free software, I would like to point out the absolutely delicious documentation of the Godot game engine (which is used to develop Animatron, a real-time environment to create visual poetry), which is both understandable, complete, and up-to-date.

Documentation efforts like this one could help mitigate this problem and make craft live coding languages available both for potential users and future researchers.

Patterns in Punctual

These are some of the pattern creation ideas that are (or will be) presented in the *complete guide to live-coding visuals in Punctual* and which can be studied during the proposed workshop.

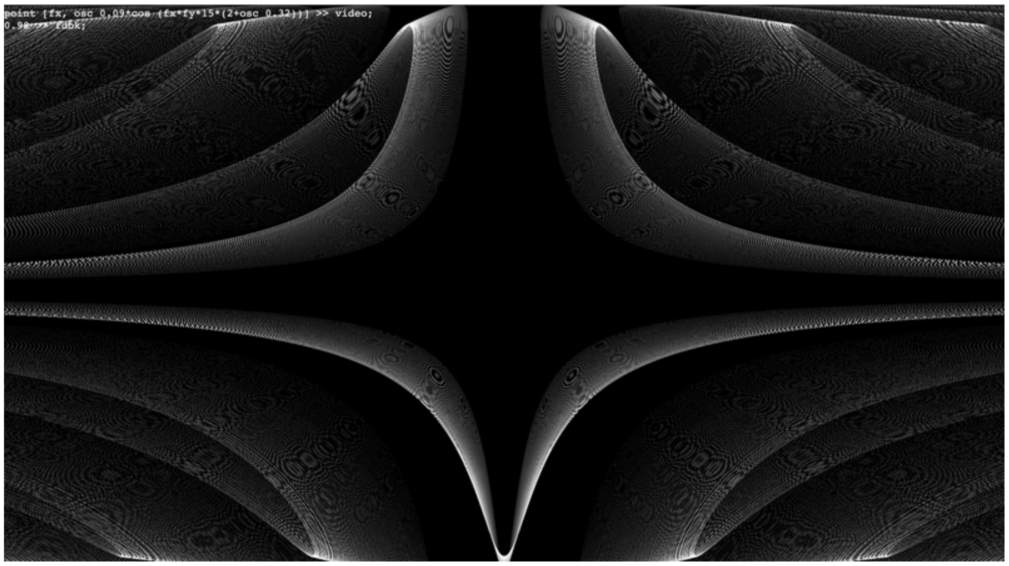
Many of these patterns are dynamic and best viewed pasting and running the code inside Estuary or the standalone online Punctual.

For these patterns, I added a summary of the explanation in the guide (which can be a bit confusing, as here we are skipping a lot of steps).

Using Punctual to draw mathematical functions that create patterns

Here, we use Punctual capacity to define points to draw a mathematical function of the form `y=f(x)`. Our function depends on the fragment coordinates (`fx` and `fy`) and include two oscillators to make the resulting pattern evolve through time. Lastly, we add a fair amount of feedback to create a blurring effect:

```
point [fx, osc 0.09*cos (fx*fy*15*(2+osc 0.32))] >> video;
0.98 >> fdbk;
```



```
point [fx, osc 0.09*cos (fx*fy*15*(2+osc 0.32))] >> video;
0.98 >> fdbk;
```

