Creative Coding - Week 4

Livecoding Visuals with Hydra

Lecturer: Dr Michael Cook (mike.cook@kcl.ac.uk)

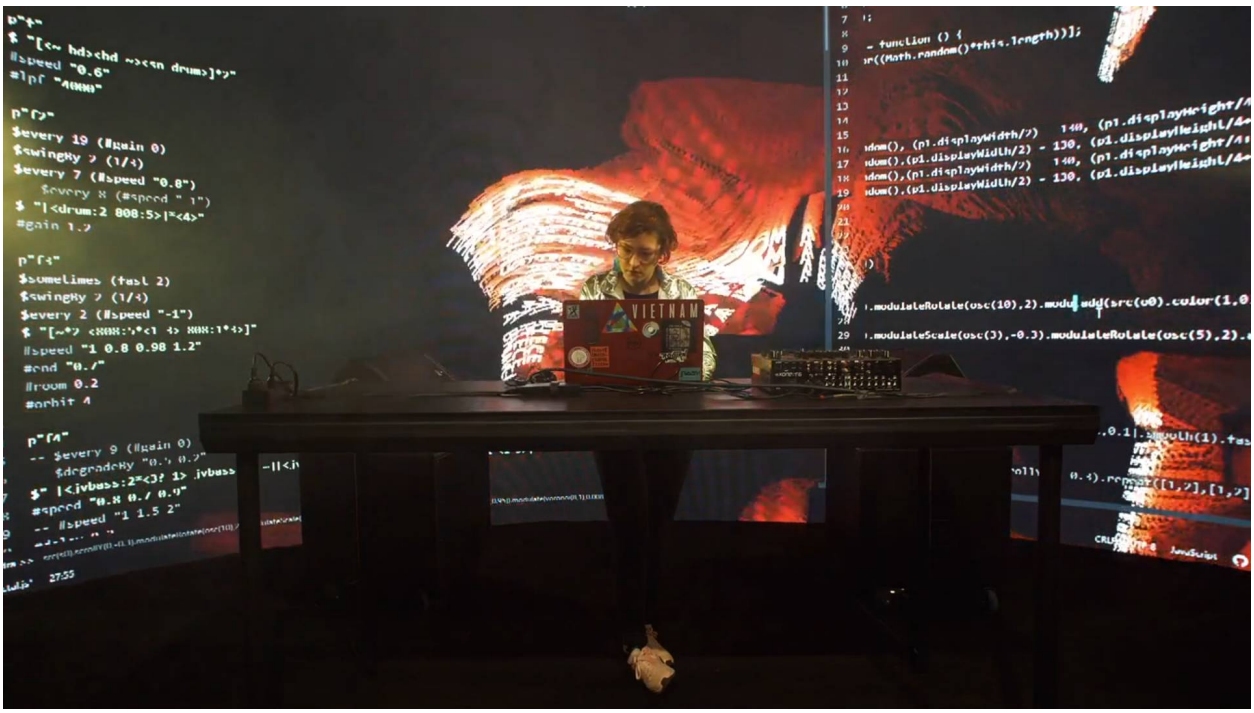


Figure 1: Florencia Alonso, a livecoder, performing generated music and visuals at the same time. The visuals, coded on the right-hand side, are written in Hydra. Photo via: Piksel Kidz

This week we're continuing our look at livecoding by learning a visual livecoding language called Hydra. Like Strudel, Hydra can be used for offline creative coding, but it excels at working in a live environment and people often use Hydra as a visual accompaniment to generated music (as seen in the image above). Hydra can be connected to other inputs and outputs, including p5.js and Strudel, which we'll get a taste of later on.

Hydra also represents a different paradigm for creative production than the ones we've seen so far. Previously, we mostly wrote code as a series of imperative *instructions*, and we imagine our output as being built up one block at a time - either a series of brushstrokes, or a series of musical notes that are played or drawn one after another. When we code in Hydra, we mainly think in terms of *information streams*. Data comes from a source, and we can shape and redirect it to other places before displaying it on the screen. You can find Hydra here:

https://hydra.ojack.xyz/

1 Basic Hydra

When you open Hydra, it will load a random example program, run it, and pop up a window. Feel free to take your time reading it, but when you're done close the window, and either hit the trash can symbol in the top-right, or click on the screen, select all the text, and delete it. Now we're

ready to write our first program!

