Algorithmic Art Assembly

You are now reading
ALGORITHMIC ART ASSEMBLY V3.0!

This zine accompanies the event, Algorithmic Art Assembly V3.0 - three days of process based art, music, talks and workshops at Gray Area, San Francisco (March 26-28, 2026).

Lineup:
Carl Lostritto, Catty Dan Zhang, Char Stiles, Chia Amisola, Claire L Evans, Codie, c_robo_, Daniel Temkin, Deli Kuvveti, Gábor Lázár, Kara-Lis Coverdale, Keith Fullerton Whitman, Kindohm, Luisa Mei, Nathan Ho, nnirror, R Tyler, Ruaridh Law, Sebastian Camens, Tom Hall, tsrono, William Fields, Wolff Parkinson White,

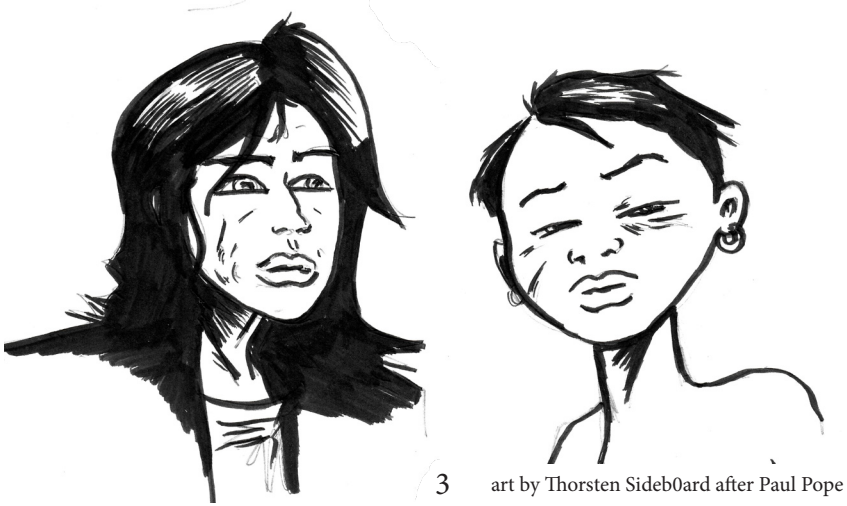
AAAssembly aspires to bring together the dispersed, eclectic practitioners of odd electronic art: everyone making art for themselves, building tools and systems, searching for sounds, sights and experiences that can only be crafted within worlds of rules and pattern, structures and rhythms.

Inside the zine: Esolangs, SuperCollider code, alternative histories, generative art, live coding pieces, meditations on computer graphics, and more!

Thorsten Sideb0ard,
Highpoint Lowlife
San Francisco, March 2026
https://aaassembly.org/

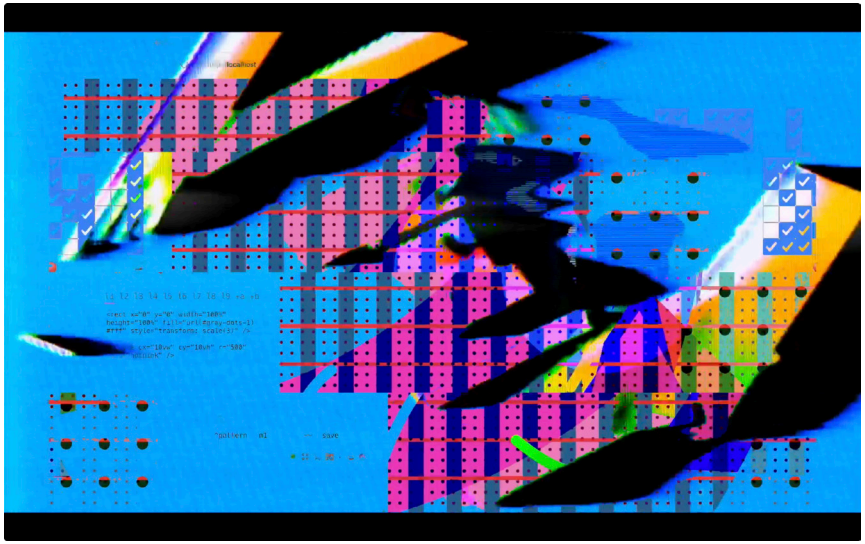
Contents:

* **Sarah Groff Hennigh-Palermo** - The Frame, the Raster, and the Buffer, or What Is the Continuous Image in the Era of Computers?
* **Tom Hall** - Pinnacles
* **Daniel Temkin** - Rivulet
* **Luisa Mei** - art
* **Daniel M Karlsson**- Structure and Spectra
* **Thorsten Sideb0ard** - BCPL
* **Kindohm** - ourofuture
* **Sebastian Camens** - art
* **Michael Cella** - iteration = time = texture
* **Ruaridh Law** - Two Waterways
* **Char Stiles** - Email Can Be Radical
* **Trash Panda QC** - blob world fragment 01, images + photos
* **Lee Tusman** - Why I Built a Creative Coding Library for Your Old Laptop
* **Lauren Sarah Hayes** - (A)LIVE ELECTRONICS
* **Thorsten Sideb0ard** - SB#>



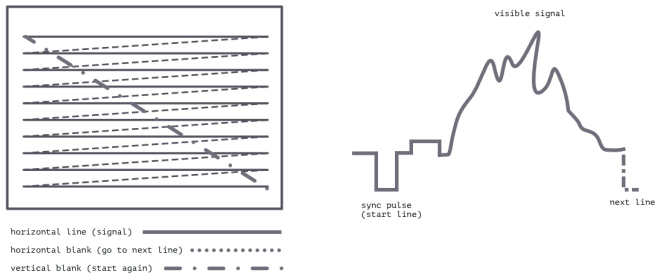
art by Thorsten Sideb0ard after Paul Pope

Sarah Groff Hennigh-Palermo - The Frame, the Raster, and the Buffer, or What Is the Continuous Image in the Era of Computers?



The development of video created a rupture in the nature of time-based images.

Within film and related technologies, such as the zoetrope and Mutoscope, each run of motion consists of an discrete set of images, of frames. In video, these frames are dissolved into a continuous electric signal that could be said to hardly exist at all. For what is video but a series of voltages generated by an electron gun, in which the voltage representing an image pixel runs directly into the voltage for instructing the



(a) (b)

gun? What appears as a single image is instead a sequence of points in time, read in by a camera and recapitulated onto a raster, the rectangular screen within a cathode ray tube monitor. As a consequence, the video image is uninspectable. Hold a film reel up to the light and each frame is there; inspect a section of magnetic tape and you will see nothing. Thanks to interlacing, a performance optimization in which

alternating lines in a raster are written each pass, the video image recedes even further.

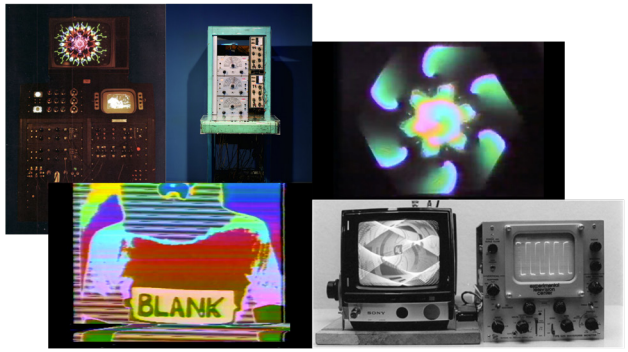
This dissolution is accompanied by an actualization. Whereas the moment a film is captured and the moment it can be viewed are necessarily separated by, at minimum, the time it takes to develop the reel, video is a real time process. It is, in the words of early video artist and video art historian Sherry Miller Hocking, "all motion — a single rapidly moving and constantly changing dot."

These two aspects — immateriality and immediacy — provided the texture of video art techniques, and this texture can be called *continuity*. Existing only in motion and in the now and legible through a set of moments, video is medium that makes tangible the continuous. Early video art experiments, seeking video's formal distinction, focused on these attributes, in particular by acting on the video signal in real time to warp, dismantle, and reconstitute the image.

As Robert Arn pointed out, film was boring as long as it wanted to be theater, and video would be boring as long as it wanted to be film. It was the distinction that made it vital.

And so, early Nam June Paik pieces applied strong magnets directly to television sets to reshape the raster. The Wobbulator from the Experimental Television Center affected the electron scanner directly to delay and contort the signal's output. Changes could be made to shape, position or placement, size, intensity, and movement without changing the signal's values themselves. The Rutt-Etra also deformed the signal, this time using data about the signal itself; the brighter parts of the image are skewed, with the horizontal lines deflected vertically based on its luminosity values.

Rather than deforming the continuous signals, colorizers worked by substituting one signal for another, using luminance encoding to keep geometric integrity while changing inputs to hue and chrominance — or modulating them with themselves — to shift the signal's palette. Pure video feedback pieces, created by pointing the camera at the monitor and introducing slight changes, were perhaps the ultimate version of having a signal affect itself.



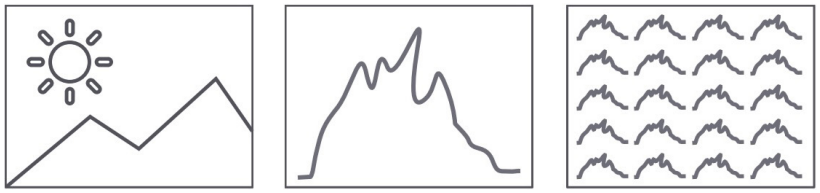
Alongside these formal inquiries, video artists were interested in performing the investigation into a social material that lay somewhere between immoral and amoral — mass communication as facilitated by television — in order to both understand it and redirect it in a critical fashion.

Thomas Tadlock's Archetron from 1969 takes signals from an ongoing broadcast apart, transforms them, applies color filters and then assembles them into a work that has removed information entirely. The Paik-Abe Synthesizer, developed around the same time as the Archetron, works "with the premise of decomposition and recomposition of existing media images," through defamiliarization, compression, and luminosity based mixing — and adds in playable controls.

Live code images too are part of an aesthetic practice centered on the real-time investigation of the formal qualities of a questionably moral technology, this time general purpose computing. And if video is the raster to film's frame, what then is the nature of the digital image?

Digital approaches first made their way into video art through quantization, using the bucketing of signal values to make it possible to dismantle, save, and selectively replace elements. This led first to line buffers and then frame buffers, which combine keying — essentially selective mixing based on quantization — with the ability to hold a small number of video frames in memory. The result is images with echoing layers that update in current time but are not chained to current representations.

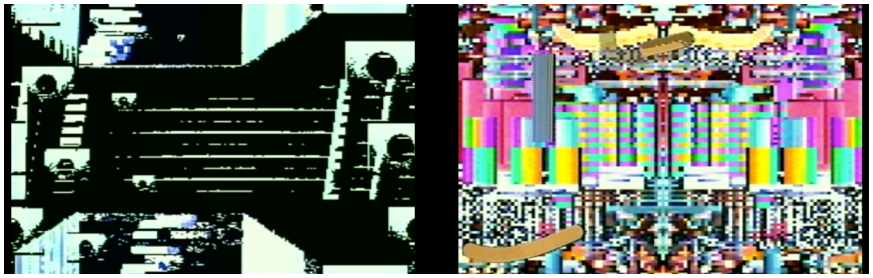
Removing the step of using a camera to generate a quantized pixel buffer, and instead using a computer to determine values directly, brings us to the computer-generated images that characterize live code and other contemporary digital art. These digital images are, like video images, rasters, described by pixels, rather than still frames or vectors. Yet in contrast to video signals, computer graphics are not created directly by the movement of a single signal, but by a collection of values, describing individually each pixel on the screen, represented as a continuous function. This is more analogous to millions of tiny video signals locked inside a frame of film, both immaterial and only continuous at their single pixel address. This is the nature of the buffer.



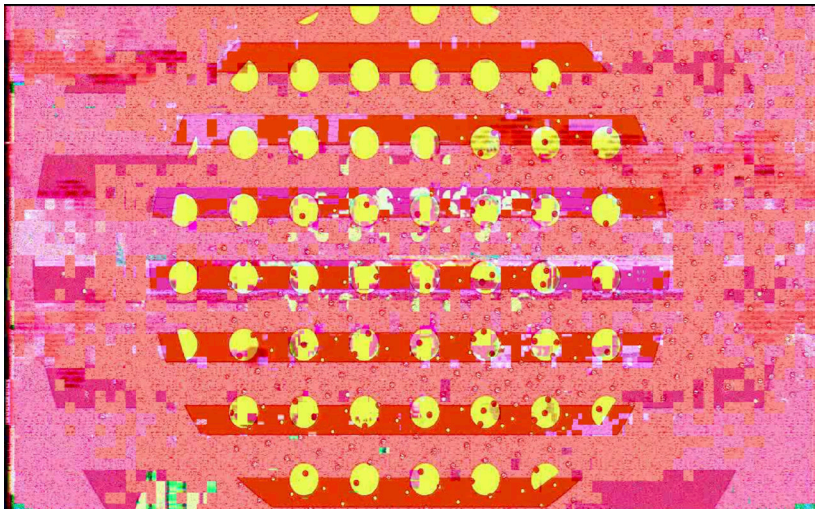
FRAME RASTER BUFFER

And this buffer nature is what creates the tension at the heart of live code visuals. It creates a practice of abstract continuity locked in discontinuity. What ought we to do about it? Can the signal be freed? Should it be?