Figure 1

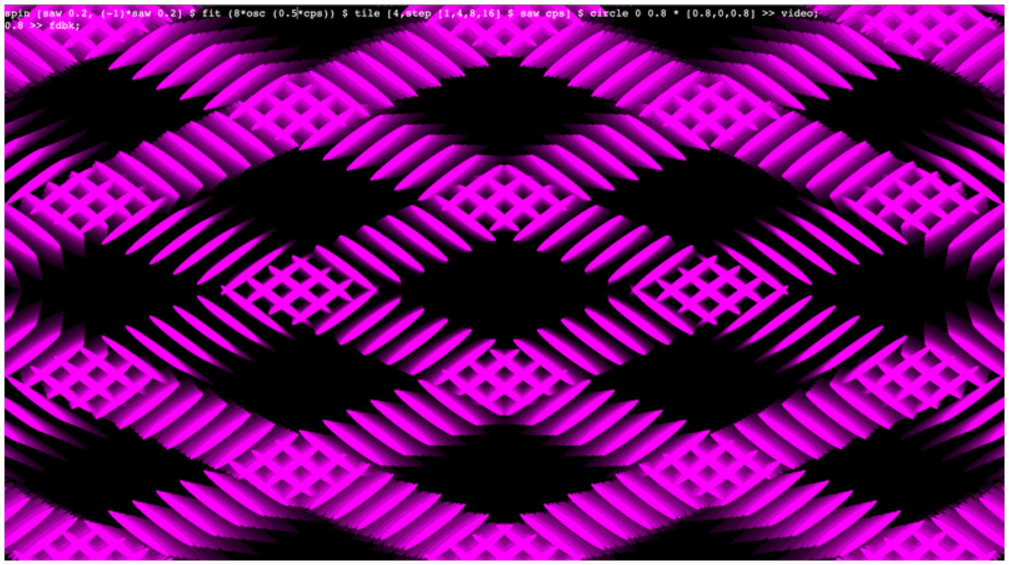
Deforming the aspect ratio through time

Writing a *complete* guide to something implies going through each and every function and possibility, and sometimes this process leads to amazing discoveries.

The `fit` function modifies the aspect ratio of the display, and it’s often used to fit images into the screen, or to avoid the distortion of some shapes (for example, circles) due to the screen being (usually) wider than higher.

But, under further examination of its possibilities, we see that `fit` can also be used in more creative ways. In this example, we draw a pattern of circles, and then use `fit` to modify the aspect ratio as time passes, creating an interesting effect. `step` modifies the number of circles through time, and `spin` is used to duplicate the pattern, and spin each copy in opposite directions, to create a dynamic symmetry:

```
spin [saw 0.2, (-1)*saw 0.2] $ fit (8*osc (0.5*cps)) $ tile [4,step [1,4,8,16] $ saw cps] $ c
0.8 >> fdbk;
```



```
spin [saw 0.2, (-1)*saw 0.2] $ fit (8*osc (0.5*cps)) $ tile [4,step [1,4,8,16] $ saw cps] $ circle 0 0.8 * [0.8,0,0.8] >> video;
0.8 >> fdbk;
```

Figure 2

Using polar coordinates to create symmetries and complex patterns from simple shapes

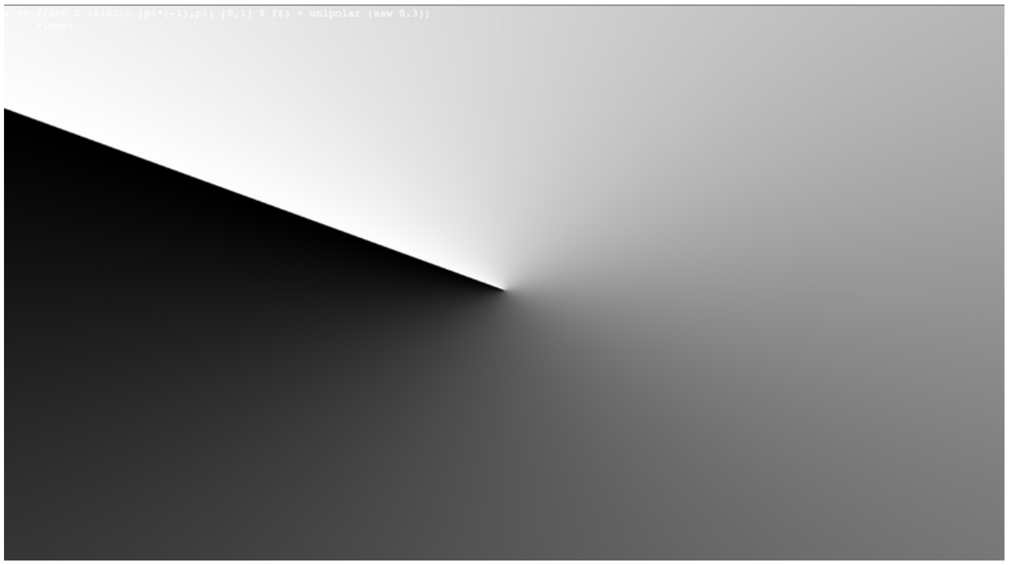
This example is a complex one, and it involves many of the ideas present in Punctual.

