Patch Histories and Forking Paths: Version Control as Creative Practice in Modular Synthesis

Michael Palumbo

A Dissertation Submitted to
The Faculty of Graduate Studies
In Partial Fullfilment of the Requirements
for the Degree of
Doctor Of Philosophy

Graduate Program in Digital Media
York University
Toronto, Ontario

July 2025

Abstract

This dissertation addresses a gap in software for multiplayer modular synthesis, demonstrating how version control systems and web technologies are well-suited to closing it.

The rationale rests on two observations. First, a modular synthesizer is, by definition, an interoperable system: individual modules can be recombined, swapped, or even distributed among several performers to create a shared instrument. Yet in practice most hardware and software setups are optimized for a single player, leaving little support for real-time, many-hands interaction.

Second, a defining pleasure of modular synthesis is that the instrument can be rewired on the fly, transforming its topology mid-performance. Capturing those evolutions is notoriously difficult. Hardware musicians rely on photographs or handwritten patch notes, which rarely recreate a state with fidelity. Software modular synthesizers make it possible to save entire patches, but they record only the endpoints—full snapshots—without the in-between: the granular sequence of cable changes, button presses, and knob turns that give a performance its arc.

As these incremental transformations are vital to how the instrument can be played, analyzed, and studied, the field needs tooling that treats patch histories as first-class data, editable and shareable across multiple players in real time.

Drawing on the three established generations of version control systems, this dissertation also posits a fourth generation, defined by the emergence of platform-based social coding and real-time protocols that support live co-editing of documents.

A new musical instrument named *Forking Paths* is introduced to address this gap: a system that integrates software version control architecture into a virtual modular synthesizer to document the entire process of patching—creating patch histories. This capability facilitates new techniques in digital music performance, such as analyzing differences between knob turns across histories, merging gestures, isolating and looping segments of history, and

branching from specific points for further exploration. Moreover, a real-time version control system enables the instrument to support multiple players from the ground up, facilitating participatory and collective patching of a shared modular system.

Keywords: version control systems, digital musical instruments, modular synthesis, distributed creativity

Dedication

I dedicate this dissertation to Jean Buchanan, my grandmother.

Acknowledgements

This dissertation would not exist without the guidance, generosity, and support of many people over many years.

First, to my supervisor Dr. Graham Wakefield: thank you for working on so many of these projects with me, for always walking into the Alice Lab with a cheery and generous greeting of *hellllloooo*, and for reminding me—often—that there is life and a career on the other side of a PhD ("Hey, that's an interesting new idea, why not save it for the postdoc?"). Your encouragement to learn Node.js led directly to many of the projects detailed here; your patience and clarity helped guide our work through its many stages, and your willingness to code alongside me was both welcome and inspiring.

To Dr. Joel Ong, thank you for our many collaborations and for your unwavering support during times when I struggled to see myself as an artist. Your belief in my work—and in me—meant more than I can say.

To Dr. Andreas Kitzmann, Dr. Jason Nolan, and Dr. Kurt Thumlert, thank you for your thoughtful engagement, encouragement around performance and synthesis, and helpful nudges to "get the damn comprehensives done."

To Dr. Doug Van Nort, for welcoming me into the Dispersion Lab, trusting me to implement and manage the lab's GitHub organization (the context in which I first began to see the creative potential of version control systems!), introducing me to the literature on distributed creativity, and your collaboration on the *Git Show* project.

To Michael Longford, thank you for always having my back—and for facilitating my transition from Theatre and Performance Studies into the Digital Media program, especially given that I was joining for its inaugural year.

To the incredible graduate program assistants in the departments of Digital Media, and of Theatre and Performance Studies, who somehow made navigating York's bureaucracy seamless: Yameng Zou, Aishah Mairaj Rashid, Helen Ogundeji, Andrea Di Florio-Sgro, Dawn Burns, Susanna Talanca, and Carianne Hathaway—thank you.

