Caleb Mitchell

13 May 2024

Music Technology Capstone

Digital Cubism: Algorithmic Drum Sequencing with Max MSP

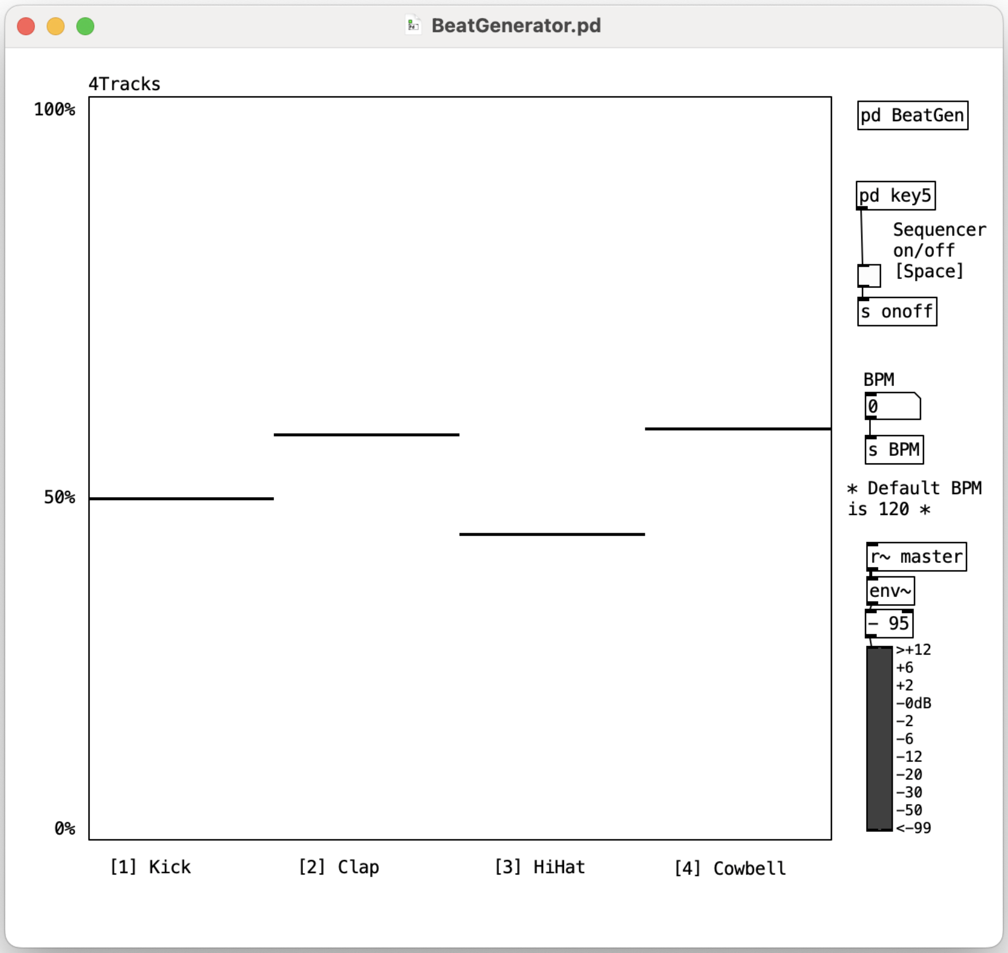
Introduction and Artistic Influence

Through my time living in Chicago and being exposed to diverse kinds of dance music, I've fallen in love with certain strains of minimal techno and house music. One artist that has made me rethink the possibilities of techno is known as Hieroglyphic Being, a Chicago techno artist that has a more live and improvisational approach to techno than the typical artist within the genre. Their approach to recording techno tracks is to create multiple arrangements of drum sequences and melodies which they switch between on the fly based on feel. They record this live take as the final track, giving them more organic control of when and how new patterns and ideas come into the mix. The album Synth Expressionism/*Rhythmic Cubism* by Hieroglyphic Being came to serve as the creative foundation for what would be my capstone performance. The phrase "Rhythmic Cubism", accompanied by Hieroglyphic Being's approach to live and eclectic techno sequences, has stuck with me and left me fascinated by its implications. I interpret the idea of Rhythmic Cubism as creating collages of rhythmic patterns that, when woven together create the impression of a rhythm, like the visual cubist movement in which all perspectives of a subject are combined to create an altered impression of this subject.