In the first line, `r` is defined, for each fragment, essentially as the fragment's angle.

The angle is rescaled (with `linlin`) from a -π to π range to a 0 to 1 one, to adapt it to an intensity. Then we add a saw oscillator (from 0 to 1) and finally keep only the fractional part (`fract`).

If we draw `r` the result is a kind of *sonar* effect, as the whiteness of a fragment only depends on the angle and the time:

```
r << fract $ (linlin [pi*(-1),pi] [0,1] $ ft) + unipolar (saw 0.3);
r >> video;
```

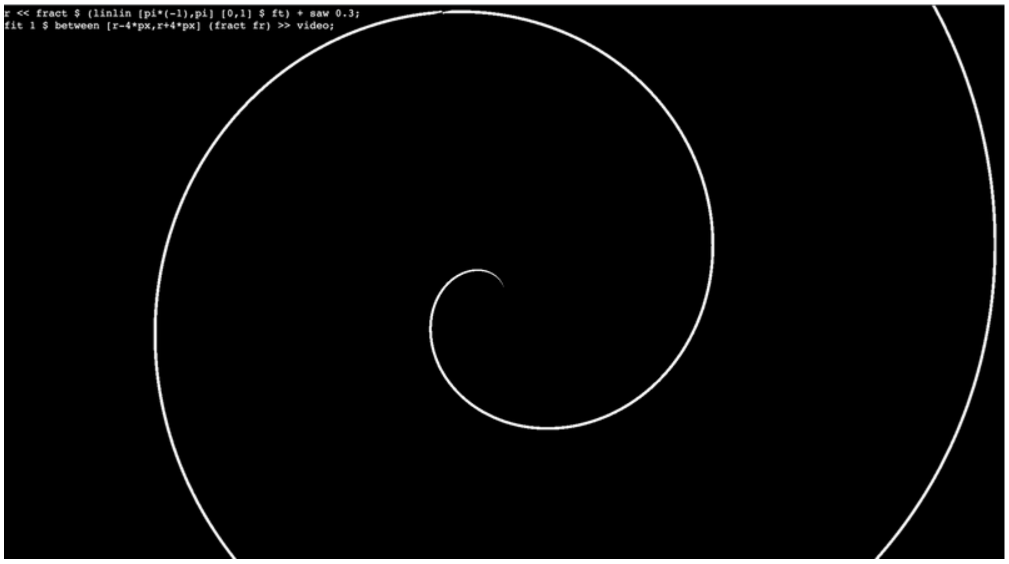


```
r << fract $ (linlin [pi*(-1),pi] [0,1] $ ft) + unipolar (saw 0.3);
r >> video;
```

Figure 3

In the next step, `between` is introduced. In essence, `between` receives two values and a graph and returns 1 if the graph is between the two values, or else 0. Here, if the fractional part of a fragment's radius is near enough to the previously computed `r` for that fragment, the result 1, otherwise is 0. This creates a kind of spiral shape:

```
r << fract $ (linlin [pi*(-1),pi] [0,1] $ ft) + saw 0.3;
fit 1 $ between [r-4*px,r+4*px] (fract fr) >> video;
```



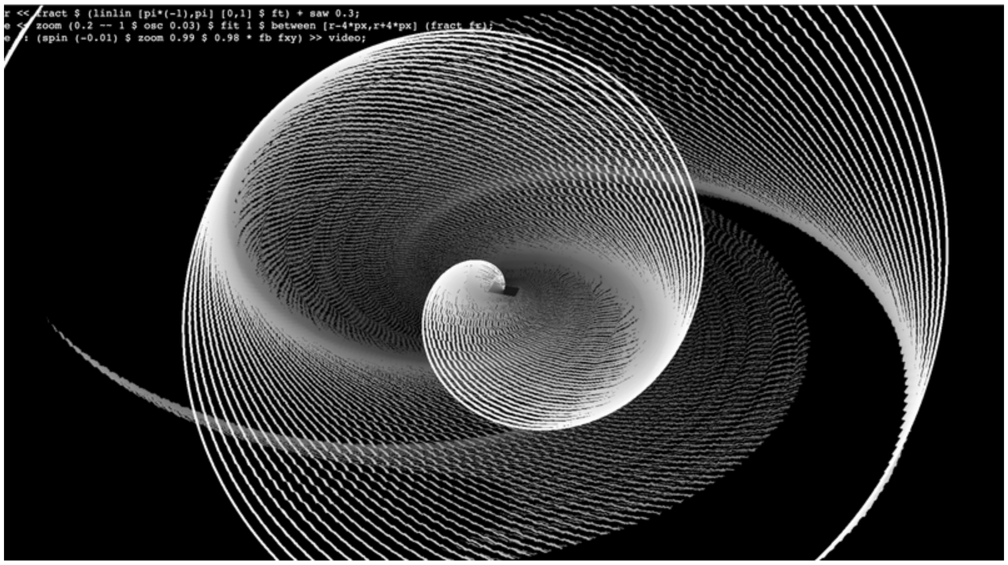
```
r << fract $ (linlin [pi*(-1),pi] [0,1] $ ft) + saw 0.3;
fit 1 $ between [r-4*px,r+4*px] (fract fr) >> video;
```