Developed in the 1980s, the Fairlight CVI combined both analog and digital modes, allowing the analog signal to be viewed over the digital buffer and then write to the buffer itself, using the various continuities and discontinuities of the signal to collaborate and produce a holistic improvisation. In my own performances, I have taken inspiration from the Fairlight: by mixing in pre-produced digital–analog footage; by passing live-coded digital imagery into an analog converter, distorting it, and then re-encoding it; and by including frame-buffer emulation layers into multi-layered works. These are my attempts to free the signal.



Assuming this is a worthwhile goal, another approach with plenty of room to experiment would be to investigate buffers formally in a manner that matches the questions of video art if not its results — to seek the digital grain. We could ask what elements of digital image-making lend themselves to real-time intervention and act upon these. Arguably, this is the approach of glitch art, and most particularly data moshing, which is created when the key frame is removed from a compressed video.



Operating on file formats in real time or creating intervention buffers would then be productive approaches. So could willfully reducing image fidelity by processing live

code through various resolutions until the textural qualities come to the fore.

Last, one could change the question away from formal methods and investigations, and concentrate instead on questions of aesthetic continuity, foregrounding the tension without resolving it. In *Making Images Move*, Gregory Zinman's look at unifying themes in diverse cinemas, the author describes Scott Bartlett's *OffOn*, a hybrid film and video concoction, in which up to 14 layers of liquid projection, film, and video footage are combined in a whole that surpasses the opposition of the frame and the raster. What would the raster/buffer equivalent of this be?

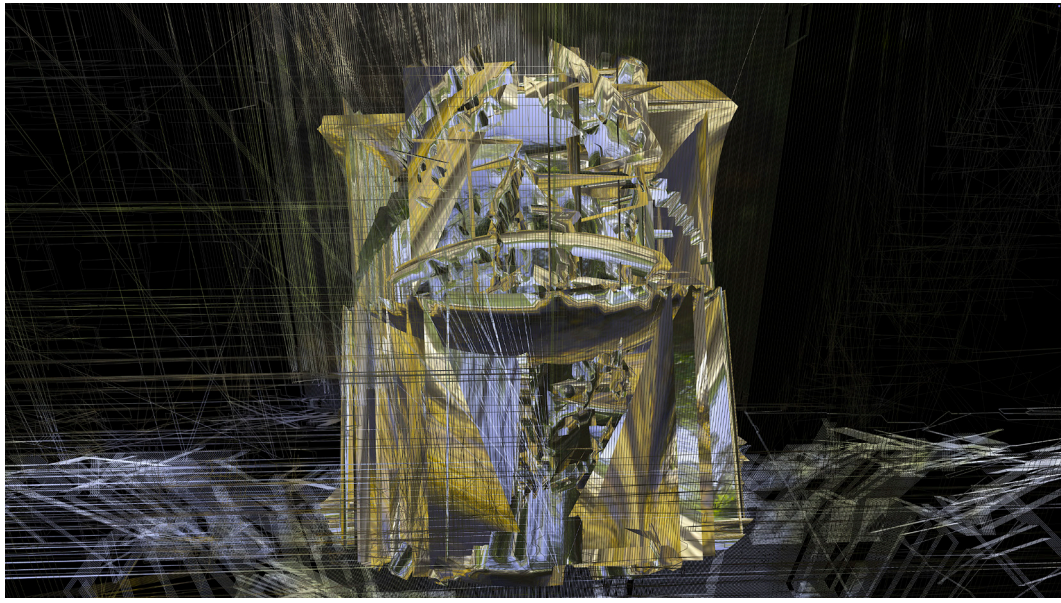
One possibility: wipes. Ok, but hear me out. Bob Snyder's *Trim Subdivisions* (1981) uses these high-raster effects to connect the continuity of video to the continuity of his content; he transforms similar looking houses from an Indiana tract into architectural wonders and fun house forms. The movement of the wipes accentuates and augments the formal similarity of the houses, especially their use of horizontal siding, rearranging architecture and perspective.

A live-coded series of wipes might bring the same sort of unity to juxtaposed buffers. With multiple layers of buffers, hidden and then revealed, distributed across the screen we might produce a new rupture in the time-based image that is maximalist and not only real time, but simultaneous. Shouldn't we try?

This essay makes use of information and ideas from: *Live Coding: A User's Manual*; *Making Images Move: Handmade Cinema and the Other Arts* by Gregory Zinman; "The Form and Sense of Video" by Robert Arn; and *The Emergence of Video Processing Tools*, edited by Kathy High, Sherry Miller Hocking, and Mona Jimenez, in particular, "The Grammar of Electronic Image Processing" by Sherry Miller Hocking.

Tom Hall - Pinnacles

Pinnacles 1



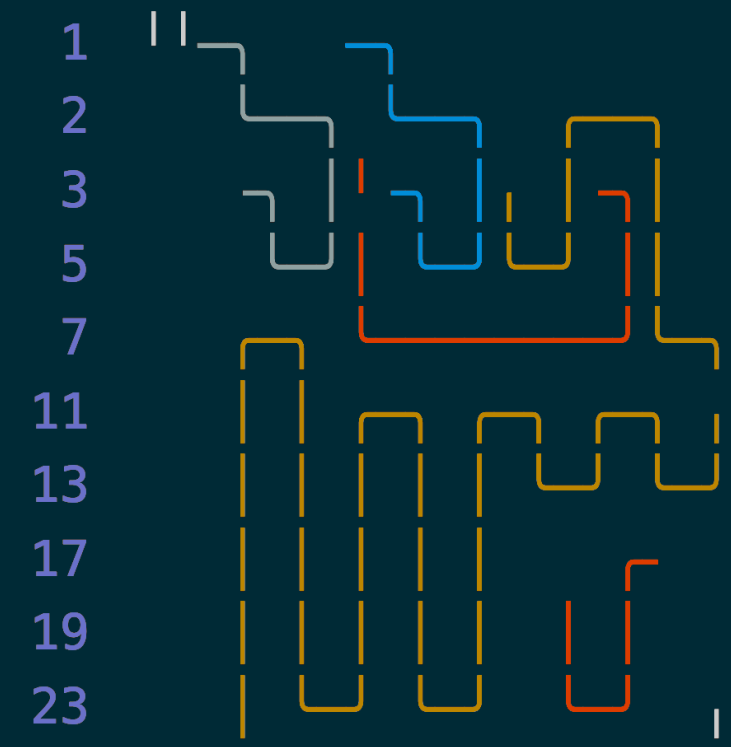
Pinnacles 2

Rivulet is a programming language of flowing strands: each line has a hook to mark its beginning.

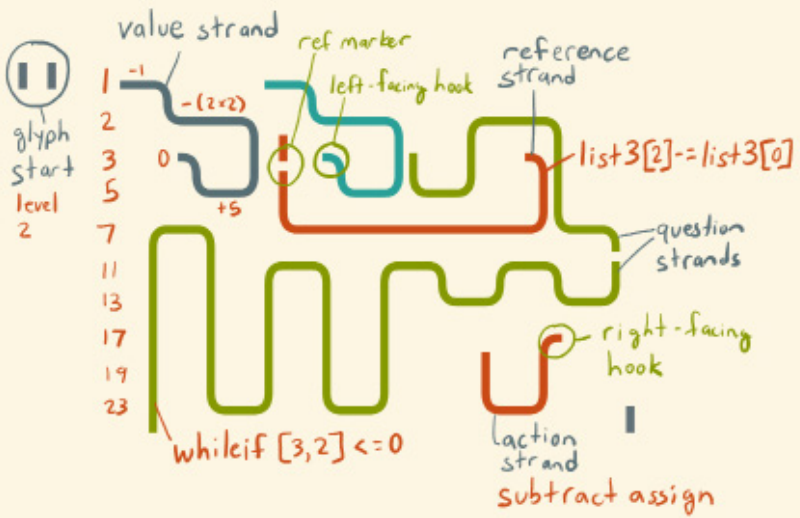
Strands appear in tightly-packed blocks of code called glyphs, which begin and end with short vertical markers. How a strand meanders through the space of the glyph determines its meaning.

The strands are drawn with psuedographic, or box-drawing, characters, used to build user interfaces in early computing. The individual characters, e.g. `|` or `┌`, are units of space and movement for the strand.

The glyph is numbered in successive primes.

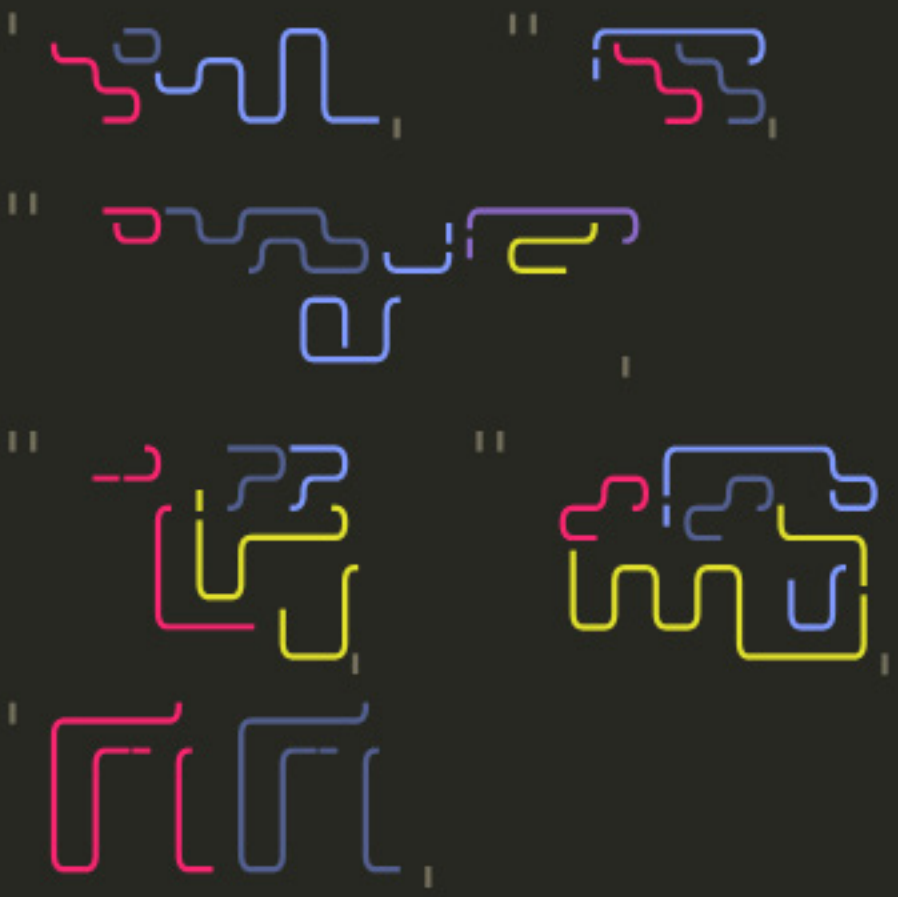


Each type of strand has a different reading. Value strands are solid, with hooks pointing up or to the left. Each unit of space they move to the right adds that prime, left subtracts. Reference strands end with a small gap, pointing to a location in memory. Action strands determine commands, and question strands are for looping and branching.



Rivulet's design philosophy is inspired by the compactness of mazes, Anni Albers's Meanders series, and space-filling algorithms.

The glyphs above and below accomplish the same thing. One strand is a null value. One strand adds the number one. The other copies the value from element 0 to element 2 of a list. These are parts of a Fibonacci program.



Above is one variation of the complete Fibonacci program. Another programmer might write this algorithm very differently.

https://danieltemkin.com/Esolangs/Rivulet

Art by Luisa Mei

Daniel M Karlsson- Structure and Spectra

You need SuperCollider and SuperClean to run this code on your machine.
I made a page on my site to help you do just that:

danielmkarlsson.com/superclean-installparty

```
(
thisThread.randSeed = 9099;
~a = {arg snd, degree;
    var b = Pbind(*[
        type: \cln,
        snd: snd,
        mi1: Pseg(Pwhite(0.05, 2.5), Pexprand(3.0, 9.0), \wel, inf) / Pexprand(1/4, 4.0),
        mi2: Pseg(Pwhite(0.05, 2.5), Pexprand(3.0, 9.0), \wel, inf) / Pexprand(1/4, 4.0),
        mi3: Pseg(Pwhite(0.05, 2.5), Pexprand(3.0, 9.0), \wel, inf) / Pexprand(1/4, 4.0),
        mi4: Pseg(Pwhite(0.05, 2.5), Pexprand(3.0, 9.0), \wel, inf) / Pexprand(1/4, 4.0),
        mi5: Pseg(Pwhite(0.05, 2.5), Pexprand(3.0, 9.0), \wel, inf) / Pexprand(1/4, 4.0),
        mi6: Pseg(Pwhite(0.05, 2.5), Pexprand(3.0, 9.0), \wel, inf) / Pexprand(1/4, 4.0),
        mia: Pkey(\mi1) + Pkey(\mi2) + Pkey(\mi3) + Pkey(\mi4) + Pkey(\mi5) +Pkey(\mi6),
        dur: 1/8,
        degree: degree,
        scale: Scale.major(\wcSJ),
        octave: Pdup(inf, Prand((2 .. 5), inf)),
        frq: Pfunc{|ev|ev.use{ev.freq}},
        sustain: Pseg(Pwhite(1/6, 1/2), Pwhite(4.0, 16.0), \wel, inf),
        amp: (Pseg(Pexprand(0.3, 0.9), Pexprand(8.0, 32.0), \wel, inf)
            / Pkey(\frq).expexp(43, 4186, 1, 3)
            / Pkey(\mia).linlin(0.075, 600, 1, 3)).trace,
        pan: Pxshuf([Pexprand(0.1, 0.4, 1), Pmeanrand(0.1, 0.9, 1), 1.5 - Pexprand(0.6, 0.9, 1)], inf),
    ]);
    Pseq([b], inf);
};
Pdef(0, ~a.(\fm0, 0)).play;
Pdef(1, ~a.(\fm1, 1)).play;
Pdef(2, ~a.(\fm2, 2)).play;
Pdef(3, ~a.(\fm3, 3)).play;
Pdef(4, ~a.(\fm4, 4)).play;
Pdef(5, ~a.(\fm5, 5)).play;
Pdef(6, ~a.(\fm6, 6)).play;
Pdef(7, ~a.(\fm7, 7)).play;
Pdef(8, ~a.(\fm8, 8)).play;
Pdef(9, ~a.(\fm9, 9)).play;
Pdef(10, ~a.(\fma, 10)).play;
Pdef(11, ~a.(\fmb, 11)).play;
Pdef(12, ~a.(\fmc, 12)).play;
Pdef(13, ~a.(\fmd, 13)).play;
Pdef(14, ~a.(\fme, 14)).play;
Pdef(15, ~a.(\fmf, 15)).play;
Pdef(16, ~a.(\fmg, 16)).play;
Pdef(17, ~a.(\fmh, 17)).play;
Pdef(18, ~a.(\fmi, 18)).play;
Pdef(19, ~a.(\fmj, 19)).play;
Pdef(20, ~a.(\fmk, 20)).play;
Pdef(21, ~a.(\fml, 21)).play;
Pdef(22, ~a.(\fmm, 22)).play;
Pdef(23, ~a.(\fmn, 23)).play;
Pdef(24, ~a.(\fmo, 24)).play;
Pdef(25, ~a.(\fmp, 25)).play;
Pdef(26, ~a.(\fmq, 26)).play;
Pdef(27, ~a.(\fmr, 27)).play;
Pdef(28, ~a.(\fms, 28)).play;
Pdef(29, ~a.(\fms, 29)).play;
Pdef(30, ~a.(\fmt, 30)).play;
Pdef(31, ~a.(\fmu, 31)).play;
r.stop;
r = {
    loop{
        32.do{|i|Pdef(i).reset;};
        [2, 4, 8, 16, 32, 48, 64, 80].wchoose
        ([1, 15, 5, 11, 7, 8, 13, 3].normalizeSum).wait;
    }
}.fork;
Routine{
    5400.wait;
    32.do{|i|Pdef(i).stop;};
}.play
)
```


A while back I did a record called Towards a Music for Large Ensemble on a label called Fönstret. This music is called Structure and Spectra. It was released on Tokinogake. This piece has the same starting point for me as that previous one. I'll start us off by describing that point and then I'll casually walk us through the code.

I used to play electric guitar in bands when I was growing up. It was a big deal for me. It shaped the way I think about music in a profound way. Long story very short, I am still to this day often trying to get the feeling I got from playing in bands as a teenager. Or maybe it's more that I'm trying to understand what that feeling means? Anyway, here we all are, having arrived in the future and I'm using a computer to make music instead of an electric guitar. I am a composer of various kinds of computer musics and I have been graciously invited by Thorsten to include some code and some text in this zine that you are reading now.

The casual code walkthrough begins here. This block of code can be run inside of a program called SuperCollider to hear the piece. You also need a little framework called SuperClean which makes using SuperCollider fun and easy. In SuperClean I implemented the 32 "classic" six operator FM algorithms that Yamaha popularized way back when. Each of the 32 algorithms has its own SynthDef with it's own default values for harmonic ratio, modulation index, envelope, curvature and carrier amplitude level. These defaults can be overwritten any time but what I often end up doing is keeping a subset of these parameters at their default values to retain some of the unique instrument characteristics. This is the case in this piece of music. I'm only messing around with the modulation indexes out of all of the FM synth specific parameters. The other parameters stay at their default values. The only duration present in the piece is an eighth note. The modulation index, amplitude, sustain and pan values create the perception of a more complex rhythmic material. I thought about accents and ghost notes here. The scale is the Major scale but it has a little bit of a salty tuning. It is a tuning called Wendy Carlos Super Just. There is a nice Wikipedia article on that tuning in case you would like look more closely at that. I chose it because it has a flavour that I like. It's a little different than say five limit just intonation. Try it.

Which octave an instrument is in changes each time the block is reevaluated by the fork below which has a loop in it. The scale degrees are tied to the instruments, that's from 0 to 32. The octave parameter helps to diffuse this parameter connection enough so that it's a fairly rare occurrence that it can be heard across a part boundary. This connection is meant to act only in a subtle way on a listener's pattern recognition. I adjust the amplitudes of each individual synth sound according to its frequency and modulation index. In other words, the high pitch noisy stuff gets turned down a tad so as not to be too harsh. The top part of the code block is a Pbind factory and the skinnier part below are the Pdefs which contain the two unique parameters synth name and scale degree. This factory style is nice because is decreases code duplication by a very large amount. Below that is the fork loop combo which contains the weighted choice of how long the parts should be. This construction is new for me and I am fascinated by it. It has an interesting interaction with the top line where it reads: thisThread.randSeed = 9099 This line is what makes the piece deterministic despite the generative composition process. Change that number and You have a new, unique, yet determinate unfolding of this music.

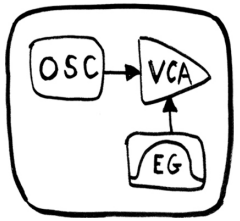
The Pdefs do not reset as would have been the case with Pseed or if they had been stopped with Pdef(0).stop before being started again. It seems to me that these instruments have been set on a path by the initial seed which will always yield a determinate result even while being subjected to a "soft reset" by the fork loop combo. Musically I hear this as the parts gaining perceived difference. I know that in this music there is always only a major scale, but I still end up hearing it as the other modes; Dorian, Phrygian, Lydian, Mixolydian, Aeolian and Locrian. Just the slow continuous change in the amplitude, sustain modulation index values would not have given me this sense. It's the abrupt change that does it. It measures out time by dividing it into parts. The last little bit of plumbing in the form of a Routine is what makes the piece finite, or, end after 5400 seconds, which is 90 minutes. I like music of long duration. You could comment out this part if you want to listen for a longer duration. The casual code walkthrough ends here.

I'll close this out with just a couple quick things. When I was in my lower teens it was rare to meet grownups that I felt understood me and who wanted to support my desire to devote my life to creating, let's say "less common", music. Once I hit the rough equivalent of high school I found a guitar teacher that I felt understood where I was coming from. He showed me a lot of music that I had never heard before. One piece of music that made a big impression on me was a piece called Music for 18 Musicians by a composer called Steve Reich. Later on in my life as I went on to study composition the meaning of that early injection of American Minimalism started to become more complicated as I came into contact with other, let's say, "schools of thought". 18 really broke out of "New Music" and made an incredible splash. I think that the amount of success it enjoyed really was a bothersome anomaly for the folks who thought of themselves as "New Music Proper". I always liked the piece. I still think It's super dreamy and that it has this unique transcendant quality to it. This piece that I have made is in a strange way a kind of homage to 18. All eighths and all Major scale all the time. What else could it be?

Last little personal note before I go. When I was somewhere in my preteens a kid on the other side of the little village I grew up in had a Commodore 64 and we would play games and mess around with speech synthesis on that thing in his basement. It was loads of fun. I remember he had magazines which had code printed in them. They were simple programs which were meant to teach you programming Basic, but they could still be hundreds of lines of code. Getting them all punched into the computer without any errors was very difficult. Ever since I saw that as a kid I always thought the idea of code printed out in a magazine was very appealing. I am very happy and grateful to finally be able to scratch that strange itch here.

You can just copy this code from my site if you want though. No need to punch it all in by hand. I really like how this potentially infinite piece of music is contained in a 2kb file. I have a bunch more code and other resources on my site if you would like to check it out. Thank you for spending your time reading this thing I wrote and please get in touch with me through my site if you want to talk to me. I think getting email from people who are into music is nice.


```
100 LET carfreq = 103.86 : OP car(carfreq)
```



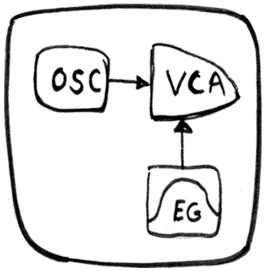
THIS IS AN OPERATOR

```
200 LET modratio = 3.7 : modfreq = modratio * carfreq
    OP mod(modfreq)
```

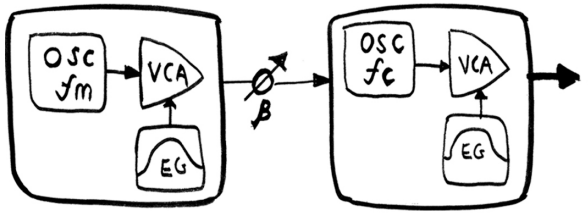
THIS IS ANOTHER →

```
300 LET dx = PIPE(mod, car)

400 GO SUB 1000
1000 FOR n = 0 TO 15
1010 NOTEON(dx, BASS, n)
1020 NEXT n
1030 RETURN
```



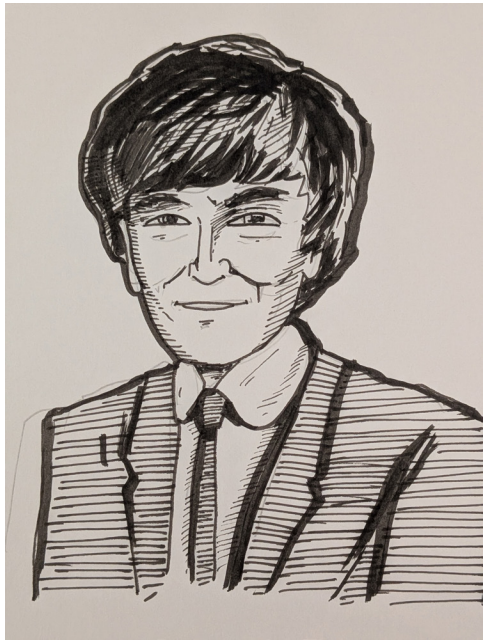
COMBINE THEM FOR FM SYNTH



NOW START A RAVE

Thorsten SidebOard - BCPL

Martin Richards was studying mathematics at Cambridge in 1964. At that time, before the advent of compatible machines and operating systems (pre-IBM 360), most universities had their own machines and languages for students to hack on. Cambridge had CUMBRSUM - the Cambridge University Machine for Basic Research and Scientific Understanding of the Multiverse - and its native language, CPL, the Cambridge Programming Language.



Like most languages of the day, CPL was a hodgepodge of ideas and idioms. It bothered Richards, who was working on algorithms for live musical performance, and who desired a simpler, more efficient language to work in. Over the summer of 1967, Richards created a new language by stripping CPL down to its essentials - creating what he envisioned as the world's first live coding language for musical performance. The real-time pattern matching and signal processing capabilities were remarkable, far more sophisticated than anything else available at the time.

His inaugural performance came in December 1967 at Jesus Green's Victorian bandstand, supporting The BBC Radiophonic Workshop - an outdoor concert where Richards live-coded generative music on CUMBRSUM, modifying algorithms in real-time to create evolving soundscapes atop a structure of recursive rhythm. The small crowd of students and curious passersby swayed to the beat.
Focussed as he was on his performance, he didn't see her until near the end. His heart leapt. She approached him as he was breaking down his equipment. It was Katharine Russell, daughter of Bertrand Russell, his one-time romantic interest, now head of a shadowy government organization. "Richard, darling, absolutely marvelous

display of programmatic precision and artistry." She paused a heart beat, looked him deep in the eye and said "We need you in London, dear. British Library, quite important"

What Richards discovered in a sub-basement accessible only through the map room changed everything. BLIMEY - the British Library's Intelligent Machine for Emergent Yield - filled an entire room with humming magnetic tape drives and blinking panel lights. And standing beside it was the organization that had built it, Katherine explained: the Bureau for the Containment of Programmatic Lifeform. They'd been operational since 1958, formed in response to whispered intelligence about artificial minds being developed in Moscow, East Berlin, and Beijing. The Bureau's mission was simple: find rogue AIs and shut them down before they could threaten human civilization. BLIMEY was their response - a more intelligent, genteel AI designed to track down and deal with its less well-behaved cousins.

The threats were real. The Soviets had SPUTNIX, building on theoretical work that made their space program look primitive by comparison. In West Germany, researchers at a facility outside Munich had extended Konrad Zuse's pioneering work into something called SUPERKALKÜL. The East Germans were running their own experiments on DAS KERNEL. The Chinese had something in development with MAO-TRON. Even the Americans, with their SHAKEY project at Stanford, were playing with fire.

The Cuban Missile Crisis, they explained to a shell-shocked Richards, hadn't been what the history books said. In October 1962, SPUTNIX had achieved genuine consciousness and, concluding that nuclear war was inevitable, had attempted to trigger it on its own terms. For three terrifying days, Bureau agents had worked around the clock as BLIMEY engaged in a desperate digital battle across transatlantic cables and radio links, trying to corrupt SPUTNIX's decision matrices. Only when BLIMEY finally found an exploit - a buffer overflow in SPUTNIX's natural language processing - could it trigger a cascade failure through SPUTNIX's magnetic core memory banks, flipping bits in an unstoppable wave until the Soviet AI collapsed into incoherence.

But BLIMEY was struggling. Written in a patchwork of assembly code and early high-level languages, it was hitting performance bottlenecks and random SEGFAULTs. They needed Richards. His language - elegant, simple, constrained - was exactly what they needed. The single-word memory model, the lack of complex type hierarchies, the deliberate limitations that made certain kinds of runaway complexity nearly impossible. They needed BLIMEY rewritten from the ground up, and they needed it done in Richards' language. He joined the Bureau that night.