I have also grown interested in algorithmic music in the past few years thanks to discovering artists like Ryoji Ikeda, Mark Fell, and Gábor Lázár. I find the ability to add more organic forms of chaos and live experimentation to live performances of electronic music to be a fascinating idea, as it incorporates some of the feeling of stumbling upon sounds and arrangements that is familiar to instrumental performance in genres like jazz. Keeping the synthetic sonic aesthetics of techno and combining them with the arrangements and spirit of free jazz were the main creative goals of my capstone project.

Process of Creation

For my capstone project, I decided to create an algorithmic drum sequencer that could create new drum and synth patterns on the fly to use in a live performance. I chose to expand upon a previous software sequencer that I had made using the program Pure Data. Here is a screenshot of the original Pure Data project's UI:



This sequencer consisted of 4 tracks of drums with a slider tied to each track that set the probability of each step in a 16-step sequence being triggered. With a slider set at 50%, regenerating a pattern would result in each one of the 16 steps having a 50% chance to be activated and thus trigger a one-shot sample playback. This sequencer also included simple transport controls like a pause and play button, and a way to set the BPM.

Problems with this design included:

- Limited to 4 tracks of drums
- Hard coded samples, cannot swap sounds on the fly
- No control over pitch playback of samples
- No control over pattern lengths
- No ability to mute drum tracks on the fly

- Sequencer clock works in message domain (block processing) not in signal domain, results in imprecise timing when CPU is under large load

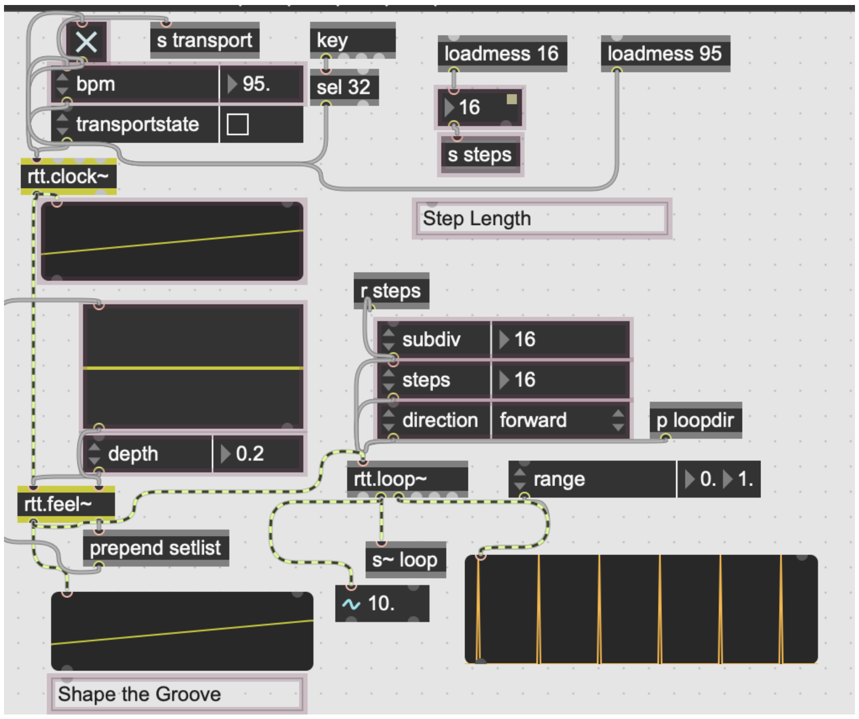
Due to these limitations, I decided to rebuild this base project using the program Max MSP, a proprietary software built upon the framework of Pure Data, but with more in-built objects and functionalities. This allowed me to avoid building more modules from scratch in Pure Data to implement ideas that were already native to Max MSP, allowing for quicker implementation of more complex ideas. In my research on algorithmic sequencing in Max MSP, I discovered the Rythm and Time Toolkit, a package for Max MSP created by Philip Meyer in 2023. From his website, Meyer describes RTT as a package for Max MSP that is “devoted to modular, signal-based sequencing.” I quickly realized that this package had most of the modules that I had envisioned for my sequencer already pre-built, providing an excellent framework to further abstract upon. For instance, the object [rtt.clock~] creates a clock in the signal domain, outputting a simple signal ramp that goes from 0 to 1 based on a given BPM value. This solved the issue of sequencing in the message domain from my original Pure Data patch, as I could easily create a clock in the signal domain.