To my parents, Sharon and Ralph, and my sister Kate: thank you for encouraging my decision to pursue an undergraduate degree as a mature student, for your patience and support of my many, many interests over the years, and for your unwavering enthusiasm for proofreading. To my sister-in-law Andi and to the Palumbos, Lippetts, and Buchanans—thank you for being part of the circle that holds me up.

To Bryanna: thank you for being an incredible ear and shoulder, and for holding my insecurities with care. For beach days in the middle of the week, movie nights, walks with Gumbo, being there to keep things fun, and helping me stay grounded throughout the many long months of dissertation writing and Forking Paths development.

To Degan, my therapist: thank you for helping me grow into the person I am. You were there through the hardest years of my life, and have stayed through some of the greatest.

To Nicola, my PhD coach and ADHD strategist: thank you for helping me understand the patterns of self-doubt that slowed my progress, for giving me tools to actually move forward and learn how to get out of my own way.

To the many electronic musicians who playtested Forking Paths and gave such thoughtful, detailed feedback: your insights shaped both the software and this dissertation.

To the artists, audiences, collaborators, and production teams of Exit Points: thank you for trusting me, performing with me, and building something communal and so incredibly special and true to my heart.

To Ezra J. Teboul, Einar Engström, and Andreas Kitzmann, the editors of *Modular Synthesis: Patching Machines and People*—thank you for including the chapter that I co-authored with Graham—*From 'What If?' to 'What Diff?'*—in your groundbreaking book on modular synthesis culture.

To York University: for funding through the Graduate Fellowship, FGS Bursaries, and the W. Lawrence Heisey Graduate Award in Fine Arts.

To CUPE 3903, for extended health benefits that supported my concussion recovery and many therapy sessions throughout this PhD.

To the Helen Carswell Foundation, Community Music Schools of Toronto and Richard Marsella: for supporting my two community music research projects during my time at York.

To Paul Stillwell and Scott Lepore, for always making space for me to come by and talk about this work at Frequency Freaks meetups, and to Stephen Humphrey for documenting my presentation at the April 2025 Frequency Freaks workshop.

To Sarah Greene and The Tranzac, for supporting the Cool New Instruments Night, and for giving me a space to hone how to talk to a general audience about this work.

To my fellow grad students—Aida Khorsandi, Ilze Briede (Kavi), Hrysovalanti Fereniki Maheras, Marcus A. Gordon, Grace Grothaus, Nick Fox-Gieg, Ian Jarvis, Rory Hoy, Kieran Maraj, Omar Shabbar, Denise Rogers Valenzuela, Hannah Rackow, Kathe Gray, Em Piro, Shannon Hughes, Tabia Lau, Signy Lynch, and Thea Fitz-James: you do incredible work and its been a pleasure to work alongside you this past decade.

To my lifelong friends and early bandmates Pouya Hamidi, Maneli Jamal, and Alex Tanas: for always being down to support each other as we have continued to pursue our own distinct and remarkable musical paths, and for encouraging me at times when I had all but given up.

To Dr. Eldad Tsabary, who introduced me to networked music performance and software art during my undergraduate degree in Electroacoustic Music—those early ideas still echo throughout this work.

To Brian Su: thank you for handing me a modular synth when I was at my lowest point as an artist. I didn't know it then, but clearly there was no turning back!

Finally, to the people I haven't named here, but whose presence mattered—thank you.

Preface

This dissertation is submitted as a complex digital thesis consisting of a written manuscript accompanied by a curated archive of software artefacts. Some of these programs remain fully runnable—including the live website Forking Paths (https://forkingpaths.onrender.com)—while others survive only as source-code snapshots, build artefacts, or screen-captures that document earlier, now-deprecated experiments. Taken together, the collection charts an eight-year investigation into how version control concepts can reshape electronic music practice.

Although Forking Paths is the principal case study of this dissertation, the earlier prototypes are essential: they map the conceptual steps that led from initial experiments (e.g., using a software respository during the process of academic writing, a projectional editor, a VR synthesizer) to the current browser-based modular synthesis environment. Where those legacy projects depend on larger, now-defunct or unmaintained systems, they are preserved as read-only repositories and screenshots. Their scholarly value lies in the design decisions, conceptual investigations, and technical problems they reveal, not in whether they can still be run today.