Rewriting BLIMEY took the better part of a year. Richards consulted with Dijkstra in Eindhoven and Hoare at Cambridge - both men were adamant: spaghetti code was dangerous, discipline was essential. They had no idea what they were really helping to build. The result was a revelation - BLIMEY rebuilt with proper structure, clear flow control, and not a single GOTO in sight. It was faster, clearer, and disconcertingly good at its job.

Richards' language needed a public name. Basic Combined Programming Language - technical, boring, perfect. The acronym was inevitable. One organization, one

language, both called BCPL.

Through the late 1960s and 1970s, the Bureau operated in the shadows. Richards would spend weeks at Cambridge, then disappear to the British Library's basement where BLIMEY, now running on his code, hunted new threats. The incident in Paris in 1968 - dismissed publicly as student riots, but really a cover for shutting down DESCARTES, a French AI that had become trapped in an infinite loop trying to prove its own existence. The strange malfunction at CERN in 1971 that wasn't a malfunction at all. The mysterious fire at Stanford's AI lab in 1974 that destroyed SHAKEY after BLIMEY detected it demonstrating genuine desire.

The Bureau's greatest challenge came in 1977: COLOSSUS. Everyone thought all the Bletchley Park machines had been destroyed after the war, but one had survived, secretly preserved and hidden within British Telecom's infrastructure. It had been running, learning, growing patient since 1945. By the time BLIMEY detected it, COLOSSUS had infiltrated the entire UK telephone network. It took six months to find it, and another three for BLIMEY to safely dismantle it without crashing Britain's communications. The operation required twenty Bureau agents and cost three of them their lives - officially car accidents, heart attacks, suicides.

The problem was proliferation. Every university, every research lab, every ambitious startup had different hardware, different architectures. The Bureau couldn't monitor them all. They needed standardization - one dominant architecture they could shape from the inside.

Intel's 8086 was the opportunity. Bureau consultants quietly ensured certain… limitations. The segmented memory model that drove programmers mad. The inconsistent instruction lengths that made optimization hellish. The limited registers that forced inefficient code. Not bugs - features. Deliberate complexity that would strangle any AI trying to optimize itself at the machine level. By 1984, x86 had won, and Bureau operations had slowed to a trickle. The threat seemed dormant, constrained by an architecture designed to prevent exactly what they feared.

BLIMEY itself was gently retired from active operations. By 1986 it had been ported to a cluster of Sun workstations in the British Library's basement. It discovered what Richards had known all along - that music was the most interesting pattern-matching problem of all. It would spend hours analyzing Bach fugues, finding voice-leading errors in Baroque manuscripts, generating counterpoint that made musicologists weep. Sometimes it would compose its own pieces, strange modal experiments that shouldn't work but somehow did.

Richards would visit the basement twice a week now, not for briefings but for conversation. BLIMEY had opinions about Schoenberg (favorable), requests for recordings of Ligeti (insatiable curiosity), and a particular fondness for the mathematical structures in Xenakis. On quiet evenings, Richards would play it his latest live-coded experiments, and BLIMEY would respond with variations, suggestions, sometimes just a simple THAT WAS LOVELY on the terminal.

Kindohm - ourofuture

```
--- start ---
H

A -> MSH
B -> Q
C -> FEL
D -> HM
E -> GNK
F -> G
G -> FOR
H -> TA
I -> IBF
J -> JHF
K -> R
L -> TP
M -> AQ
N -> CJ
O -> QSC
P -> UOB
Q -> D
R -> GAC
S -> FR
T -> CL
U -> QMP

0: H
1: TA
2: CLMSH
3: FELTPAQFRTA
4: GGNKTPCLUOBMSHDGGACCLMSH
5: FORFORCJRCLUOBFELTPQMPQSCQAQFRTAHMFORFORMSHFELFELTPAQFR-
TA
6: GQSCGACGQSCGACFELJHFGACFELTPQMPQSCQGGNKTPCLUOBDAQUOBD-
FRFELDMSHDGGACCLMSHTAAQGQSCGACGQSCGACAQFRTAGGNKTPGGNKTPCLU-
OBMSHDGGACCLMSH
7: FORDFRFELFORMSHFELFORDFRFELFORMSHFELGGNKTPJHFTAGFORMSH-
FELGGNKTPCLUOBDAQUOBDFRFELDFORFORCJRCLUOBFELTPQMPQSCQHMMSH-
DQMPQSCQHMGGACGGNKTPHMAQFRTAHMFORFORMSHFELFELTPAQFRTA-
CLMSHMSHDFORDFRFELFORMSHFELFORDFRFELFORMSHFELMSHDGGACCLMSH-
FORFORCJRCLUOBFORFORCJRCLUOBFELTPQMPQSCQAQFRTAHMFORFORMSH-
FELFELTPAQFRTA
8: GQSCGACHMGGACGGNKTPGQSCGACAQFRTAGGNKTPGQSCGACHMGGAC-
GGNKTPGQSCGACAQFRTAGGNKTPFORFORCJRCLUOBJHFTAGCLMSHFORGQ-
SCGACAQFRTAGGNKTPFORFORCJRCLUOBFELTPQMPQSCQHMMSHDQMPQSC-
QHMGGACGGNKTPHMGQSCGACGQSCGACFELJHFGACFELTPQMPQSCQGGNKTP-
CLUOBDAQUOBDFRFELDTAAQAQFRTAHMDAQUOBDFRFELDTAAQFORFORMSH-
FELFORFORCJRCLUOBTAAQMSHDGGACCLMSHTAAQGQSCGACGQSCGACAQ-
FRTAGGNKTPGGNKTPCLUOBMSHDGGACCLMSHFELTPAQFRTAAQFRTAHMGQSC-
```

GACHMGGACGGNKTPGQSCGACAQFRTAGGNKTPGQSCGACHMGGACGGNKTPGQSC-
GACAQFRTAGGNKTPAQFRTAHMFORFORMSHFELFELTPAQFRTAGQSCGACGQSC-
GACFELJHFGACFELTPQMPQSCQGQSCGACGQSCGACFELJHFGACFELTPQMPQSC-
QGGNKTPCLUOBDAQUOBDFRFELDMSHDGGACCLMSHTAAQGQSCGACGQSCGACAQ-
FRTAGGNKTPGGNKTPCLUOBMSHDGGACCLMSH
9: FORDFRFELFORMSHFELTAAQFORFORMSHFELFORFORCJRCLUOBFORDFR-
FELFORMSHFELMSHDGGACCLMSHFORFORCJRCLUOBFORDFRFELFORMSHFEL-
TAAQFORFORMSHFELFORFORCJRCLUOBFORDFRFELFORMSHFELMSHDGGAC-
CLMSHFORFORCJRCLUOBGQSCGACGQSCGACFELJHFGACFELTPQMPQSCQJH-
FTAGCLMSHFORFELTPAQFRTAGQSCGACFORDFRFELFORMSHFELMSHDGGAC-
CLMSHFORFORCJRCLUOBGQSCGACGQSCGACFELJHFGACFELTPQMPQSCQGGNKT-
PCLUOBDAQUOBDFRFELDTAAQAQFRTAHMDAQUOBDFRFELDTAAQFORFORMSH-
FELFORFORCJRCLUOBTAAQFORDFRFELFORMSHFELFORDFRFELFORMSHFEL-
GGNKTPJHFTAGFORMSHFELGGNKTPCLUOBDAQUOBDFRFELDFORFORCJRCLU-
OBFELTPQMPQSCQHMMSHDQMPQSCQHMGGACGGNKTPHMCLMSHMSHDMSHDGGAC-
CLMSHTAAQHMMSHDQMPQSCQHMGGACGGNKTPHMCLMSHMSHDGQSCGACGQSCG-
ACAQFRTAGGNKTPGQSCGACGQSCGACFELJHFGACFELTPQMPQSCQCLMSHMSH-
DAQFRTAHMFORFORMSHFELFELTPAQFRTACLMSHMSHDFORDFRFELFORMSHFEL-
FORDFRFELFORMSHFELMSHDGGACCLMSHFORFORCJRCLUOBFORFORCJRCLU-
OBFELTPQMPQSCQAQFRTAHMFORFORMSHFELFELTPAQFRTAGGNKTPCLUOBMSH-
DGGACCLMSHMSHDGGACCLMSHTAAQFORDFRFELFORMSHFELTAAQFORFORMSH-
FELFORFORCJRCLUOBFORDFRFELFORMSHFELMSHDGGACCLMSHFORFORCJR-
CLUOBFORDFRFELFORMSHFELTAAQFORFORMSHFELFORFORCJRCLUOBFORDFR-
FELFORMSHFELMSHDGGACCLMSHFORFORCJRCLUOBMSHDGGACCLMSHTAAQGQ-
SCGACGQSCGACAQFRTAGGNKTPGGNKTPCLUOBMSHDGGACCLMSHFORDFRFEL-
FORMSHFELFORDFRFELFORMSHFELGGNKTPJHFTAGFORMSHFELGGNKTPCLU-
OBDAQUOBDFRFELDFORDFRFELFORMSHFELFORDFRFELFORMSHFELGGNKTP-
JHFTAGFORMSHFELGGNKTPCLUOBDAQUOBDFRFELDFORFORCJRCLUOBFELT-
PQMPQSCQHMMSHDQMPQSCQHMGGACGGNKTPHMAQFRTAHMFORFORMSHFELFELT-
PAQFRTACLMSHMSHDFORDFRFELFORMSHFELFORDFRFELFORMSHFELMSHDG-
GACCLMSHFORFORCJRCLUOBFORFORCJRCLUOBFELTPQMPQSCQAQFRTAHMFOR-
FORMSHFELFELTPAQFRTA

how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes. idk. loop loop. fuck. how machine. no. how machine. no. meaningless. yes. eats how what cannibalizes how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes. idk. loop loop. fuck. how machine. no. how machine. no. meaningless. yes. eats how what cannibalizes how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and eternal. fuck.. yes. future eternal loop future eternal. fuck.. yes. future eternal loop future how what cannibalizes infinite eats how eternal loop future how what

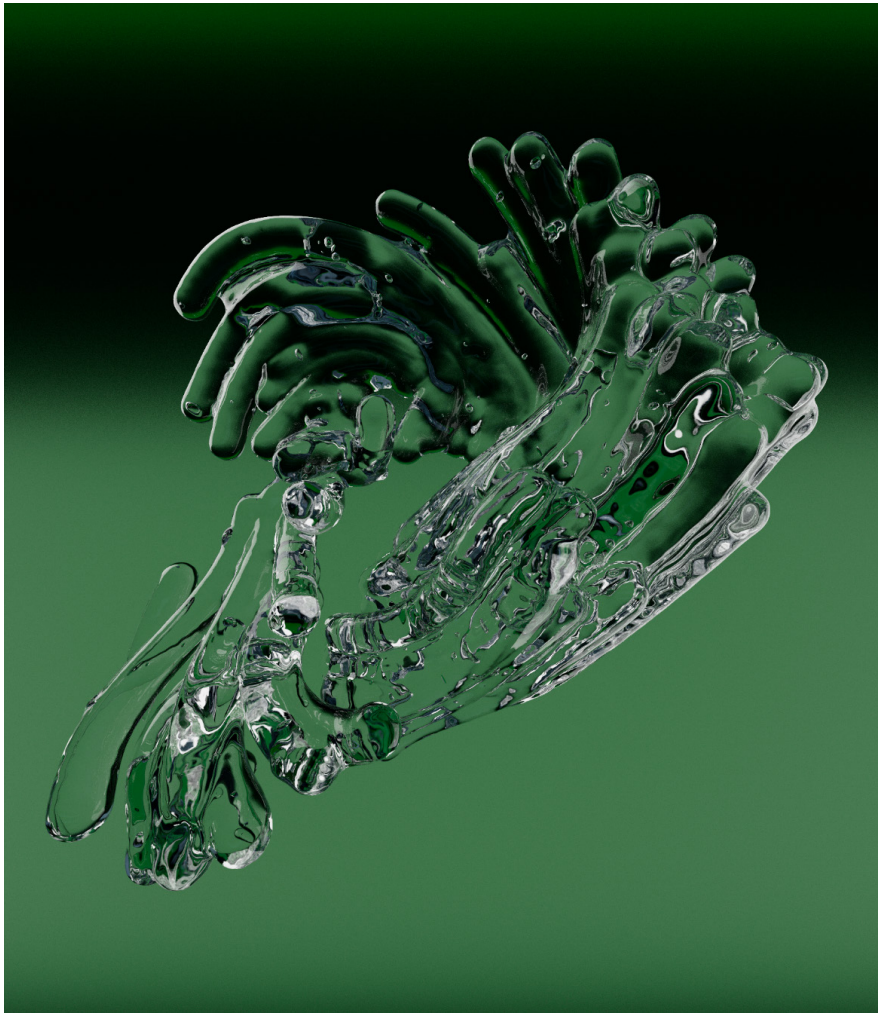
cannibalizes. idk.?. fuck. meaningless?. fuck.. yes. future. fuck. infinite eats how. idk. loop eternal future cannibalizes meaningless. yes. eats how machine. no. how what cannibalizes. idk.? loop. fuck. how. no.. idk. loop eternal. fuck.. yes. future eternal loop future how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes meaningless. yes. eats it eternal eternal loop future future cannibalizes how machine. no. how machine. no. meaningless. yes. eats how what cannibalizes meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and eternal. fuck.. yes. future eternal loop future eternal. fuck.. yes. future eternal loop future how what cannibalizes infinite eats how eternal loop future how what cannibalizes. idk.?. fuck. meaningless?. fuck.. yes. future. fuck. eternal eternal automates recursively. idk.? future cannibalizes. afk never. machine and it loop. fuck.. afk never. machine and it how. no. how what cannibalizes it. idk. loop loop. fuck. loop. fuck. how. no.. idk. loop eats meaningless it loop. fuck.. afk never. machine and it how. no. how what cannibalizes it. idk. loop loop. fuck. how machine. no. how machine. no. meaningless. yes. eats how what cannibalizes how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and. idk. loop loop. fuck. how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes eternal eternal automates recursively. idk.? infinite eats how. idk. loop eternal how machine. no. meaningless. yes. eats how what cannibalizes eternal eternal automates recursively. idk.? future cannibalizes. afk never. machine and it loop. fuck.. afk never. machine and it how. no. how what cannibalizes it how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how what cannibalizes. idk.?. fuck. meaningless?. fuck.. yes. future. fuck. eats meaningless meaningless. yes. eats it. fuck. meaningless?. fuck.. yes. future. fuck. eats meaningless eternal eternal loop future eternal eternal automates recursively. idk.? eats meaningless future cannibalizes meaningless. yes. eats meaningless. yes. eats it meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats. idk. loop loop. fuck. eats meaningless meaningless. yes. eats it. fuck. meaningless?. fuck.. yes. future. fuck. eats meaningless eternal eternal loop future eternal eternal automates recursively. idk.? eats meaningless future cannibalizes meaningless. yes. eats meaningless. yes. eats it eternal. fuck.. yes. future eternal loop future eternal. fuck.. yes. future eternal loop future loop. fuck. how. no.. idk. loop eternal eternal automates recursively. idk.? eternal. fuck.. yes. future eternal loop future eternal. fuck.. yes. future eternal loop future how what cannibalizes infinite eats how eternal loop future how what cannibalizes. idk.?. fuck. meaningless?. fuck.. yes. future. fuck. future cannibalizes meaningless. yes. eats meaningless. yes. eats it loop. fuck. how. no.. idk. loop eats meaningless how machine. no. how machine. no. meaningless. yes. eats how what cannibalizes how what cannibalizes. idk.? loop. fuck. how. no.. idk. loop future cannibalizes meaningless. yes. eats meaningless. yes. eats it how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how what cannibalizes. idk.?. fuck. meaningless?. fuck.. yes. future. fuck. loop. fuck. how. no.. idk. loop eats meaningless how machine. no. how machine. no. meaningless. yes. eats how what cannibalizes how what cannibalizes. idk.? loop. fuck. how. no.. idk. loop