The Rhythm and Time Toolkit also includes objects that integrate my previous simple random algorithms for drum pattern generation, but the modularity of the package allowed for me to easily add complexity and control onto of this. Features of my sequencer built upon RTT objects includes:

- 8 tracks of midi tracks, able to output to midi compatible devices over USB
- Random generation of gate patterns based on user input on multislider, one slider per step, setting the probability the step gate is turned on per every pattern regeneration
- Melodic sequencing of every track, with variable number of steps and range of notes
- Note sequences can be quantized to a specific scale, allowing for random melodies to be in key with each other
- Variable master pattern step length
- Variable subdivisions of master clock loop
- Ability to play sequences forwards, backwards, or a completely random step on each step

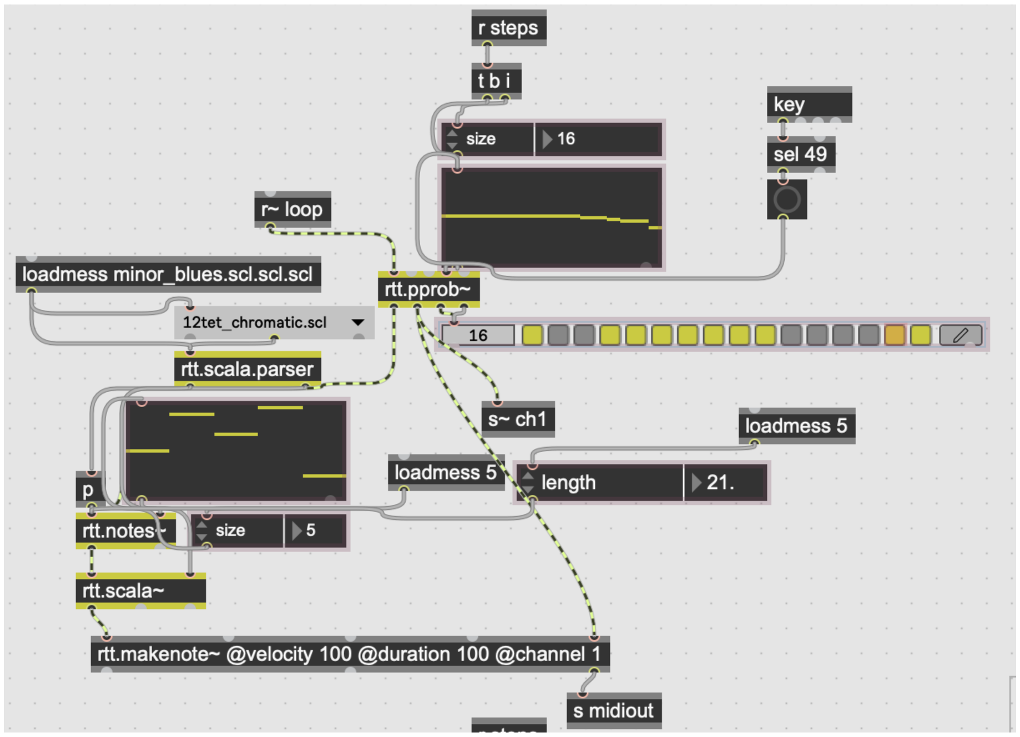
Code Excerpts and Signal Flow

Transport and Clock modules:



- [rtt.clock~] takes on/off toggle (0 or 1) to trigger the start of a phasor ramp, running at a frequency relative to the set BPM (beats per minute, tempo) value sent to the object. The [key] object here checks for input from the space bar, starting and stopping the sequence on every press.
- [rtt.feel~] gives a way to dynamically speed and/or slow the ramp from [rtt.clock~], providing a way to create leading or lagging rhythms similar to that of a live drummer. The multislider object connected to [rtt.feel~] allows the user to draw in a contour of how the sequence's speed should modulate over the course of one loop, where higher slider values increase the speed and lower values decrease the speed. The sliders when centered do not affect the playback speed. The depth parameter allows the user to control how harshly [rtt.feel~] modulates the rhythms playback.
- The output of [rtt.feel~] is fed to the [rtt.loop~] object, which subdivides the ramp from [rtt.clock~] based on the step length and subdivision parameters provided. This works by sampling the clock's ramp at equal divisions, meaning if the step length were 4, the ramp would be sampled at 0, 0.25, 0.5, 0.75, and 1. Every time the ramp is sampled at these intervals, [rtt.loop~] outputs a signal impulse, which is then routed out to a send object to allow for easier patching to each drum channel

Individual Track Pattern Modules:

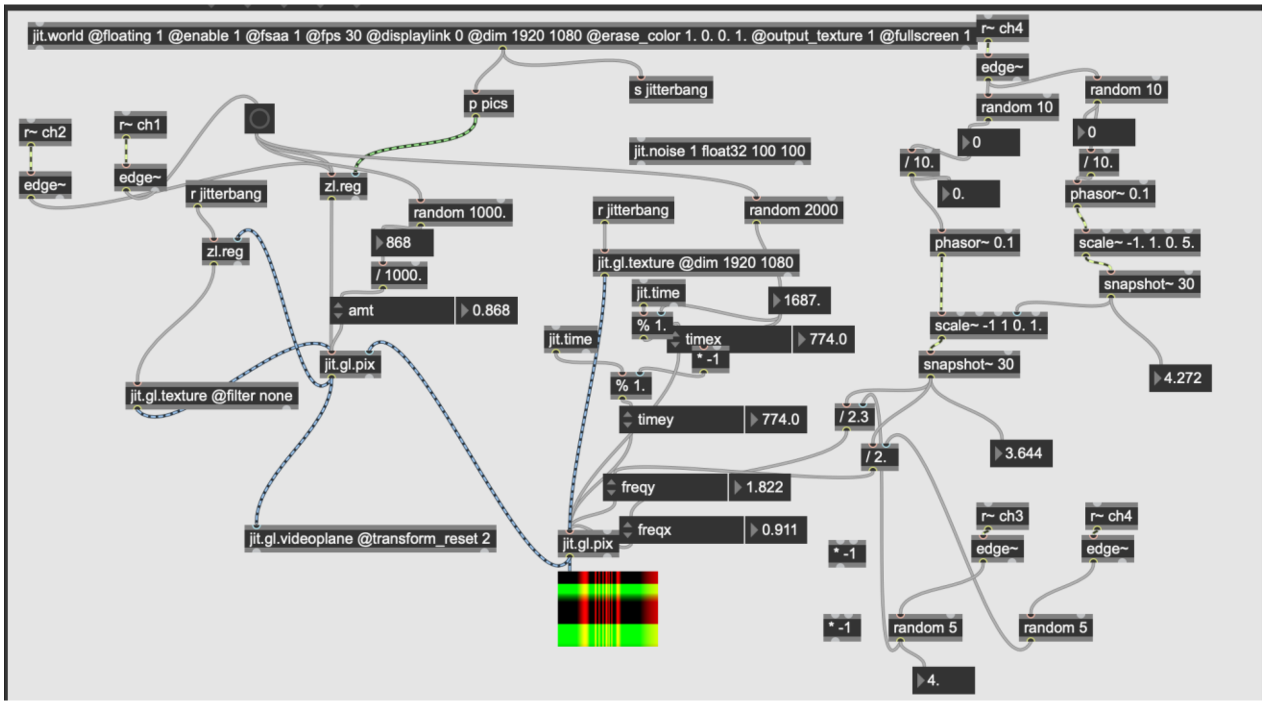


- [rtt.prob~] is the main object responsible for randomly generating new drum patterns on the fly. It receives signal impulses from the [rtt.loop~] object to continually step through each step of the sequence. [rtt.prob~] receives the master pattern length through the [r steps] object, variably changing the step length whenever the number is changed. The multislider connected to [rtt.prob~]'s input sets the probability of each step's gate being turned on when a pattern is regenerated. This means that for a pattern of 16 steps, there will be 16 sliders, each corresponding to a step. When a slider is