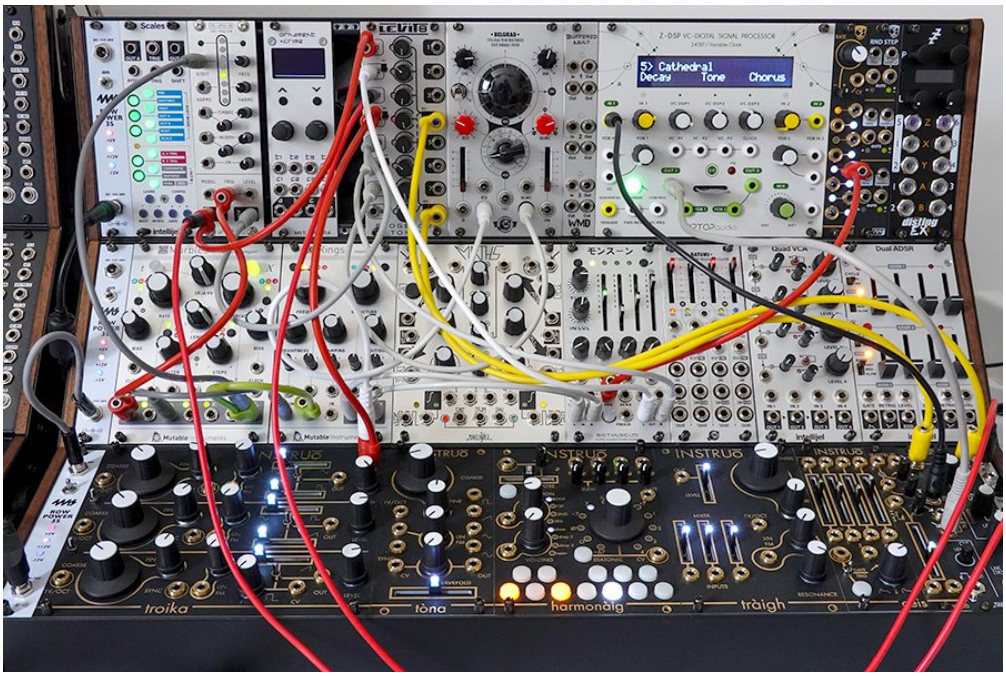


Figure 2: A modular synthesiser.

Hydra programs work a little like modular synthesisers. Modular synths can produce a variety of simple signals, like a continuous tone of noise. By connecting that tone to different filters, envelopes or other modules, we can transform that tone in different ways, for example making its volume vary, its pitch wobble, or adding noise to the signal. Let's write a very basic Hydra program to see what we mean by this.

```
1 osc().out()
```

Once you've put this code into your Hydra window, press the play button in the top-right, or press CTRL + SHIFT + ENTER. You should see a scrolling series of black and white bars.

Warning: as with p5js, it is possible to create strobing or flashing effects accidentally with Hydra. Please be careful if you are sensitive to these effects, and close your browser tab if you feel unwell.

Hydra uses a *very* compact syntax which makes it very hard to guess what a lot of its code does, so let's break this simple line down. First, `osc()` is short for oscillator. It produces a signal that oscillates between high and low values. `out()` then takes this source, and sends it to the default output (our monitor). Then we see the oscillating signal as black and white bands, fading through grey between them.

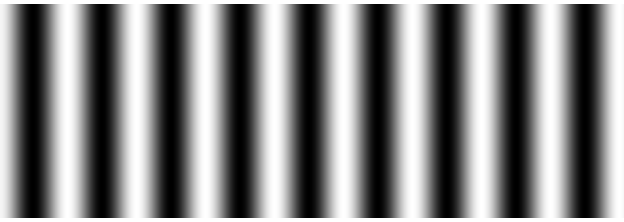


Figure 3: The output of an oscillator in Hydra.

1.1 Modifying Sources

`osc()` produces a *source*, but before we output it we can do other things to it. Try this:

```
osc()
.rotate()
.out()
```

From now on, we'll write each part of our Hydra code as a separate line, as I think it makes it easier to think about each stage of the process. This extra command rotates the source before sending it to the output.