The written manuscript performs three complementary roles:

- **Contextualisation** — Chapters 1 & 2 locate the research within version control systems, distributed creativity, modular synthesis, computer music, and web-based musical instruments.
- **Documentation** — Chapters 3 & 4 trace the iterative design path from the earliest experiments to the final architecture of *Forking Paths*.
- **Evaluation** — Chapter 5 analyses play-test data and public performances, and synthesizes the project's contributions and outlines future work.

- **Conclusion** — Chapter 6 distills the project's contributions: it defines *Forking Paths* as a version-controlled modular synth, notes present limits (two-user cap, split UI, single-threaded server), and outlines next steps to decouple the VCS core, unify the interface, and scale to ensemble-size sessions.

As Chapters 3 and 4 cite code listings and screenshots, the following inventory of digital items specifies every software artefact, notes whether it is runnable or archival, and cross-references the chapter sections where it appears. Readers are encouraged to explore the Forking Paths live site — https://forkingpaths.onrender.com — before (or while) engaging with Chapter 4 and 5, as firsthand interaction renders the subsequent technical discussions far more concrete. For projects that no longer compile, consult the screenshots in Chapter 3 and the read-only source snapshots included as Digital Item B.

Table 1: Digital & Multimodal Components

Label	Location / Access	Description
Digital Item A	https://forkingpaths.onrender.com Source: https://github.com/michaelpalumbo/ forkingpaths/tree/archive/dissertationJune	The live site is the primary artefact; The snapshot preserves the full source, and developers may rebuild and launch it locally using Vite
Digital Item B	https://github.com/michaelpalumbo/self-referent https://github.com/michaelpalumbo /spth6155 https://dispersionlab.github.io/gitshow https://github.com/dispersionlab/gitshow /wiki https://github.com/worldmaking/alicenode /tree/master/client https://github.com/worldmaking/cards /wiki/List-of-Operational-Transforms https://github.com/worldmaking/vr-in-vr https://github.com/worldmaking/mischmasch	Source repositories analysed in Chapter 3; archived read-only copies ensure that code listings and design-rationale claims remain verifiable.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation addresses an existing gap in software designed for multiplayer modular synthesis, and demonstrates how version control systems (VCS) and web-based technologies effectively meet this challenge. The motivation for this work arises from two primary observations. Firstly, music-making is predominantly communal and participatory, yet contemporary modular synthesis software and hardware are predominantly designed for solitary players. Secondly, one of the most characteristic features of modular synthesis is that it enables musicians to alter instrument configurations dynamically through patch connections, but effectively documenting these modifications remains challenging. Hardware modular synthesizers often rely on ancillary documentation such as written notes or photographs, methods that inadequately capture precise states. Software solutions such as VCV Rack (Belt 2017) permit saving patch states, yet these project files fail to record incremental changes and gestural details, such as live patch reconfigurations or specific knob adjustments. While digital audio workstations typically handle recorded gestures as automation, this functionality is largely absent in modular synthesis tools. Consequently, the absence of incremental documentation complicates tasks essential to musical improvisation, analysis, and scholarly study, and limits the ability for multiple musicians to meaningfully collaborate, revisit, or build upon each other's work in a shared modular environment.

In response, this dissertation proposes integrating VCS capabilities directly into a virtual modular synthesizer environment. Forking Paths, the system developed through this research, captures real-time transformations of patch states and parameter controls, meticulously documenting the entire patching process and creating comprehensive patch histories. This integration allows musicians to analyze detailed differences between knob adjustments, merge gestures, isolate and loop history segments, and branch from specific points for further exploration. Moreover, Forking Paths is inherently multiplayer, fostering participatory and

communal music-making from its core design. Utilizing network technologies and module interoperability, the system enables collaborative ensemble improvisation, overcoming typical limitations associated with single-user desktop software.