Figure 4

Next, we define `e` as this spiral, applying a zooming effect depending on an oscillator to make the pattern more dynamic.

Finally, we use feedback to create the ending pattern. For each frame, we take the previous one, slightly zoom it out, and rotate it to create the final result:

```
r << fract $ (linlin [pi*(-1),pi] [0,1] $ ft) + saw 0.3;
e << zoom (0.2 ~~ 1 $ osc 0.03) $ fit 1 $ between [r-4*px,r+4*px] (fract fr);
e +: (spin (-0.01) $ zoom 0.99 $ 0.98 * fb fxy) >> video;
```



```
r << fract $ (linlin [pi*(-1),pi] [0,1] $ ft) + saw 0.3;
e << zoom (0.2 ~~ 1 $ osc 0.03) $ fit 1 $ between [r-4*px,r+4*px] (fract fr);
e +: (spin (-0.01) $ zoom 0.99 $ 0.98 * fb fxy) >> video;
```

Figure 5

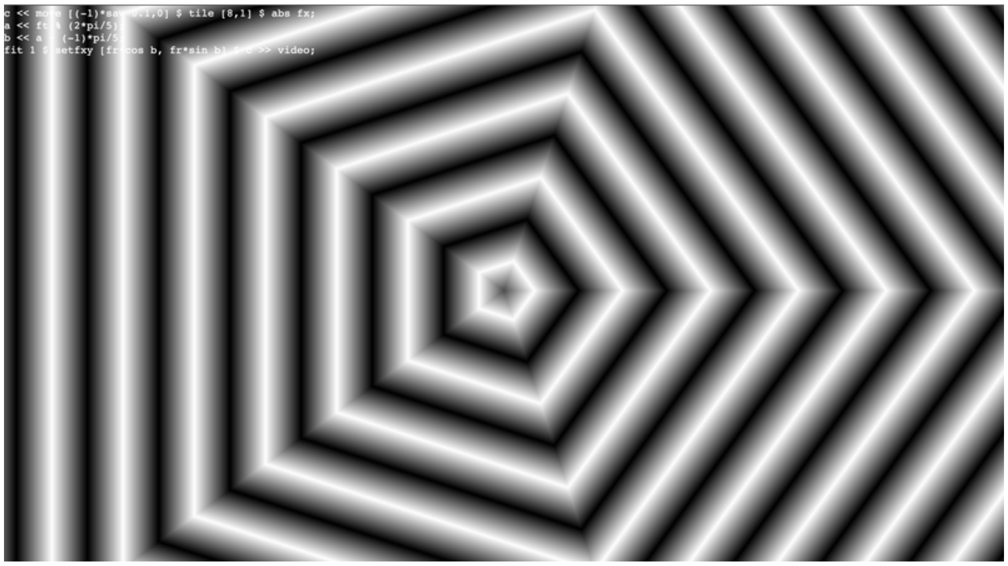
Using low-level Punctual functions to recreate Hydra functions

Hydra is arguably the best known live-coding language for visuals. It is possible to recreate some popular Hydra functions using Punctual capabilities.

In this example, `c` is an approximation to the `osc()` Hydra function, and the whole example is an approximation to `osc().kaleid(5).out()`.