Many of these functions can be given arguments too. For example:

```
osc(20, 0.1, 2)
.rotate(1.55)
.out()
```

The arguments to `osc()` sets the *frequency* of the signal – a higher number means more bars – the *speed* at which the oscillator runs, and the final number affects a colour mapping that's applied to it. The argument to `rotate()` tells us how much to rotate by, measured in radians. Try picking some of the following functions and add them to your chain, just make sure they appear before `out()` - again **please be aware that this may produce strobing/flashing effects without warning**, however I have tried to pick functions with a very low chance of doing so:

```
.kaleid()
.colorama(2) //recommend putting a number of some kind in
.pixelate()
.thresh()
```

1.2 Multiple Sources

Simply outputting Hydra sources is really cool, but sources can also be used as inputs, too. First, let's run this program. This introduces two new types of source: `voronoi` which is a sort of mosaic-like collection of shapes[1] and `noise` which is a 2D noise pattern similar to Simplex or Perlin noise.

```
voronoi(25, 1).out(o0)
osc(30, 0.2, 2).out(o1)
noise(15, 1).out(o2)

render()
```

[1] It's a Voronoi diagram, which is a way of cutting up a surface into a number of polygons based on a bunch of dots, which act as the center of each polygon. It produces a lot of cool effects and is used a lot in generative art. More here.

When you run this, you should see your screen split into four, with three patterns showing and one blank. This is because Hydra has four distinct output channels, `o0`, `o1`, `o2` and `o3`. For each of the sources above, we've piped them to a different output channel. By default, Hydra will show us `o0` but `render()` makes it display all four, like this.

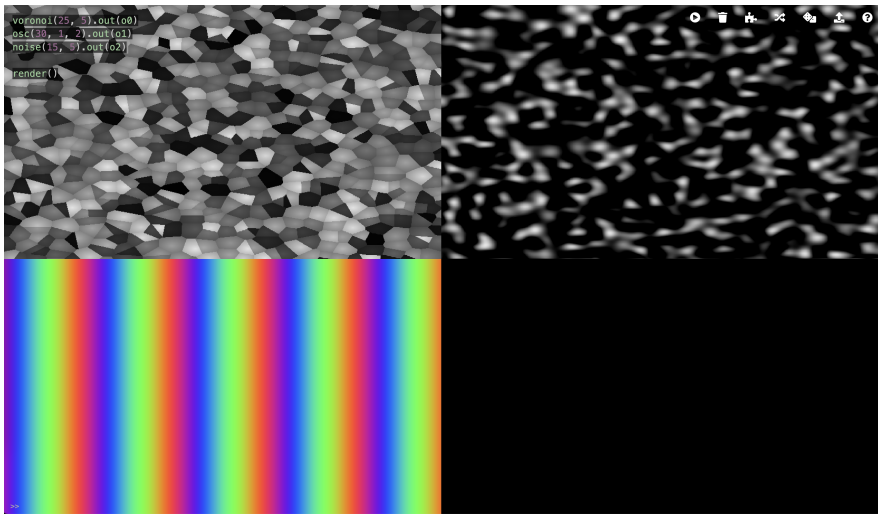


Figure 4: Hydra's four different outputs, all producing different things.

This is useful for all kinds of things, like we can work on multiple visual things at once and switch between them when performing. But we can also connect them up to each other. Let's add one more thing to the program:

```
voronoi(25, 1).out(o0)
osc(30, 0.2, 2).out(o1)
noise(15, 1).out(o2)

src(o1)
  .mult(o2)
  .out(o3)

render()
```

This new bit does three things. First, it creates a new source out of whatever is currently showing in `o1`. This is our oscillator. It then *multiplies* the source by `o2`. What does this mean? Well, if we look at `o2` we can see it's an image full of pixels, and the pixels vary in intensity, from black (zero intensity, i.e. 0) to white (full intensity, in this case 1). Hydra takes each pixel from `o1` and multiplies it with the corresponding pixel in `o2`. This means that where a pixel in one image is black, it multiplies by zero, cancelling the colour from the other image out. Where the pixels are white or colourful, they result in white or colourful pixels in the result. Finally, we output this new channel to `o3`.

What you should now see, in the bottom-right corner, is a new channel that looks like a combination of the noise field and the oscillator. It sort of looks like we've used the noise field as a cookie cutter to cut out bits of the oscillator? There are many different ways of blending two textures together. Try replacing `mult` with some of these commands, or add them in before or after:

```
.add(-something-)
.sub(-something-)
.diff(-something-)
```

```
.blend(-something-)
.mask(-something-)
```

The `-something-` here can be replaced with any source, including things like `osc()` or `voronoi()`, as well as other output channels.