While this dissertation makes a case for the creative potential of version control in modular synthesis, it does not assume that all artists are actively seeking such systems. Rather, it responds to a persistent but under-articulated challenge: the lack of visibility into process and change over time. Informal documentation practices—such as photographing patch cables, writing down knob positions, or, in the case of software synthesis, manually saving multiple versions of a patch—suggest a tacit desire for more robust support. Feedback from playtests and public demonstrations conducted during this research confirms that artists often wish to revisit earlier states.

This research explores how integrating VCS into modular synthesis software can support collaborative improvisation, real-time gestural documentation, and creative reuse of patch histories. The central guiding hypothesis is that embedding musical creativity within a version-controlled environment enables new modes of musical memory, collaboration, and structural articulation in improvisational contexts.

Technically, Forking Paths introduces a novel version-controlled environment for modular synthesis, enabling real-time branching, merging, and recall of patch states and parameter gestures. A history-based sequencer allows performers to structure and remix prior states as compositional material, while a gesture editor supports fine-grained transformation and playback of modulations over time. These tools are supported by a custom backend that integrates a VCS directly into the audio engine, making structural editing and collaborative performance fluid and responsive. The system also introduces multiple modes of multiplayer collaboration, allowing performers to explore different models of shared authorship and temporal control.

Forking Paths adds VCS as a central component of software modular synthesis in performance and composition, treating patch histories as navigable pathways. This foregrounds distributed creativity, positioning software as a dynamic mediator between players and their histories.

Methodologically, the dissertation employs an iterative research-creation trajectory, refining concepts through a series of experimental prototypes: *Using Git to Write a Paper*

About Using Git[1], *Git Show*[2], *Alicenode Editor*[3], *Cards*[4], *Graph Operational Transform*[5], *Modular Synthesis in Virtual Reality*[6], and *Mischmasch*[7]. Each prototype illuminates distinct facets of versioning in creative software, providing foundational insights that were integrated into Forking Paths.

The remainder of this dissertation is structured as follows. Chapter 2 reviews the literature on distributed creativity, software studies, and theoretical frameworks concerning time, creativity, and authorship. It examines relevant developments in modular and networked digital musical instruments and provides a historical overview of VCS. Chapter 3 presents the methodology of this work through several research-creation studies, each of which shaped the conceptual and technical foundations of Forking Paths. Chapter 4 details the interface, design rationale, motivations, and technical implementation of the Forking Paths system. Finally, Chapter 5 presents the formative evaluation of Forking Paths through public salons, as well as playtesting, and concludes by reflecting on broader implications, limitations, and avenues for future research.

[1]Early experiments with VCS began in text-based academic writing, where a versioning system was used as a creative tool to both write an essay and contemporaneously reflect on its development.

[2]Git Show was a research-creation project that introduced VCS to a group of composers.

[3]The Alicenode Editor was a browser-based tool for remotely updating an interactive artwork in real time, with an interface for navigating different file versions.

[4]Cards was a prototype for editing code through multiple structural views.

[5]Graph Operational Transform (GOT) was an editing system that allowed real-time collaboration on modular synthesizer graphs by tracking and synchronizing structural changes.

[6]Modular Synthesis in VR (MSVR) combined multiplayer VR and modular synthesis to explore shared, spatialized sound-making, with real-time responsiveness and versioned interaction.

[7]Mischmasch emerged as a ground-up rewrite of MSVR in Node.js, integrating graphics, sound, and input systems into a native environment to better support collaborative modular synthesis.

Chapter 2

Literature Review

The literature review addresses thematic areas central to this dissertation, including: distributed creativity, web-based audio and visual tools, networked creativity, VCS and state management systems, collaborative performance practices with modular synthesizers, software-based and mixed-reality musical instruments (VRMIs and MRMIs), visual programming languages, live-coding, and improvisation practices, and theories related to instrument design and collaborative creativity.