set to its maximum, the corresponding step's gate will be turned on 100% of the time the pattern is regenerated, and this probability can be set at any value from 0% to 100%. This allows the user to quickly draw in a contour of a rhythm they would want from a certain drum track and retrigger the [rtt.prob~] object to create new patterns based on this probability. This excerpt is for midi channel 1, and whenever the user presses the number key 1 and/or changes the multislider values, a new pattern is created. The yellow and grey UI object connected to [rtt.prob~]'s output visualized what pattern is outputted.
- [rtt.scala.parser] is the object that quantizes note values to a set musical scale. A drop down list of scales is fed to its input, and it quantizes the values of [rtt.notes~] to all snap to notes within the set scale.
- [rtt.notes~] takes in a list of normalized values (floats between 0 and 1) and outputs a list of midi note values. Its size parameter sets the amount of steps within the sequence of notes. This means that if the pattern length is greater than the size of

[rtt.notes~] sequence, the sequence of notes will loop. The length parameter decides the range of notes within the scale. Using higher lengths allows the user to expand the range of pitches, allowing for higher pitched notes to be programmed.

- [rtt.scala~] takes midi note values from [rtt.notes~] and maps them to a musical scale, as sent from the [rtt.scala.parser] object.
- [rtt.makenote~] receives the midi note values as set by [rtt.scala~] and compiles all other midi data into a list per each note. This sets specific midi channels per channel, and provides all the necessary midi data needed to output to a midi compatible device.
- [rtt.makenote~] is routed to a master [midiout] object, an inbuilt Max object that handles the routing of midi messages to send internally or to an external device.
- This entire code snippet was copied and pasted for each channel, with updated sends, receives, and channel numbers based on its corresponding channel.