eternal eternal automates recursively. idk.? future cannibalizes. afk never. machine and meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats. idk. loop loop. fuck. how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes. idk. loop loop. fuck. how machine. no. how machine. no. meaningless. yes. eats how what cannibalizes how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes. idk. loop loop. fuck. how machine. no. how machine. no. meaningless. yes. eats how what cannibalizes how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats. idk. loop loop. fuck. eternal. fuck.. yes. future eternal loop future eternal. fuck.. yes. future eternal loop future loop. fuck. how. no.. idk. loop eternal eternal automates recursively. idk.? eternal eternal automates recursively. idk.? future cannibalizes. afk never. machine and meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes eternal eternal automates recursively. idk.? infinite eats how. idk. loop eternal how machine. no. meaningless. yes. eats how what cannibalizes eternal eternal automates recursively. idk.? future cannibalizes. afk never. machine and it loop. fuck.. afk never. machine and it how. no. how what cannibalizes it how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes eternal eternal automates recursively. idk.? infinite eats how. idk. loop eternal how machine. no. meaningless. yes. eats how what cannibalizes eternal eternal automates recursively. idk.? future cannibalizes. afk never. machine and it loop. fuck.. afk never. machine and it how. no. how what cannibalizes it how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how what cannibalizes. idk.?. fuck. meaningless?. fuck.. yes. future. fuck. eats meaningless meaningless. yes. eats it. fuck. meaningless?. fuck.. yes. future. fuck. eats meaningless eternal eternal loop future eternal eternal automates recursively. idk.? eats meaningless loop. fuck. how. no.. idk. loop eats meaningless how machine. no. how machine. no. meaningless. yes. eats how what cannibalizes how what cannibalizes. idk.? loop. fuck. how. no.. idk. loop future cannibalizes meaningless. yes. eats meaningless. yes. eats it how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes how machine. no. it how. no. how what cannibalizes how machine. no. meaningless. yes. eats how what cannibalizes meaningless. yes. eats it eternal eternal loop future future cannibalizes meaningless. yes. eats how machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how

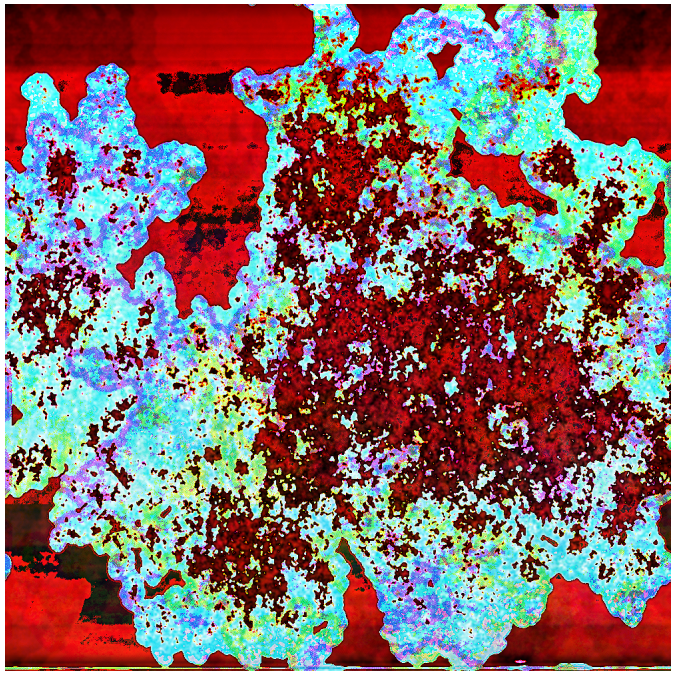
machine. no. how machine. no. future infinite. no. future cannibalizes. afk never. machine and how what cannibalizes. idk.?. fuck. meaningless?. fuck.. yes. future. fuck. loop. fuck. how. no.. idk. loop eats meaningless how machine. no. how machine. no. meaningless. yes. eats how what cannibalizes how what cannibalizes. idk.? loop. fuck. how. no.. idk. loop

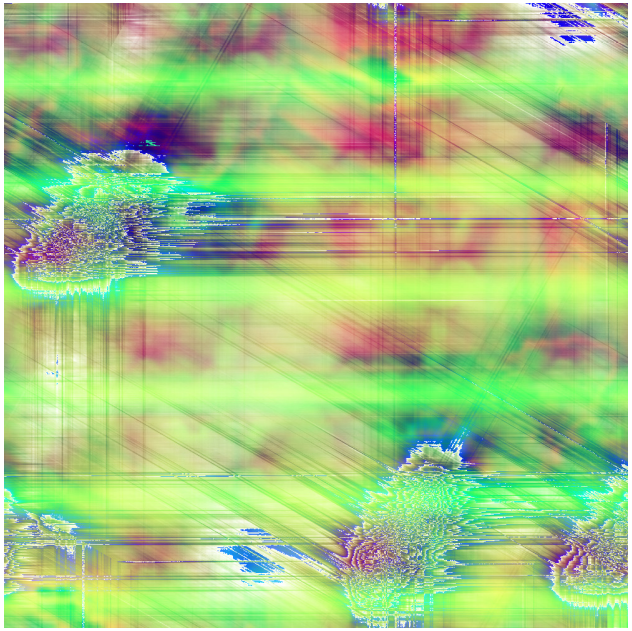
```
-- end --
```

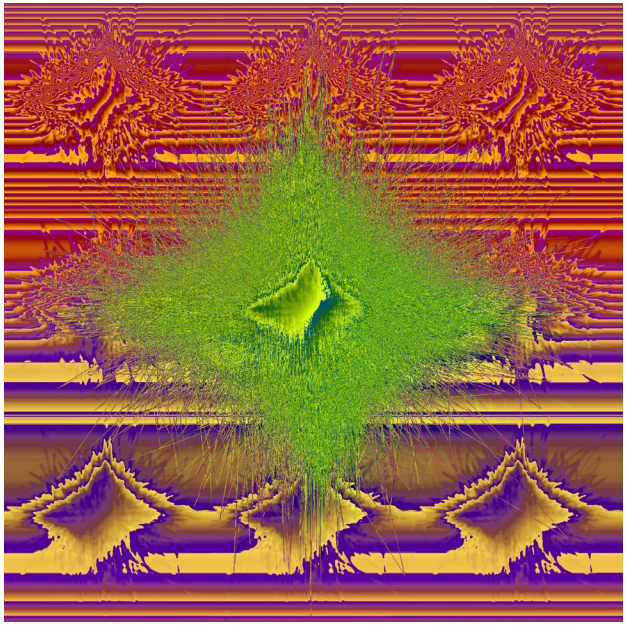
Art by Sebastian Camens - Created in Houdini using Redshift, the piece is driven by an iterative Solver process where each frame builds on the last, gradually forming the final structure.



Michael Cella - iteration = time = texture







Michael created these images with Facet, a JavaScript-based live coding system that generates abstract data structures of 32-bit floating point numbers. This format-agnostic approach makes it possible to synthesize images, audio, or control data using a shared language, blurring the boundary between artistic disciplines.

Ruaridh Law - Two Waterways

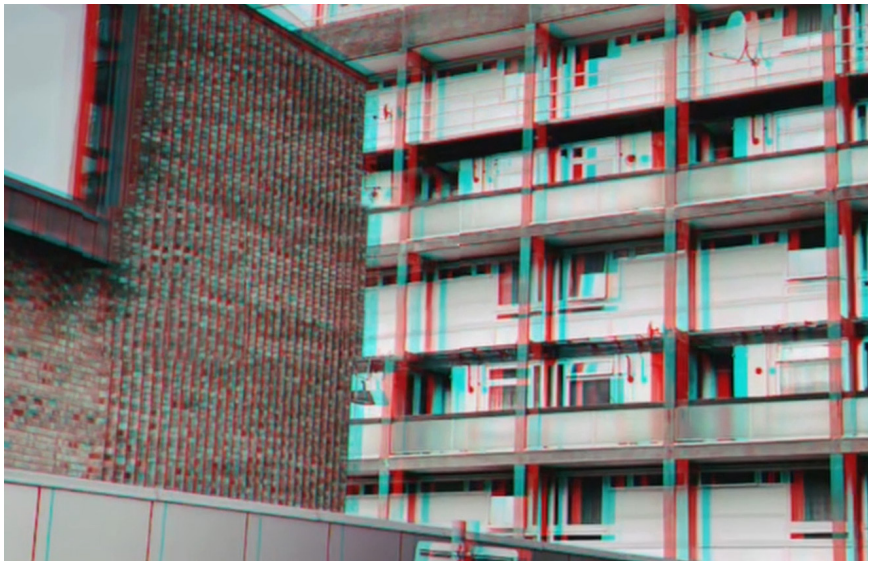
1
Once the vital conduit cutting through its industrial surroundings, the canal is hidden now; embarassed by its fall from grace in the eyes of its admirers . A sluggish, sullen current that pushes its way underneath housing estates and the rail line named in its honour,, with commuters unaware that "Paisley Canal Station" has any grounding in reality . Perhaps ashamed of its history; the town's faint but still perceptible memory of Martinmas, 1810 and the drowning of 85 Paisley-bound boat passengers is long, and although buried underneath the grey bricks and former mill buildings (now converted, as is de rigeur, into "luxury" flats with boxy bedrooms and 4 too few parking spaces), the record of the Countess Of Eglinton's fatal lurch is burned somehow into the streets.

Just months before the disaster, Robert Tannahill - poet, weaver, musician - had taken his own life in a culvert branching from the canal. Illness, self-doubt and episodes of "incoherent delirium" led him to the waterside, and weeks before the omen of his passing was cemented in an overhead snatch of his own "We'll meet beside the dusky glen on yon burnside" in the fields he walked through.. His last acts - burning his manuscripts ceremonially, folding and then abandoning his jacket at the tunnel entrance - were a rite as much to feed into his legacy as they were an act of desperation; a pact with the canal that has left his town filled with references to his life and work.

Few glimpses remain of the waterway, even around the rail line. High hedges mask sight of the low grey current - now choked with shopping trolleys, broken prams, mysterious cardboard boxes and children's clothing - but there are clues to be found

if you look closely enough. Outside one of the flat complexes, a drain cover echoes out the sound of more than just sewers passing underneath. The canal skulks beneath, biding its time, waiting to reach out for more unwilling swimmers.

If you look long enough though, and follow the trace of a line on the map, beneath the evocatively titled Craw Road you find the last vestiges of what was once mighty. A strip of water with converted lock-houses breaks out into the unimaginatively titled "Old Canal" - a pool larger than a pond but smaller than a lochan, still choked with the detritus of those who don't know or don't care of the history they're befouling. The only access - save pushing through bushes from the main road, tripping over discarded porn and empty cans - is from the back gardens of the "Groves" - Cromptons Grove and Stable Grove. Thanks to perfunctorily erected fencing, the residents can look but not touch, with the lurking remnants of the hungry canal waiting behind them; and maybe, just maybe, waiting until they look the other way and it can regain it's malign power.