Hydra has a lot of good documentation with interactive examples that also show you how to use images and webcams as inputs, and also to use modulating filters to change outputs in interesting ways. Modulations work a lot like these blending operations do above, in that they take a layer and then do something to it using another layer. For example, to return to our example again:

```
voronoi(25, 5).out(o0)
osc(30, 0.2, 2).out(o1)
noise(15, 1).out(o2)

src(o2).modulateRepeat(o1).out(o3)

render()
```

`repeat` is an existing command in Hydra, but it applies uniformly to the whole output. `modulateRepeat` allows us to apply repetition to the image in different amounts, based on the data or signal in the source we give it. What you see when you run this is the noise field repeated across the screen, but the amount it is repeated it dependent on the oscillator, which gives us these weird banding effects.

If this all seems a bit much – that's perfectly normal! This is hard to visualise, because you have to get used to thinking about mathematical transformations and changes over time. The best way to get to grips with it is just to dive in and experiment. Press the 'shuffle' button in the top right to view a random sketch, or change things at random and see what happens.

2 Challenge 1: Hydra Solo

Spend some time getting to grips with Hydra and see what you can make! Sometimes even random changes can produce surprising effects. In the top-right corner is a dice icon, if you click it Hydra will randomly change the numbers in your sketch and rerun it (there is no way to undo this, beware!)

If you're struggling to get started, hit the 'shuffle' button in the top right (two arrows crossed over each other) to get a random community-made sketch, and start editing/deleting things!

For a more specific challenge, here are four things I made with Hydra. How close can you get to remaking them? Each one is just a few lines of code[2].