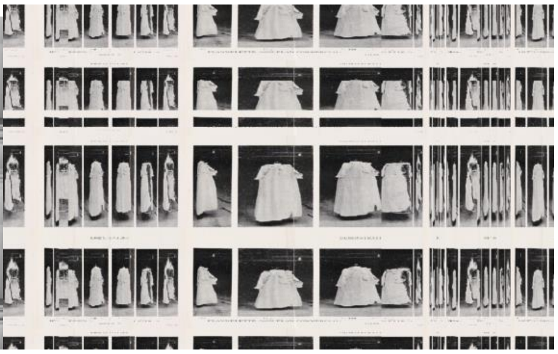
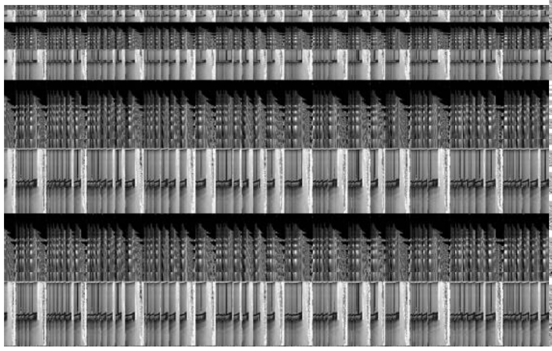
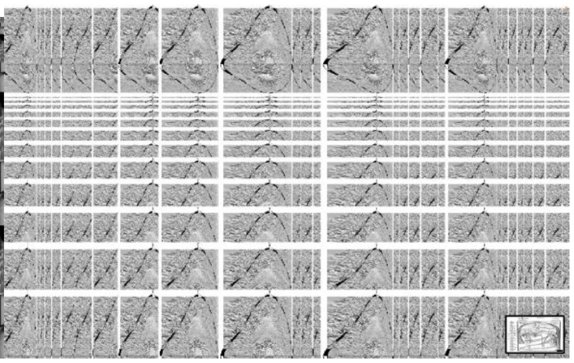
Visual Generation Jitter Patch:



- Code based on Ned Rush's Youtube tutorial on Texture Feedback Processing (https://www.youtube.com/watch?v=PomKrUNqdAk&list=PLerLtHwIDOSV8glmczju3ogcuGeYlhMMr&index=31&ab_channel=NedRush)
- Processes images by creating lines of feedback that scan the image, causing glitchy tears in images that randomly vary in intensity and speed
- Takes impulses from sequencer channels 1, 2, 3, 4 to change feedback processing whenever steps in these channels are triggered

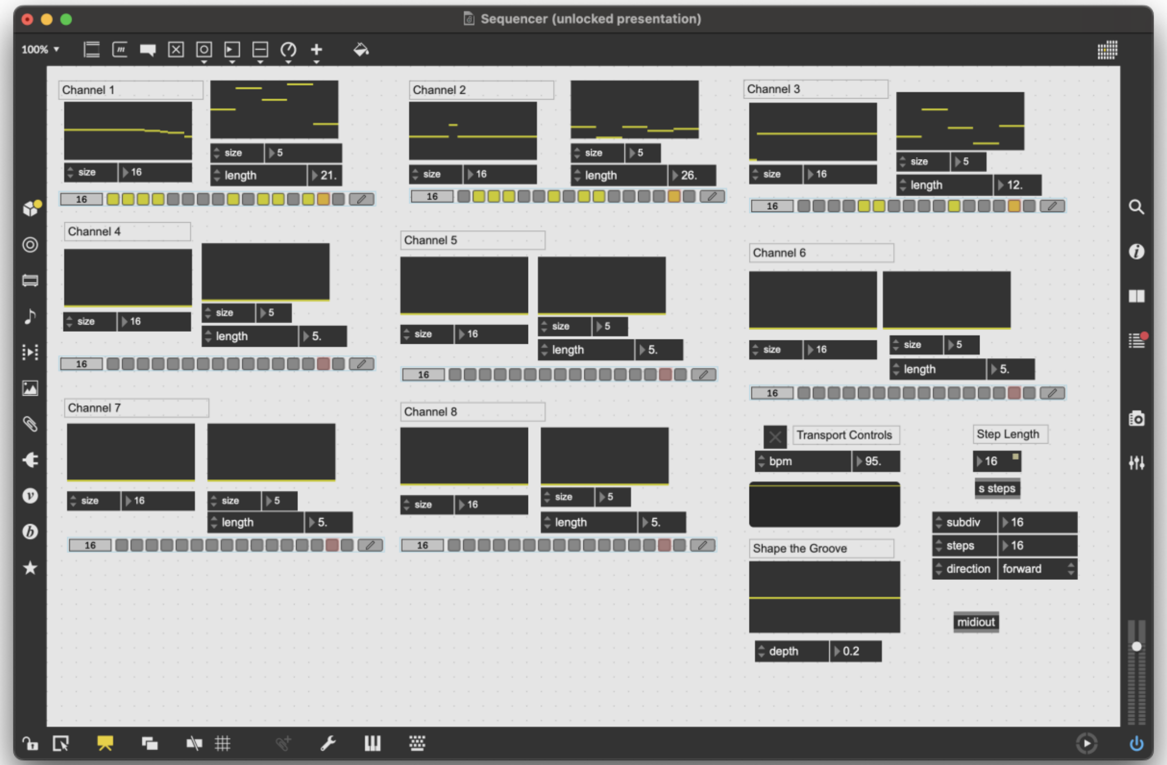
- [jit.world] instantiates the video output window and sends bangs at the same framerate as the video output to smoothly update the output image
- [p pics] abstraction loads a folder of images, and randomly picks one image to output every time channel 3’s sequencer sends out an impulse
- [jit.gl.texture] creates feedback texture map visualized in pwindow object connected to [jit.gl.pics] output. The speed and depth parameters of these lines of feedback are randomly changed when sequencer channels 3 and 4 are triggered, resutling in random numbers being outputted and scaled accordingly to shape the feedback patterns