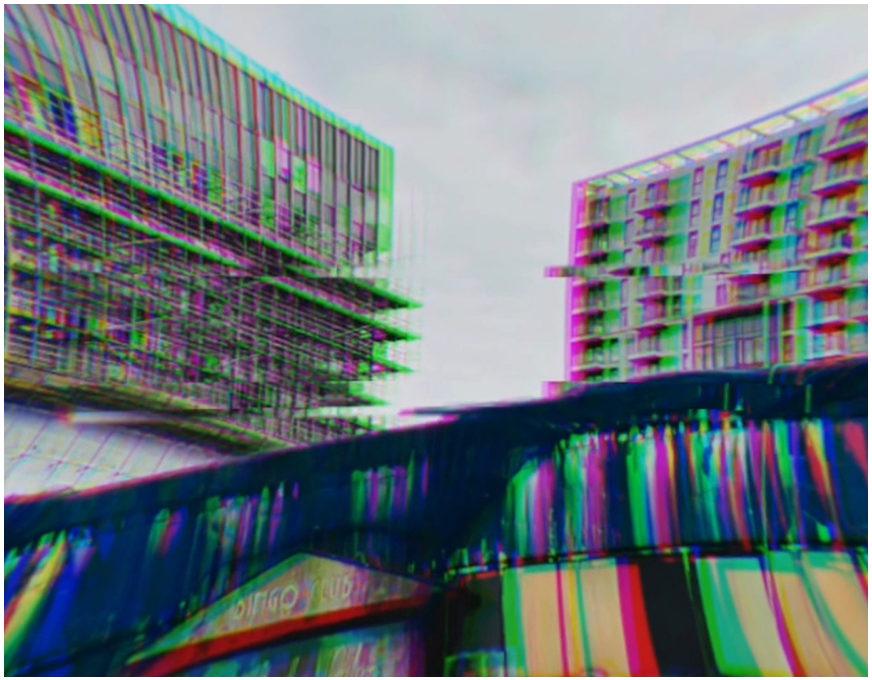


2
110 years later and 420 miles distant from Tannahill - in Southwark the General Strike was underway. Merely 9 days, and yet ingrained into the history of the working class and the Left, the clashes between police and workers left blows that rang not just in the ears of the disenfranchised workers but all the way up to the present day. The capitulation of the Unions led to worse suffering - and ultimately, death - than the original conditions were responsible for; but the workers rising up and striking a blow directly at mine-owners, plutocrats and landlords has echoed throughout the 20th century and onwards.

Not for the first time, the Thames was seen as a barrier; a raised barricade to keep the unruly mob at bay in Elephant and Castle whilst those-who-owned slept in their feather beds just north. The City - then as now - as haven for those with, and the

ends of the bridges as the start of an uncomfortable zone of no-go for the wealthy.

Fast forward 100 more years and the Heygate Estate is demolished, mired in controversial payments, secret deals, slush funds and cronyism. Just a few more years on and the original shopping centre (an icon of brutalist architecture) was crumbled back to the rubble it came from; the multi-cultural hub of unexpected shops and restaurants with a dizzying array of cuisines removed and sanitised to bring forth those more comfortable, palatable harbingers of the gentry - Prezzo, Waitrose,



Pret. The river, no longer a barrier, now acts like a bulwark for the gentrifiers to rest against as they plan their routes further south, into hitherto undreamt of areas for £6 flat whites and artisan sourdough.

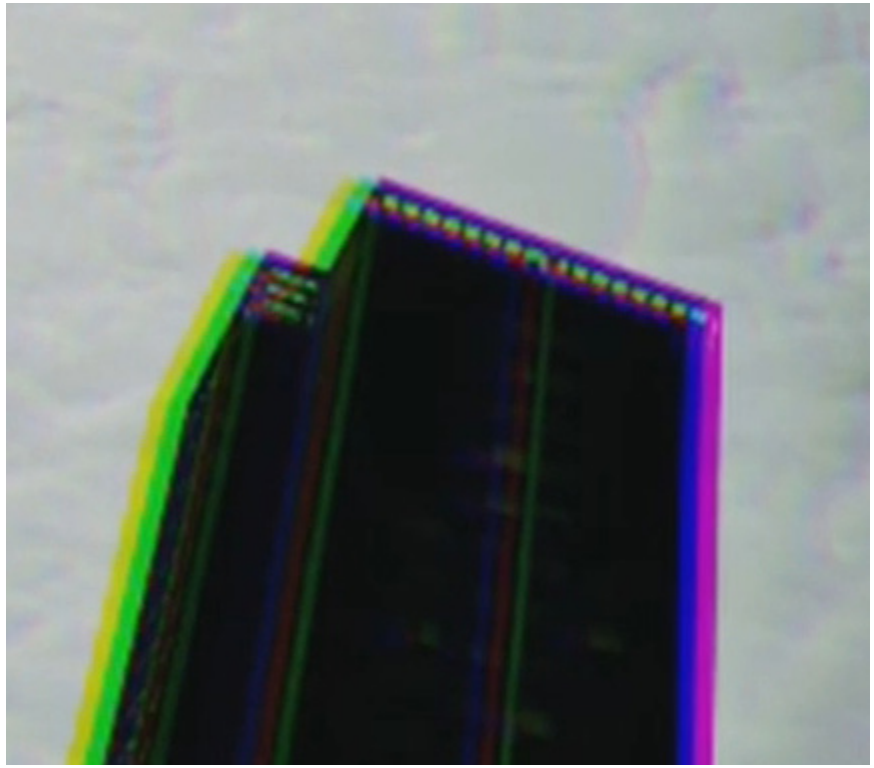
Dizzied by exhaustion, late hours and chemicals, there's many a glassy-eyed clubber had emerged from the railway arches onto Elephant Street and gaped in incomprehension at the brutalist Heygate architecture winking it's huge concrete lidless eyes in the rising dawn - or stumbled around the mobius loop of the Centre trying desperately to find the entrance to the Tube, trapped in an Escher-like ever rising staircase that somehow inexorably ended up at ground level again. Once, over stimulated and drunk, having been thrown out of an arch-dwelling club for falling asleep somewhere unsuitable, I was patted on the shoulder by a mountainous security man. "Here", he said, "You can look at the windows while you sober up and they'll look back at you. It'll be a comfort". With little authentic left but the ghosts of the Southwark Strikers prowling the streets, there's little comfort left for those without money to burn.

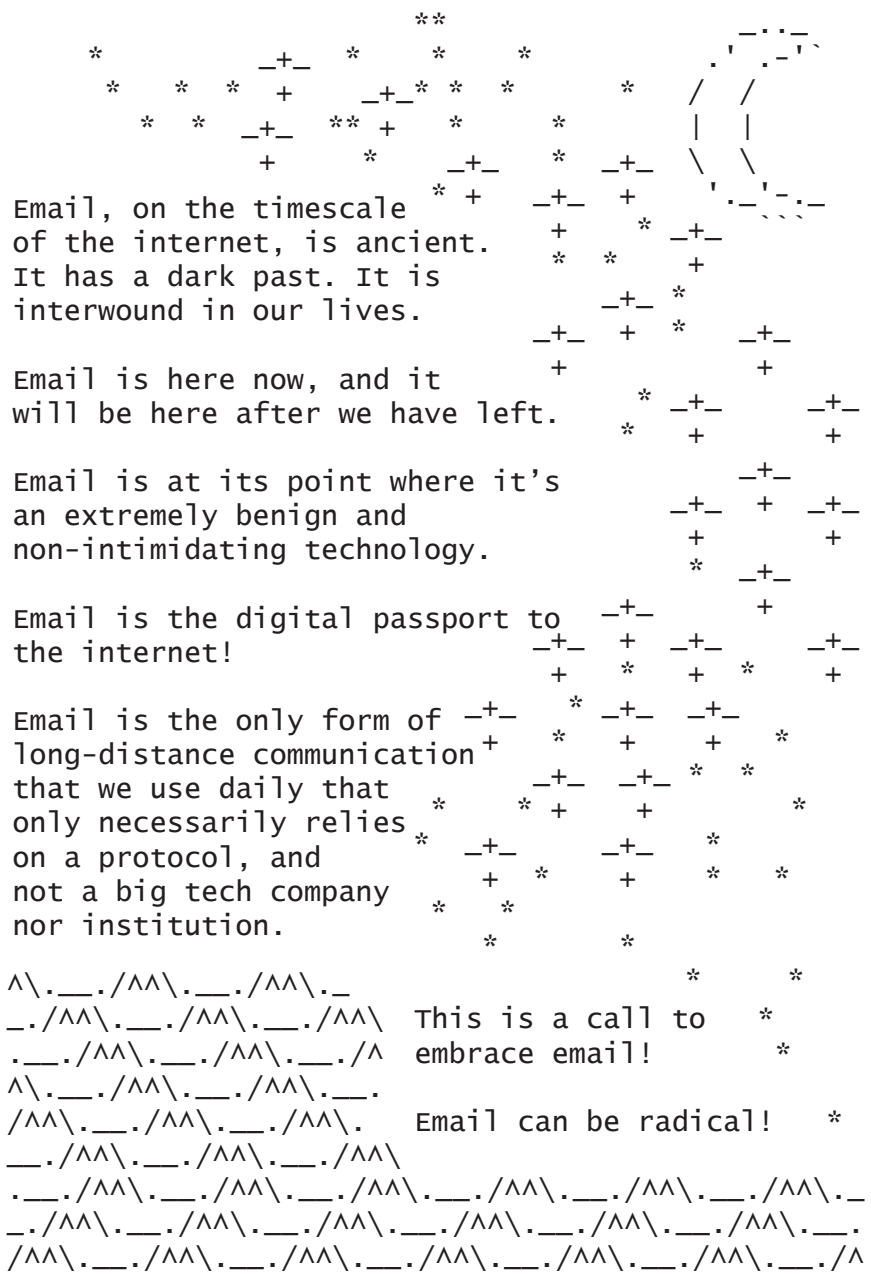
And the ghosts of the workers of the General Strike take voice with Tannahill and the Paisley weavers joining them in song:

"Going to live what we already know,
Going to live through these attempts,
We're not going nowhere,
But we still find something to believe in.

And in some part of us we know that it must prevail.
We see the soaring of the people,
The downfall of tyrants.
Hold fast to our flag, hold fast just this time
But not forever,
That's all we can do

We don't expect victory -
But these closed views we refuse"




Email, on the timescale
of the internet, is ancient.
It has a dark past. It is
interwound in our lives.

Email is here now, and it
will be here after we have left.

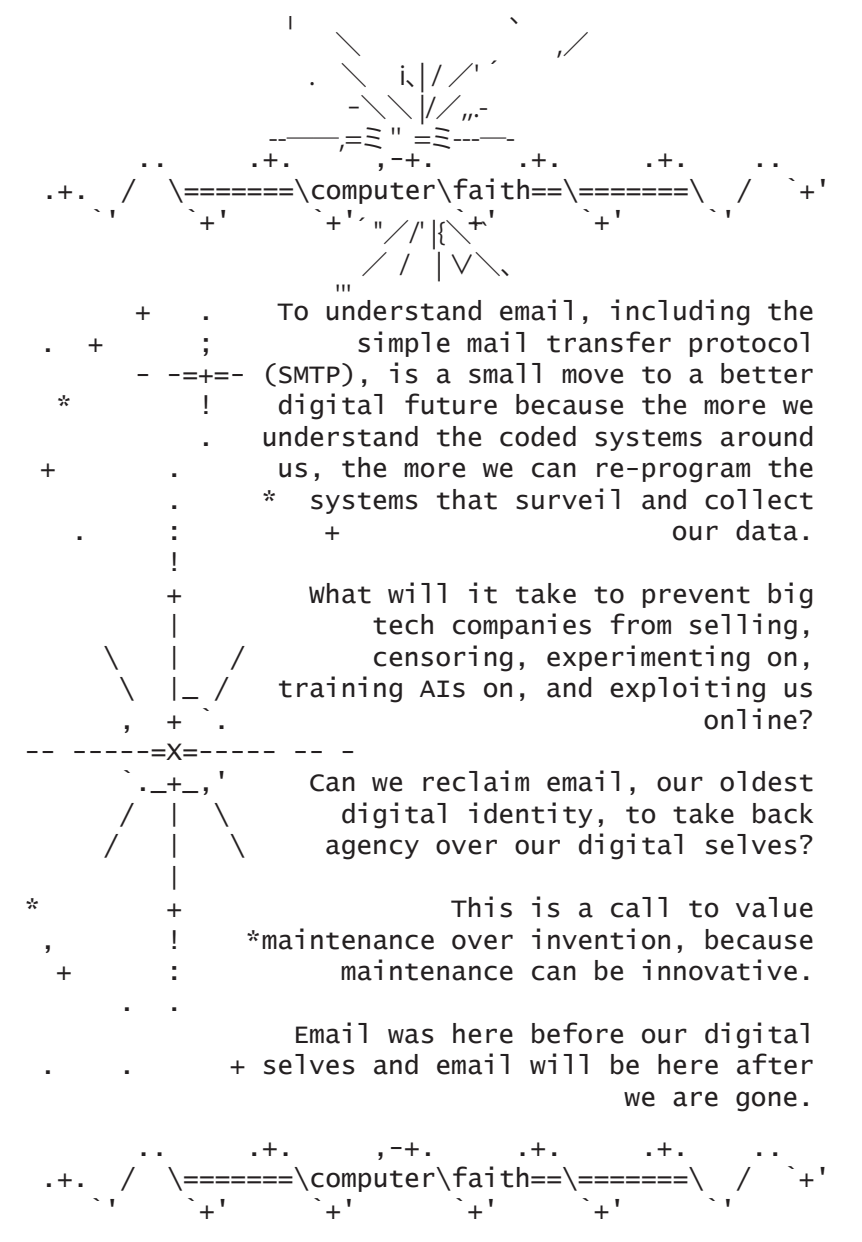
Email is at its point where it's
an extremely benign and
non-intimidating technology.