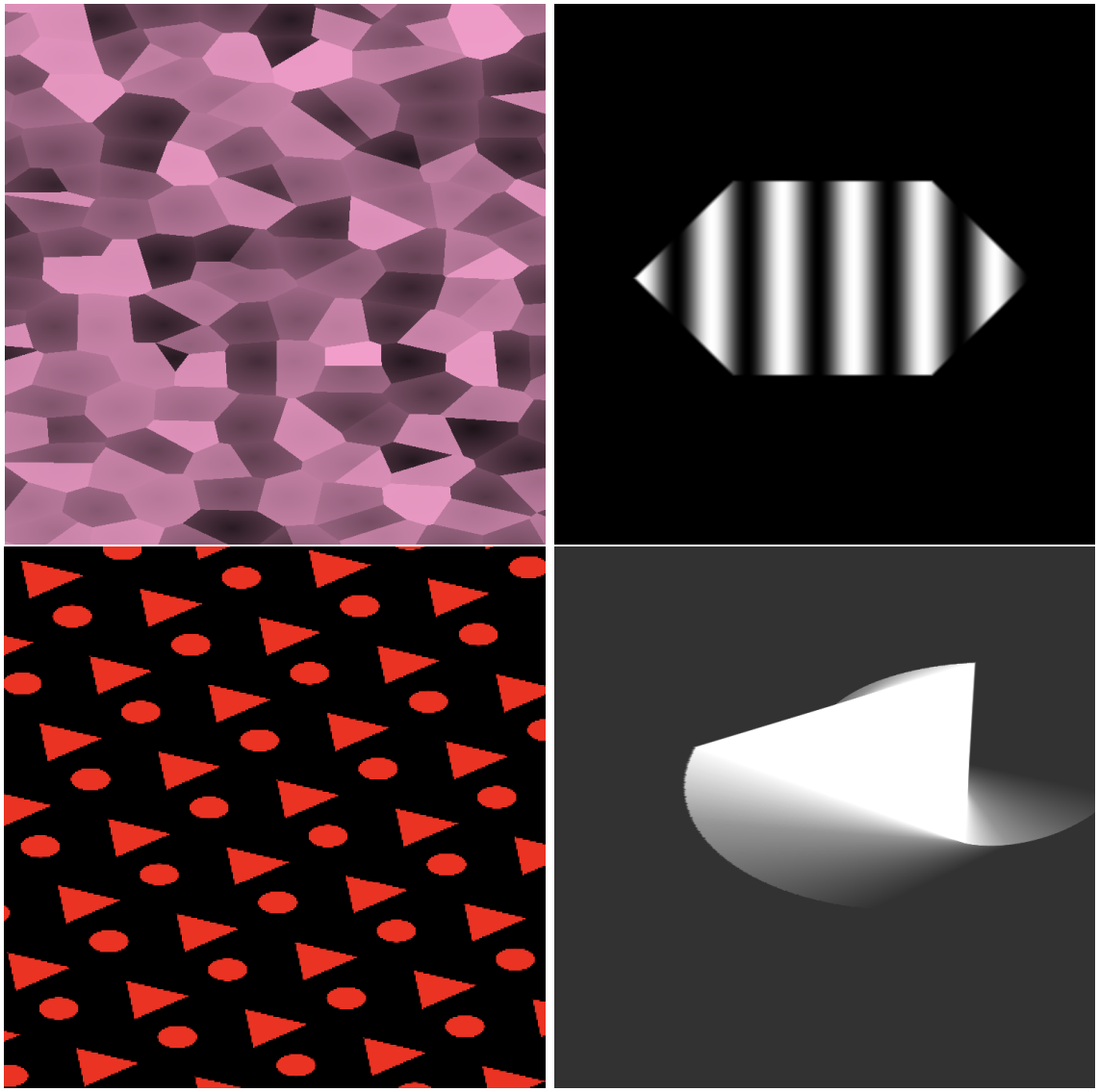


Figure 5: Some Hydra examples for you to try and recreate...

[2]The last one is a bit of a trick - it's moving. Tip/hint: it's drawing a rotating triangle, but doing something weird with it.

3 Hydra in Strudel

Strudel can support Hydra with just one line of code, which means we can write both visual and audio code in the same window (although it runs a little slower on Strudel). If you have a saved work from last week, load it back up in Strudel. If not, find one of the community Strudel songs, or you can use one I prepared here:

Sample Track

Add the following to the top of the Strudel code:

```
await initHydra({feedStrudel:1})
```

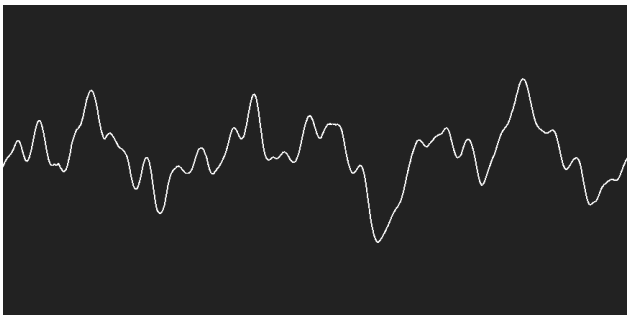
That's it! We're now ready to write Hydra code. This is a line of code that tells javascript to initialise hydra, and because it requires loading in external libraries we do it in the background, hence the `await` call, which means the instruction will be carried out asynchronously. Now you can add the following into your Strudel code, and hit play (or press CTRL + ENTER or CMD + ENTER to play the song as normal). You should hear the music start, and then see the Hydra visuals.

```
voronoi(40, 0.5, 0)
.out()
```

Writing Hydra next to Strudel is nice, but it's not *that* different to just using Hydra and Strudel in separate windows. What we'd really like to do is connect the two together. There's a few ways we can do this.

Approach 1: Listen to Strudel

Hydra can't listen to Strudel directly, but what it can do is have Strudel's own visualisations piped into it. Strudel can perform lots of different visualisations, including a 'scope' that shows particular frequencies in the output sound signal. Normally it might look like this:



We can turn this into a hydra source using the following code, which I nabbed from the Strudel docs:

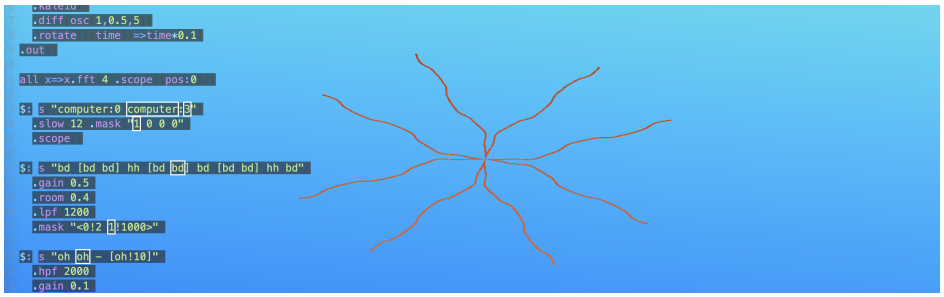
```
all(x=>x.fft(4).scope({pos:0}))
```

This pipes in the waveform above. Now we can write a little pattern that uses it:

```
src(s0)
  .kaleid()
```

```
  .diff(osc(1,0.5,5))
  .rotate(({time})=>time*0.1)
.out()
```