Examples of Visuals

Implementation in Live Performance

To use this sequencer in live improvisational performances, I connected my laptop via USB to a Digitakt, a digital sampler drum machine that I held responsible for triggering samples and outputting sound. This is the main view of the sequencer patch in presentation view, which is the arrangement I use in performance:



Channel 1
size 16
size 5
length 21.
16

Channel 2
size 16
size 5
length 26.
16

Channel 3
size 16
size 5
length 12.
16

Channel 4
size 16
size 5
length 5.
16

Channel 5
size 16
size 5
length 5.
16

Channel 6
size 16
size 5
length 5.
16

Channel 7
size 16
size 5
length 5.
16

Channel 8
size 16
size 5
length 5.
16

Transport Controls
bpm 95.

Step Length
16
s steps
subdiv 16
steps 16
direction forward

Shape the Groove
depth 0.2

midiout

The left multislider on each channel is where you set the probabilities for the steps to be randomly triggered. In a performance, I would draw a rhythmic contour that I felt would complement the sound it was connected to, and then press its corresponding channel number on the keyboard until I got a pattern that I liked. I could then change what notes would be outputted by using the second multislider, adjusting the length parameter to decide the range of notes the sound would need. This process would then be repeated for each drum track, adding more elements to the arrangement as I went. I could then shape the sounds and apply effects on the fly using the Digitakt’s sound manipulation features, resulting in complex and dynamic sounds without needing to program detailed patterns into the Digitakt‘s sequencer.

Conclusions and Reflections

The random algorithmic sequencer combined with the powerful sound manipulation features of the Digitakt provides for a highly variable organically mutating live compositions of electronic music. One of the best features of the Digitakt is its sequencer, which offers incredible control of many parameters on a step by step basis. However, all these features require time and preparation to create interesting and deep sequences, which is something you would need to do prior to performing live with the machine. My algorithmic sequencer solves this problem by offloading the work of creating complex drum patterns to controlled randomness, allowing the user to be free to manipulate the sampler's powerful sound shaping parameters on the fly while easily creating highly variable sequences.

The visual component of this performance was inspired by Ryoji Ikeda's extreme color palette of black and white, combined with glitch collage aesthetics that represented my implementation of the idea of digital cubism. All images were sourced from public domain archives, compiling images of anatomical diagrams, atlases, anthropological studies, neurological flowcharts, analog technology, and topographical images. I curated these specific images to explore the tension between the human desire to categorize and analyze ourselves as a network of systems, and our creation of machines and networks made in this self-image. I find it fascinating how we as humans have a deep need to understand and document our own bodies and neurological processes, and yet all this knowledge of ourselves is constructed through our own flawed image of ourselves. I decided to play with this idea using diagrams and drawings that try to understand our bodies as a system, disoriented through the digital systems that we have created for ourselves.