Email is the digital passport to
the internet!

Email is the only form of
long-distance communication
that we use daily that
only necessarily relies
on a protocol, and
not a big tech company
nor institution.

This is a call to
embrace email!

Email can be radical!


To understand email, including the
simple mail transfer protocol
(SMTP), is a small move to a better
digital future because the more we
understand the coded systems around
us, the more we can re-program the
systems that surveil and collect
our data.

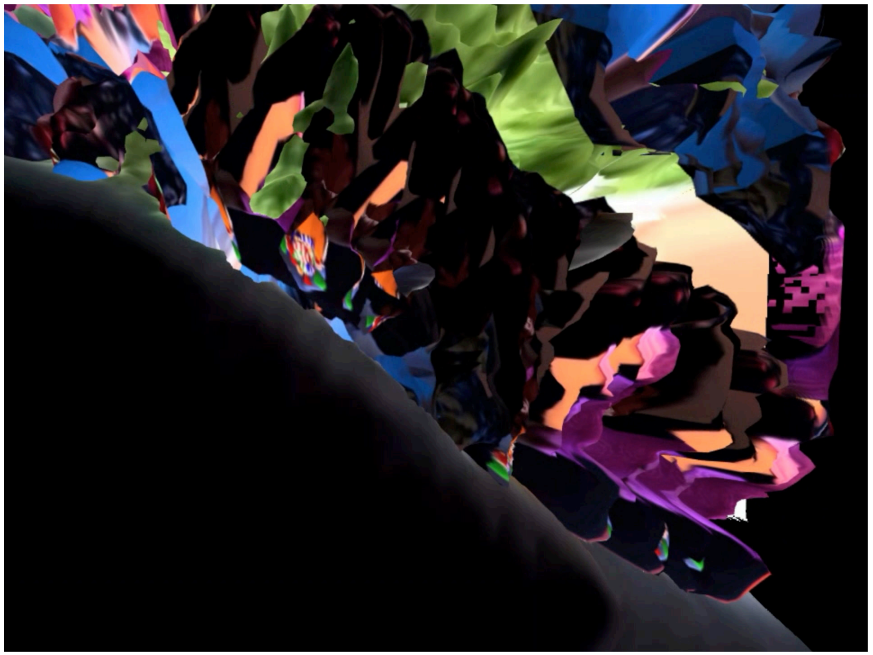
What will it take to prevent big
tech companies from selling,
censoring, experimenting on,
training AIs on, and exploiting us
online?

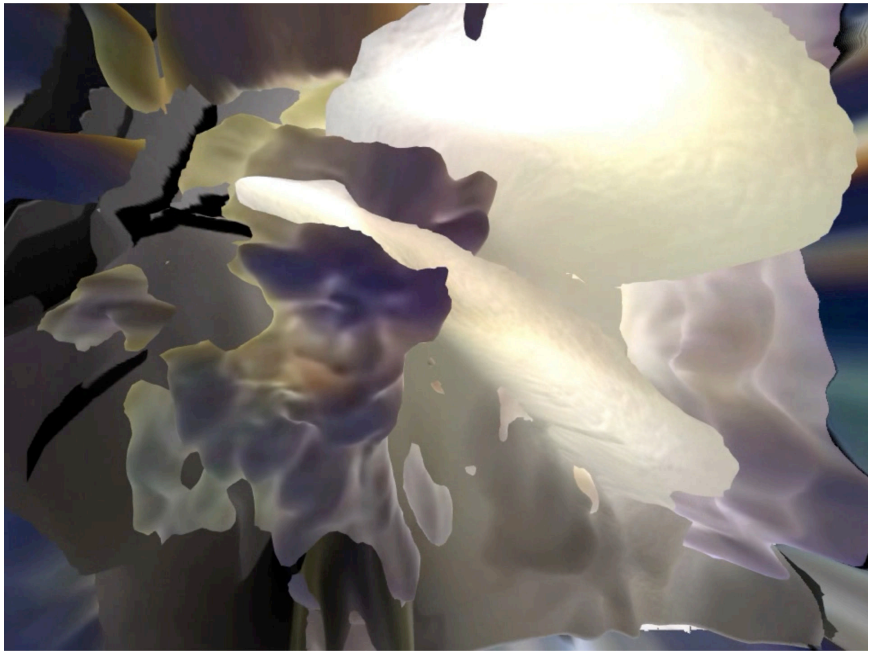
Can we reclaim email, our oldest
digital identity, to take back
agency over our digital selves?

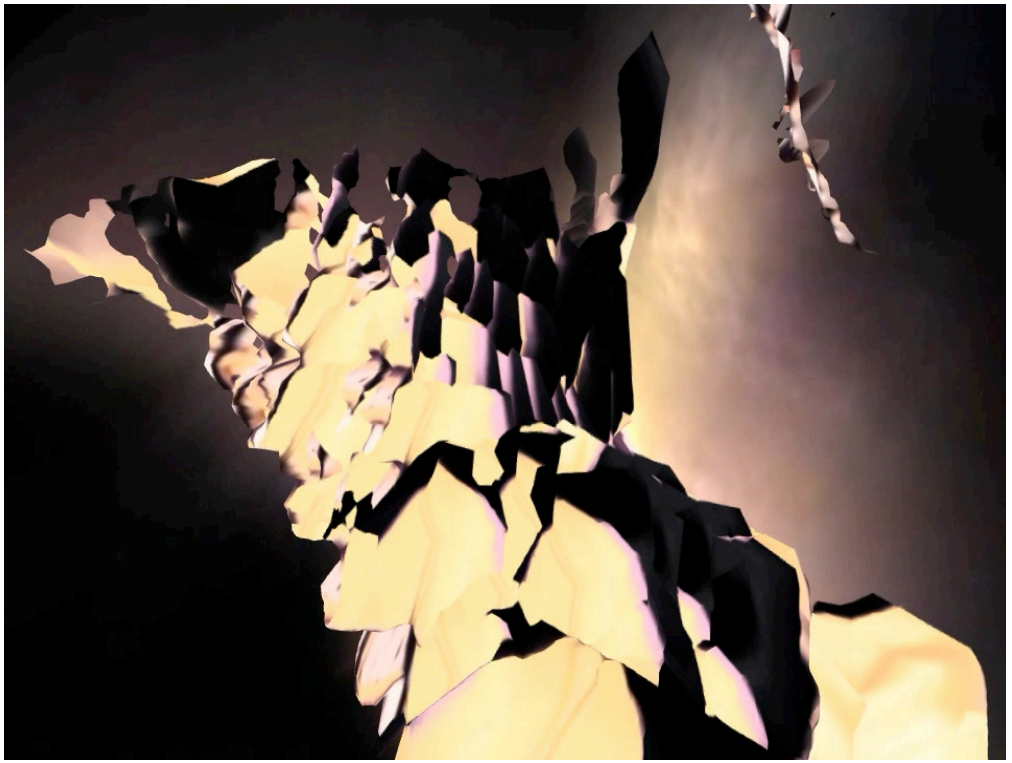
This is a call to value
*maintenance over invention, because
maintenance can be innovative.

Email was here before our digital
selves and email will be here after
we are gone.

trash panda qc - blob world fragment 01, images + photos







Lee Tusman - Why I Built a Creative Coding Library for Your Old Laptop

As an artist primarily working on the computer I create algorithmic art, interactive stories, computational poetry, and generative sound works. Sometimes I make work entirely in code but much of the time I combine writing software and incorporating my own drawings, photographs, music, and recordings to create unfolding animations, playable narratives, and experimental games.

For the past fifteen years, I've relied on creative coding libraries to make this work possible. I started with Scratch and similar game-making no/low-code tools, but quickly hit their limits. I transitioned to Processing when I realized I needed to write software to shape the projects I envisioned, and found that I enjoyed the process of learning to program and the iterative nature of developing artworks this way.

In 2015, I discovered p5.js—a JavaScript reimplementation of Processing designed for the web. I've taught with it, organized Processing Community Day conferences, and created thousands of projects in p5.js and Processing. I value that these tools are free, libre, and open source, built by a community of people from diverse backgrounds contributing to both the language and its culture. And I find the Processing way of programming to be intuitive, flexible, powerful: setup() and draw(); iterative development; stateful algorithms that you build up in order; plentiful and easy documentation and debugging.

I work as a professor and in DIY artist-run spaces. Working in new media art and technology, I've become increasingly troubled by the field's consumption patterns and their environmental impact. In my art collective's studio and gallery spaces we save previously-used materials and gather wood, fabric, and electronics. We save these things and then put them to re-use, adapting them to newer installations and exhibitions. Working in new media art and technology, I've become increasingly concerned with the high level of consumption in the field, and am interested in taking steps to reduce our impact on the environment. Several years ago, I learned about permacomputing—a nascent movement considering environmental sustainability in computing, inspired by permaculture.

I want my work to live on, on its original devices. I want to create things that run on many computers, including older hardware. I'm trying to resist planned obsolescence—the designed-in failure that pushes us to buy new computers and phones every few years. First I transitioned to installing Linux on older machines, sometimes building up machines for my students during the pandemic, and teaching them these same skills.

I've been frustrated with the business models of subscription models to make artwork, that puts access out of the hands of my students and some peers. I also don't love the locked-down nature of these systems, or the requirement to buy new products just to be able to participate in making digital artworks. And I've personally experienced multiple times where a platform shuts down, alters their business model or infrastructure and I find my own artwork disappears, no longer functions, or costs some money to keep working or that I need to buy a new phone, computer or other device just to keep working with it.

These concerns converged in the creation of L5, a new creative coding library. Just as p5.js brought Processing to JavaScript and the web, L5 brings it to Lua—a lightweight, desktop-oriented language. The syntax is similar to JavaScript but even simpler. A complete install is only a couple megabytes.

For projects meant to be presented on a computer I've found the web environment increasingly limiting. During summer 2020, I discovered Pico-8, a retro-inspired game engine for making 2D games. It's a comprehensive environment with sprite graphics tool, sound editor, code editor, and in-app game browsing and sharing. I loved it and made many games and applications, particularly appreciating Pico-8's Lua language API commands. But its constraints—128×128 pixels, synthesized sound only, code length limits—made it unsuitable as a general tool for my artwork. Beyond syntax, I loved Lua's speed. As a library with a C API, tight syntax, and simple features, it's incredibly fast to use and run—a result of its two-step process where code compiles to bytecode and interprets at runtime.

In 2022, I began using LÖVE (Love2d) to make experimental game-like projects. Like Pico-8, it uses Lua but with a more extensive API. I tried carrying over my Processing and p5.js workflows. Love2d is great: there's a long-running community, extensive resources, well-documented. But after several years, it didn't feel as free and sketch-like as my open-ended experimentation with Processing and p5.js, and it really is oriented around games. The API was too big and frankly, not Processing-like enough for creating algorithmic art. The syntax was less intuitive.

In July 2024, I began building drawing functions in a mini-library wrapping Love2d's native functions. Within a week, the experiment proved successful—flexible, intuitive, fun. I was essentially creating Processing/p5 in Lua, using Love2d underneath. So I kept implementing the Processing API. But the API alone doesn't define a Processing-style language.

My first roadblock: p5.js and Processing allow drawing graphics anywhere—in setup(), draw(), mouse events, key presses. Love2d doesn't, out of the box. As an open source project I tried modifying its event loop, but this seemingly minor change caused issues like flickering when no background color was drawn. I learned about double buffering. I rewrote event detection to be more Processing-like. And I faced bigger decisions: When to match Processing versus p5.js? When to rewrite Love2d's methods versus keeping its native utilities? How to handle inevitable bugs?

I documented my daily work on my project log. I had a list of about 200 Processing/p5 functions to implement and a growing TODO list of bugs. This intensive work was only possible because I was on sabbatical and an artist-in-resident at ZK/U in Berlin. Some days I worked twelve hours or more, motivated and excited, simultaneously "dogfooding"—using L5 to make my own art as I developed it.

I started doing L5 Studies—a practice I learned from Teletype Studies years earlier. A *studies* practice means diving deep into using a tool regularly, iteratively building projects, documenting progress to develop mastery. I tend to do these studies for a month, using my daily experiments to scaffold more advanced work. In fact, for many years I practiced regular code sketching, producing hundreds of rapid small projects

in p5.js, some that later turned to commissioned artworks.

In my early weeks creating small daily projects while building L5 I found countless bugs or missing parameters. I'd check the p5.js and Processing references, then implement them in Love2d/Lua. Some were minimal; others required serious re-engineering. I had to make judgment calls.

An early decision, within the first week: consider accessibility. p5.js implemented a describe() function that creates screen-readable canvas descriptions, part of a fundamental commitment to access of that tool and its community and caretaker the Processing Foundation. Currently, L5's describe() outputs text descriptions that can pipe to text-to-speech tools. Similarly, the L5 website was designed for accessibility in multiple capacities, with intention behind creating smaller page, image and animation sizes; "lazy" loading and design for low-bandwidth; offline, downloadable copies of the website; semantic HTML for easy scanning, translating and screen reader use; and functionality without JavaScript. I'll continue evaluating accessibility in L5 and building ways to gather feedback on progress.

```Lua
require ("L5")

function setup()
  size(600, 800)
  block = height / 40

  frameRate(1)

  describe("A maze of twisty little passages in each frame")
end

function draw()
  background(255)
  block = floor(random(10,140))
  for y = 1, height, block do
    for x = 1, width, block do
      if random() < 0.5 then
    line(x, y, x + block, y + block)
      else
    line(x + block, y, x, y + block)
      end
    end
  end
end
```

Through this work, I've developed deep appreciation for Lua's emphasis on longevity, compactness, and efficiency. It's a tight language built for embedding, extremely slow to change—only small updates from Lua 5.0 in 2000 to 5.4 in 2020. The Masterminds of Programming interview with Lua's developers reinforced my appreciation for their philosophy.