Working through the lines one by one: line 1 creates a new source out of the input from Strudel. This is just like calling `osc()` or `voronoi()` in hydra. Line 2 applies the Kaleidoscope filter to the pattern, so it warps around the screen. Line 3 applies some colour. Line 4 rotates the pattern slowly over time. Line 5 outputs it to the screen. Run it, and you should see some lines dancing around the screen looking kinda cool.



Approach 2: Shared Patterns

We can write patterns which Hydra executes in sync with Strudel. This makes it possible to precisely synchronise visual effects with the progression of your songs. I'll put a complete example here so you can see what I mean.

```
await initHydra()

let pattern = "<1 2 4 8>"

shape(100).repeat(H(pattern), 1).out()

s("<sd>").fast(pattern)
```

Note how Line 5 is Hydra, but Line 7 is Strudel. Strudel uses the pattern like it would normally - in this case, each cycle it passes the next number in the pattern to `fast()`. To use the pattern in Hydra, we pass it to the `H()` function. We can use this almost anywhere, and it will always pull the same value that Strudel is currently using.

We can use `H()`-patterns anywhere we like, of course, they don't need to be shared:

```
await initHydra()

shape(100).repeat(H("5 10"), H("10, 5"), 1).out()

s("sd")
```

Approach 3: Microphones

If you're livecoding you might not be able to access someone else's audio, or you might want to use another source anyway. You can always pull in audio from the microphone by passing `detectAudio:true` to the hydra startup line:

```
await initHydra({{detectAudio:true}})
```

This will ask permission for the mic, and then you can use it by accessing the `a.fft` object, which should be connected to the audio input. Values will be between 0 and 1, so remember to multiply them out to a useful range.

```
await initHydra({{detectAudio:true}})

voronoi(()=>(40 + a.fft[0]*10)).out()

s("[sd sd sd sd]")
```

Try playing this and then clapping your hands!

4 Challenge 2: Hydra Duet

For the rest of today's session, try making a little visualisation for whatever song in Strudel you've chosen to look at. You can use my song, your song, or one of the examples in the Patterns tab.

Alternatively (or if you're practicing at home) you can do what I do and put a DJ set on in the background and practice livecoding visuals while listening to the music. It's a bit harder, but it's good practice for adapting to something live!

5 What Next

5.1 Share your work with me!

I'd love to see (and hear) what you make, and I'd also love to be able to share people's work at future department events, or show to students in future years for inspiration! I would love it if you shared what you make - anonymously if you prefer, or with your name for full credit.

Ways you can share your work:

- Email the code, or a link to your Strudel/Hydra project to (mike.cook@kcl.ac.uk)
- Post it to GitHub and share with me (username: gamesbyangelina)

- Drop it anonymously in this form: https://forms.gle/qh8UWnqW1fNJyfxz8.

Strudel sharing works just like last time, even if you write Hydra in as well. Sharing direct from the main Hydra site is a bit different. When you run Hydra programs, it should automatically update the URL to contain a unique code that links to the code you wrote. In theory, sharing the URL with me should share your code exactly.

5.2 Further Reading

London has algoraves on regularly, which you can attend usually very cheaply (or for free) and see livecoders perform – I can almost guarantee you'll see a Strudel performer, or at least a TidalCycles performer, at any algorave. A popular place in London for them is Corsica Studios. Most algoraves have an open mic session at the start for an hour where *anyone* can plug in a laptop and perform for a few minutes. It's a great way to try out livecoding!

Some people also use Strudel/TidalCycles for making recorded/static songs, or record their live work and put it online. Graham Dunning is a nice example of this, on Bandcamp here.

There are also a lot of guides and performances recorded on YouTube. Watching someone make music live is really useful for seeing what techniques they use. I went to a workshop run by Lucy Cheesman (Heavy Lifting) who said that for most of her performances she uses maybe ten different effects and a few drum kits and instruments. You don't need to know everything about music, or everything about Strudel, to make great sounds - just get comfy with a little palette and see what you can make with it!