Note how the `kaleid` part is achieved using a combination of mathematical operations and the amazing feature to remap coordinates with `setfxy` function:

```
c << move [(-1)*saw 0.1,0] $ tile [8,1] $ abs fx;
a << ft % (2*pi/5);
b << a + (-1)*pi/5;
fit 1 $ setfxy [fr*cos b, fr*sin b] $ c >> video;
```



```
c << move [(-1)*saw 0.1,0] $ tile [8,1] $ abs fx;
a << ft % (2*pi/5);
b << a + (-1)*pi/5;
fit 1 $ setfxy [fr*cos b, fr*sin b] $ c >> video;
```

Figure 6

Using audio reactive visuals to create patterns that change and adapt to music

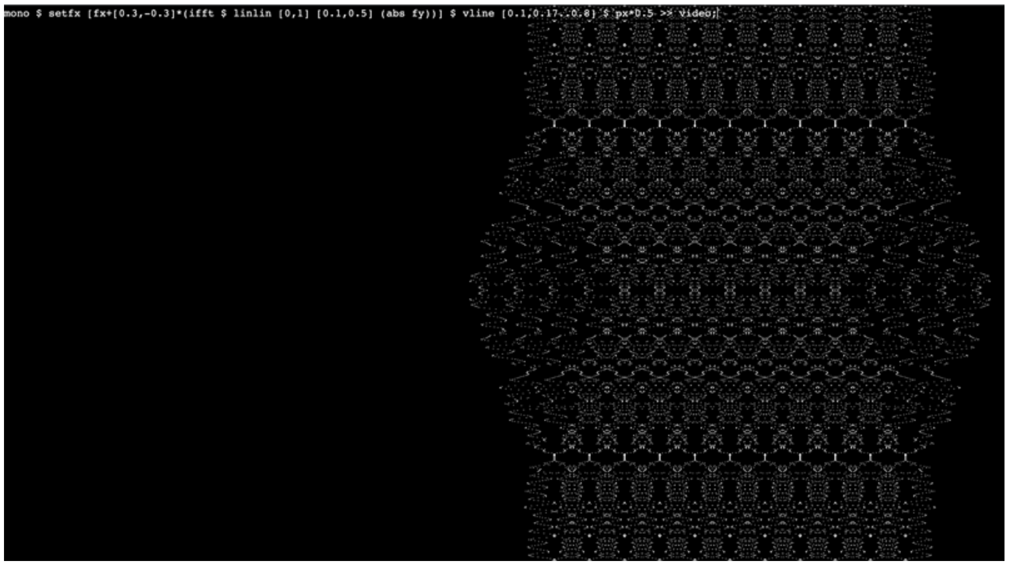
There are several ways to create audio-reactive visuals in Punctual: for instance, using oscillators whose frequency is synchronized with the music tempo, or using the intensity of predefined audio frequencies bands.

In this example, we use yet another way of achieving audio-reactive visuals by using the Fast Fourier Transform of the input sound (that is, the sound captured by a mic).

Here, we create some vertical lines using the Haskell shortcut for creating lists (`[0.1,0.17..0.8]`). This vertical lines are then modified several times. First, for each fragment, we change its x-coordinate according to an audio frequency that depends on the absolute value of the y-coordinate.

The linear rescale and other values are used to adjust the amount of deformation, `mono` is used to keep all the image white even though it has two channels, and the `[0.3,-0.3]` part doubles the transformation creating a left-right symmetry (this is the bit that creates two channels):

```
mono $ setfx [fx+[0.3,-0.3]*(ifft $ linlin [0,1] [0.1,0.5]
(abs fy))] $ vline [0.1,0.17..0.8] $ px*0.5 >> video;
```