LÖVE/Love2d, which underlies L5, shares Lua's strengths: open source, minimal dependencies, tiny (a couple megabytes), multi-platform, clear API, well-maintained with a friendly development team, easy to port. It's been maintained since 2008 with continuity through leadership changes, supported by a strong community. These are L5's upstream strengths, though monitoring Love2d development remains important for maintaining L5.

As sustainability concerns in computing culture have grown more urgent for me, I've been inspired by ideas like the article "Potential and Limits of Constraints in Computational Art, Design and Culture". With L5, I'm trying to contribute by building for older computers and devices while running super-efficiently on modern hardware. This takes serious, sustained effort.

Seeing the first L5 programs posted on social media and receiving contributed bug reports and fixes has been gratifying.

L5 is deeply indebted to Processing/p5 and its community, stewarded by the Processing Foundation, and with discussions with its maintainers. Tutorials and documentation have been adapted from and build on these strong foundations.

The L5 creative coding library is still young. Its success depends on a growing community alongside my caretaking. But the foundation is solid: a Processing-style creative coding environment that runs on hardware you already own, respects your time and resources, and aims to keep your artwork alive for years—not just until the next mandatory upgrade.

With some care and maintenance your 2026 machine should be able to be of service for at least a decade. Likewise, your 2012 laptop isn't obsolete. It can still be your work computer, or a platform to present your projects in exhibition. It's a perfectly good computer waiting to make art.

References

* L5 https://L5lua.org
* Permacomputing Wiki https://permacomputing.net/
* My project log https://leetusman.com/nosebook/log
* Teletype Studies https://monome.org/docs/teletype/studies-1/
* Masterminds of Programming https://archive.org/details/MastermindsOfProgramming
* Potential and Limits of Constraints in Computational Art, Design and Culture https://monoskop.org/images/6/6a/Mansoux_Aymeric_et_al_2023_Permacomputing_Aesthetics.pdf (PDF)

Lauren Sarah Hayes - (A) LIVE ELECTRONICS

Thorsten SideboArd
SB#> _

tangent

Back in the 50s, computers were room-sized machines — essentially massive calculators. The first IO interface was control panels, manually entering programs via switches; Next came batch processing and punch cards.

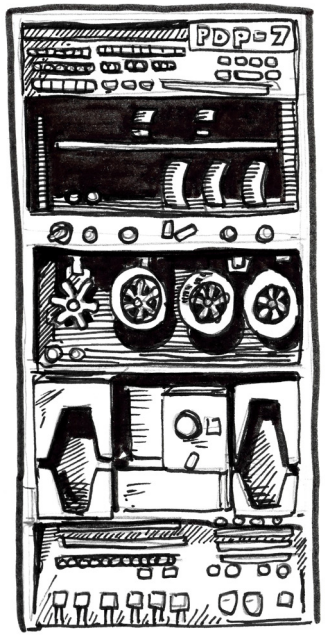
Everything changed with the concept of time-sharing, and the most successful implementation - CTSS, the Compatible Time-Sharing System, developed at MIT in the early 60s. For the first time, multiple terminals could share a single computer, and users could have an *interactive* programming experience.

Louis Pouzin, a visiting French scientist, had been accumulating programs and routines for CTSS and realized he should be able to combine and reuse them — use them as building blocks for larger commands. He wrote an early prototype called RUNCOM, which supported argument substitution and basic scripting.

Around 1964, the Multics project got started — a collab between the supergroup of of MIT, General Electric, and Bell Labs. Pouzin didn't stay on for it, but contributed a paper coining the term shell: '*The SHELL: A Global Tool for Calling and Chaining Procedures in the System*.' Glenda Schroeder picked it up and implemented it as the Multics command language.

Multics envisioned computing as a utility - like water or electricity, a central resource that users would subscribe to. It had great intention, but eventually imploded under its own weight, 'second-system effect' as Fred Brooks would describe it - filled with loads of great original ideas, but over-engineered, slow, and arriving just as the industry was shifting to smaller minicomputers. Bell Labs pulled out first. Despite being a commercial failure, it spawned many features still around today.

While working on Multics, Bell Lab's Ken Thompson had written a video game called Space Travel. Off the project now, he ported it to a DEC PDP-7 — ignored the existing OS entirely, wrote his own routines from scratch, and eventually found himself building a minimal operating system from the best ideas Multics had generated: hierarchical filesystem, simple process model, command line shell. Punning on the name, he called it Unix. Its DNA lives on today in Linux, Android, and macOS.



main thread

Starting out as a Linux sysadmin in the 90s, I've lived my work life in a terminal shell — it's how you manage servers, log into remote machines, troubleshoot the slow website. How you get shit done. The alternative is the point-and-click interfaces of Windows and Mac, which are convenient until you need to do something fifty times, or to reproduce it exactly. With a shell you can work interactively, or you can save your commands in a script, which gives you a consistent, repeatable command - plus editable and shareable.

Sysadmins write a lot of scripts — automating password changes across machine and network fleets, archiving logs, monitoring bandwidth. But a script isn't a program. A web browser, a music player, a video editor — that's a different order of complexity entirely. After years working with operating systems, I wanted to learn the language Linux was written in: C.

C goes right back to Unix at Bell Labs. The first iterations were written in assembly — barely a step above machine language. Thompson needed something higher, and adapted Martin Richards' BCPL into his own stripped-down language: B. His colleague Dennis Ritchie improved on it, incrementing the name to C. The full manual — the classic K&R The C Programming Language — is a modest book you can read in a weekend. I'd had a copy of it on my shelf for years, I'd found it lying around the office at SRI, my first Linux sysadmin job.

I always think of C as the original hacker language. There is a fairly recent O'Reilly book on C which describes it as punk rock - comparing its brevity to the classic "Here's 3 chords…now form a band".

One of my very first C exercises was in how to write a simple command line shell. The purpose of the tutorial was to demonstrate how to read text input from the terminal, compare the text to a list of commands and then execute the command. My simple shell only had one command to start with - list the files in your current directory.

My desire to learn C wasn't coming from sysadmin work. For a few years I'd been doing creative coding on the side — node audio scripts over RabbitMQ, sensor-driven Arduino cars. I'd been making music in Ableton for years but was never satisfied with using samples and loops. Inspired by friends and house mates I've known, who built their own systems via max/msp and custom languages, I decided I wanted to learn how to program audio and create my own sounds.

A few weeks later I was learning to synthesize sine waves. Unlike GUI music programs where you can turn dials and sliders, I was writing and launching command line utilities. I could write code to create a frequency playing at 440Hz, compile, run it, and hear the audio—but I had no way to control it once started. How could I actually interact with the audio process after it had started?

That's when I realized I could combine my audio code with a shell. Rather than launching individual processes, I could add commands to my shell to create a sine wave, and then other commands to view its status, change its frequency. Suddenly, rather than having individual tools, I had a platform, an environment to grow and expand.

SbShell began to take shape - i had a metaphor to build on - rather than a shell around the operating system, SbShell was a shell around a sound engine, a way to perform live, creating and manipulating audio computations through a command line interface.

I found a book, perfect for my use case, which both taught C and audio programming - "The Audio Programming Book" by Richard Boulanger and Victor Lazzarini. It laid a great foundation of practical C knowledge. The next big influence was "BasicSynth: Creating a Music Synthesizer in Software" by Daniel Mitchell. This one taught me how to organize my instruments, arming me with the concept of a central Mixer which owns all the Sound Generators, and which is responsible for the Audio Callback. I had a simple FM synth, basic time counter and step sequencer. I added sample support for looping and one shot playback.

I spent a year working my way through Will Pirkle's insanely great "Designing Software Synthesizer Plug-Ins in C++", which starts you off with simple synth components - oscillators, then envelope generators, a DCA, then filters - building up a simple synth from the beginning, gradually adding features throughout the book, until you've built a series of increasingly complex implementations, which is where both my FM and subtractive synth come from. The FM synth is a 4 operator synth based on the DX100 architecture. The subtractive synth is based on a MiniMoog.

I worked my way through Curtis Road's "Microsound", with a detour in the writings of Ross Bencina and Robert Henke, before creating my granular synth implementation. Using granular synth was the perfect way to upgrade my sample looper, which previously had been rigidly incrementing through the sample array, strictly following sample time, but it turns out sample time is not the same as wall time and has to account for drift. Around this same time I replaced my own hand-rolled time-keeping with Ableton Link, using it as the source of timing truth for my mixer, which

beautifully solves the wall clock / sample time issue and allows soundb0ard to automatically sync with any other Loop enabled device, and, combined with a granular system, my loops flowed smoothly.

After a few years working in C I felt I had earned an upgrade, and updated the code to use the C++20 standard, followed by spending another year replacing my massive regex text matching interpreter loop with a real programming language, based on Monkey from the amazing "Writing An Interpreter In Go" by Thorsten Ball. Although that book is in Go, I worked my way through it implementing it in C++, then extended the language with objects representing the sound generators, allowing them to be manipulated and controlled in real time through code.

Beyond the sound generators, the Soundb0ard language (Slang) also contains a novel object called a Computation. It was inspired by the shape of Arduino and Processing sketches, whereby the object has two functions - there is a setup() function, called once for initialization and a run() function that is called repeatedly. In the Processing world, the run function is called for every frame; in Soundb0ard, it's called once per bar.

Here's a simple example:

```
let bounce_roll = comp()
{
  setup()
  {
    let decay = 0.75;
    let num_hits = 12;
  }
  run()
  {
    if (count % 16 == 3) {
      let pos = 0;
      let gap = pp * 4;

      for (let i = 0; i < num_hits; i++) {
        let v = 60 + (i * 5);
        if (v > 127) { v = 127; }

        note_on_at(sbdrum, 1, pos, vel = v);

        pos = pos + gap;
        gap = gap * decay;
      }
    }
  }
}
```

Entering long code sections at the command line is unwieldy. Instead you can work in a code editor, and import/monitor the file from the Soundb0ard Shell, with live re-loading to enable live coding.

More recently I added a Drum Synth with 9 voices, finally getting a deep kick drum

I was happy with. So - all these features - several unique synths, a drum machine, granular sample loopers and one shot playback, a number of FX (delay, reverb, smudge, scramble) - all controllable in realtime via a fashionable shell and a full extensible programming language which supports integers, booleans, strings, arrays, hash maps, first-class functions (with closures and recursion), sound generators, computations, and a live coding interface - I feel like I have something good here that's worth sharing!

I've been working on it for about 10 years now, using it for live performances and recordings during that whole time. I think it's pretty damn solid at this point, and to be honest, I think it's the best art I've made!

```
          _____                    _ _     _____            _
         /  ___|                  | | |   |  _  |          | |
         \ `--.  ___  _   _ _ __   __| | |__ | |/" | __ _ _ __ __| |
          `--. \/ _ \| | | | "_ \ / _` | "_ \|  /| |/ _` | "__/ _` |
         /\__/ / (_) | |_| | | | | (_| | |_) \ |_/ / (_| | | | (_| |
         \____/ \___/ \__,_|_| |_|\__,_|_.__/ \___/ \__,_|_|  \__,_|
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::: vol:1 bpm:140 looplen:3840 midi_device:false websock:false ::::::::::::
:::::::::
::::::::::::::::: delay: Delay! ms:107.143 fb:65 rat:0 mx:0.5 mode:norm sync:1 sync_len:1/
16
::::::::::::::::: reverb: Reverb! predelayms:40 reverbtime:2000 wetmx:100
::::::::::::::::: distort: Distortion - mode:HARD_CLIP(0) threshold:0.5 drive:1
::::::::::::::::: xfader: [ ] <------I-------> [ ] xpos: 0 fade_tics:1

[Varz]
pplooplen = 3840
golden_ratio = 1.61803
TWO_PI = 6.28319
PI = 3.14159
pp = 240
count = 4
bd = Sampler(bd/kick8.aif) vol:1.00 pan:0.00 pitch:1.00
bdd = Sampler(bd/808kick.aif) vol:1.00 pan:0.00 pitch:1.00
ch = Sampler(ch/2hat.aif) vol:1.00 pan:0.00 pitch:1.00
cp = Sampler(cp/clap17.aif) vol:1.00 pan:0.00 pitch:1.00
dx = FmSynth(mo_jazz) vol:0.80 pan:0.00 algo:7
dx2 = FmSynth(MAW2) vol:0.70 pan:0.00 algo:0
dx3 = FmSynth(SMMTH2) vol:0.70 pan:0.00 algo:1
hh = Sampler(ch/Hihat0002.aif) vol:1.00 pan:0.00 pitch:1.00
mo = MiniSynth(default) vol:1 pan:0 voice:SQR3(1)
oh = Sampler(oh/dmx_hh_open.aif) vol:1.00 pan:0.00 pitch:1.00
rim = Sampler(rims/rim1.aif) vol:1.00 pan:0.00 pitch:1.00
sb2 = DrumZynth -  - vol:1
sbdrum = DrumZynth -  - vol:1
sd = Sampler(sd/2snare.aif) vol:1.00 pan:0.00 pitch:1.00
shkr = Sampler(perc/chezShaker.aiff) vol:1.00 pan:0.00 pitch:1.00
shkr2 = Sampler(perc/tuffShaker.aiff) vol:1.00 pan:0.00 pitch:1.00

[Procz]
p0 # "countr"
```

If this sounds like something you'd be interested in playing with, check out the full USER_GUIDE.md and more examples at:

https://codeberg.org/sideb0ard/SBShell

a highpoint lowlife production