2.1 Distributed Creativity

Contemporary theories of human activity recognize that action does not arise solely from individuals, but from interactions between people, tools, and environments. The term sociomaterial refers to this entanglement of social and material elements in shaping what people do (Orlikowski and Scott 2008). It emphasizes that people, technologies, environments, and infrastructures do not act in isolation, but continually influence and constitute each other. Activity is understood as situated, which means that it is shaped by the specific social, spatial, temporal, and material conditions in which it takes place (Lave and Wenger 1991; Dourish 2004).

Sociomaterialist thinking has been influenced by the theories of the extended mind (Clark and Chalmers 1998) and of distributed cognition (E. Hutchins 2006). The theory of the extended mind argues that mental activity does not reside solely within the brain, but is supported by tools and external representations—such as notebooks, maps, or software—which function as integral parts of an individual's cognitive process. Distributed cognition emphasizes both the integration of tools into individual reasoning, and the interpersonal, environmental and systemic conditions in which the individual is situated.

Hutchins provides an example of the navigation practices of naval crews and ships, to demonstrate how processes like reasoning, memory, and problem-solving emerge through the coordination of people, physical environments, representational media, and historical routines

Creativity, too, is always situated, scaffolded, and mediated, and emerges through an ecology of temporal, material, and relational processes. Glăveanu (2014) emphasizes the link between culture and creativity, proposing a distributed, systemic model in which creative outcomes emerge through dynamic interrelations between individuals, artifacts, norms, and institutions. This sociomaterial view of creativity treats it as extended, emerging from entanglements of human and nonhuman actors (Suchman 2007; Ingold 2013).

Sawyer and DeZutter (2009) articulate the concept of distributed creativity to describe how creative outcomes emerge from networks of participants, practices, and artifacts—whether through co-present collaboration or asynchronous, artifact-mediated exchange. This perspective shifts attention away from individual authorship and toward what Sawyer and DeZutter call collaborative emergence: improvisatory and open-ended processes in which outcomes arise through mutual responsiveness rather than centralized control.

2.2 Version Control Systems

Version control systems (VCS) are tools designed to track changes to files over time, allowing users to record, revisit, and coordinate modifications across different versions of a project (Chacon and Straub 2014). They are foundational to collaborative and iterative work, particularly in software development, where multiple contributors may be working simultaneously on the same components (Estublier 2000). At their core, VCS provide mechanisms for saving historical states, comparing revisions, and experimenting with new directions without risking the loss of previous work (Mens 2002). Over time, these systems have evolved from simple tools for managing individual files into sophisticated frameworks that support complex collaboration across teams and time (Spinellis 2005).

In computing, a delta refers to the difference between two states of a file or dataset—a minimal representation of what has changed. Rather than storing every version in full, many systems store only these deltas, which can then be applied in series to reconstruct earlier or later states. The concept originated in early work on text comparison and file differencing (Hunt and MacIlroy 1976), where algorithms were developed to compute minimal edits between sequences. As a unit of change, the delta became foundational to later systems

concerned with storage efficiency, revision tracking, and historical comparison (Hunt and MacIlroy 1976; Tichy 1985; Burrows 1994; Estublier 2000).

When applied systematically, deltas form the basis of a model of change, in which an evolving file or structure is represented as a sequence of operations or transformations (Nugroho, Hata, and Matsumoto 2020). These deltas may be stored in forward or reverse order, as in Revision Control System (RCS) (Tichy 1985); interleaved within a single file, as in Source Code Control System (SCCS) (Rochkind 1975); or structured using content-addressed storage and directed acyclic graphs (DAG), as in Git (Chacon and Straub 2014). These strategies reflect differing system priorities—whether fast access to the latest version, efficient compression, or support for nonlinear development (Spinellis 2005). Across these implementations, the underlying principle remains consistent: to represent transformation over time in a form that can be navigated, recombined, or rolled back.