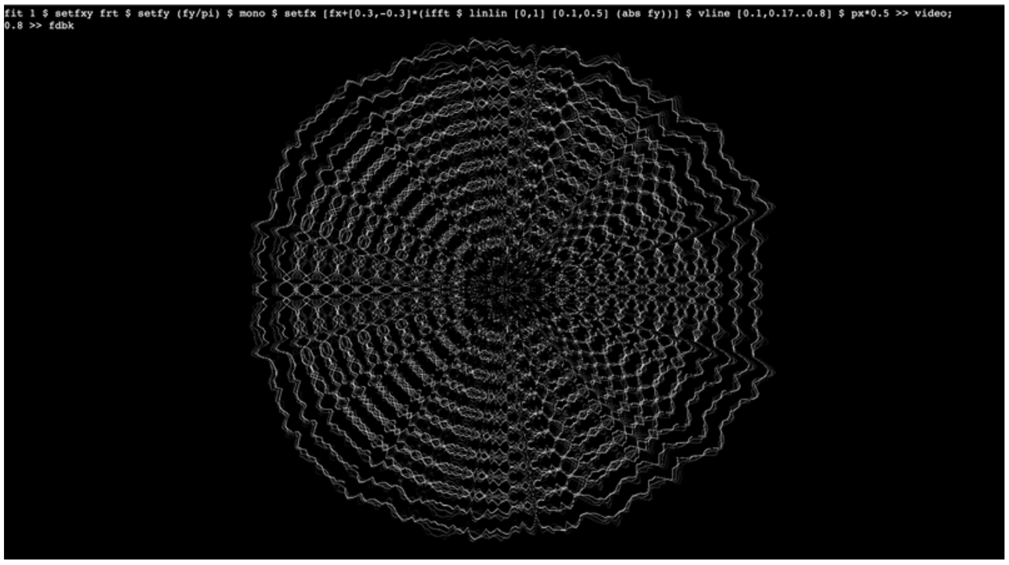


```
mono $ setfx [fx+[0.3,-0.3]*(ifft $ linlin [0,1] [0.1,0.5] (abs fy))] $ vline [0.1,0.17..0.8] $ px*0.5 >> video;
```

Figure 7

After that, we apply a second transformation that simply rescales the y-axis. This is to better distribute frequencies on the next step (try to remove the `setfy (fy/pi)` part to see the effect). Next, we apply another transformation that interprets x,y Cartesian coordinates as r,t in polar coordinates, converting the vertical lines into circles. `fit` is used to avoid getting ovals and not circles due to the aspect ratio. Finally, a bit of feedback is added to enhance the result:

```
fit 1 $ setfxy frt $ setfy (fy/pi) $ mono $
setfx [fx+[0.3,-0.3]*(ifft $ linlin [0,1] [0.1,0.5] (abs fy))] $
vline [0.1,0.17..0.8] $ px*0.5 >> video;
0.8 >> fdbk
```



```
fit 1 $ setfxy frt $ setfy (fy/pi) $ mono $ setfx [fx+[0.3,-0.3]*(ifft $ linlin [0,1] [0.1,0.5] (abs fy))] $ vline [0.1,0.17..0.8] $ px*0.5 >> video;
0.8 >> fdbk
```

Figure 8

About this guide

I decided to write this guide after a year of participating on a weekly jam at the Estuary platform, and using Punctual both in these jams and in several live performances.

Punctual is a somewhat low-level live-coding language, and has a very brief official documentation. While learning it, I always missed some more explanations and examples on how to use the distinct features the language provides.

Many of the examples presented in the guide are the result of conversations in the Discord's Estuary server with **David Ogborn** (Punctual's author, who is extremely helpful and always answers my questions), and **Bernard Gray** (who introduced me to the weekly jams and has been my partner in this journey).

The *complete guide to live-coding visuals in Punctual* is a work in progress and contributions are welcome. I expect this guide will be far more evolved by the time the Salon takes place.

I'm an IT teacher with more than 20 years of experience, and I have written a lot of documentation and tutorials for my students as well as many contributions in official product documentations, for example TidalCycles.

This project is not part of a formal study on patterns or live-coding (hence the lack of references and formality in some areas), and I'm not linked to any university or formal study group. I'm a live-coding enthusiast and practitioner, and this guide is one of my contributions to the live-coding ecosystem in particular and free software in general, to give something back to the community that so many wonderful things, both technological and personal, has generously given to me.

Conclusion

In this document, I've presented for the first time the *complete guide to live-coding visuals in Punctual*, a documentation effort which aims at creating both a full reference to the visuals aspect of the Punctual live-coding language, and a compilation of creative ideas to help anyone interested in using it.

Also, I proposed the realization of a workshop in Punctual during the Algorithmic Patterns Salon using the content of this guide, and focusing on the great capabilities that Punctual has to quickly create interesting and evolving visual patterns.