2.2.1 First Generation (1972–1982): Local, File-Centric Version Control

The earliest VCS were developed to manage individual files on standalone machines, with no support for broader project-wide coordination. Notable examples include SCCS and RCS. SCCS, introduced in 1972 at Bell Labs by Marc Rochkind, stored all revisions of a file in a single location using interleaved deltas (Rochkind 1975). RCS, introduced a decade later by Walter Tichy, extended SCCS by adopting reverse deltas, which optimized retrieval of the most recent version (Tichy 1985). Both systems treated each file as an isolated object, with no native support for tracking relationships across files. While they offered limited support for diverging development paths, this was expressed through increasingly complex numeric hierarchies (e.g., 1.2.1), highlighting the constraints of a file-centric model in the context of multi-file or collaborative projects (Estublier 2000; Spinellis 2005).

2.2.2 Second Generation (1982–2003): Centralized, Project-Oriented Version Control

As software systems grew in complexity, the limitations of file-centric version control became increasingly apparent. Larger codebases, expanding development teams, and the need to coordinate changes across multiple files led to the emergence of centralized, client-server systems. A key innovation of this generation was the commit: a mechanism for recording a coherent set of deltas—small, file-level changes—grouped into a single project-wide operation. Unlike earlier systems, where each delta was applied and tracked independently

per file, commits allowed developers to bundle multiple modifications together, typically annotated with a message and timestamp.

The commit message functions as a kind of marginalia—a note left at the edge of a process, marking intention, interpretation, or transition within an ongoing system of change. They operate as paratextual gestures: interpretive inscriptions layered alongside procedural operations. Like marginal notes in a manuscript, they are pragmatic, contingent and carry traces of authorship within a project's structure. Crucially, commits are not only intended for collaborators but for one's future self—a way of returning to an idea after forgetting its rationale, or of recovering a direction that might otherwise be lost.

This shift enabled a higher-level understanding of how a system was evolving: not just which lines of code had changed, but what those changes represented in relation to the broader project. The recursive function of the commit is especially important for later discussions on musical creativity, particularly in composition, where the ability to retrace, reinterpret, and recontextualize prior decisions becomes part of the creative process itself.

Version control in practice rarely conforms to the clean, disciplined timelines idealized in software engineering. While deltas are typically generated by machines to encode changes at the level of code or structure, commits are authored by humans. They mark a decision made about when to record change, which changes to include, and how to interpret or describe them.

Second-generation tools also introduced the concept of a shared repository—a central database that maintained the complete history of all project files. CVS (Cederqvist, Pesch, et al. 1992) extended the earlier RCS model by adding rudimentary commit tracking and networked collaboration, though it lacked atomicity. ClearCase (Galfullin and Novichkov 2000) and Perforce (Seiwald 1995) provided more robust models for large-scale industrial software development, including improved branching and access control. Subversion (SVN), launched in 2000, explicitly addressed CVS's shortcomings by implementing atomic commits, support for binary files, and a model for branching and tagging (Pilato, Collins-Sussman, and Fitzpatrick 2008). While branching was still considered risky or advanced, these systems marked a shift from managing isolated file revisions to orchestrating coordinated, project-wide change (Estublier 2000; Spinellis 2005).

2.2.3 Third Generation (2003–present): Distributed Version Control

The third generation of VCS introduced a shift from centralized coordination to fully distributed models, in which the complete project history is replicated on every contributor's

machine. This eliminated reliance on central servers for most operations and enabled flexible, offline workflows (Chacon and Straub 2014). Distribution introduced a new temporal and structural logic: multiple contributors could now diverge, experiment, and combine changes without a fixed center of authority.

A key innovation of the third generation was the refinement and normalization of branching: the creation of alternate development paths within a VCS—parallel lines of modification that diverge from a common history (Chacon and Straub 2014). While branching was technically possible in some earlier systems, it was often cumbersome or discouraged (Estublier 2000). Third-generation VCS transformed branching into a lightweight, everyday operation—enabling developers to explore new features, test ideas, or fix issues without altering the main codebase (Spinellis 2005). Once complete, these paths can be reintegrated, discarded, or maintained as independent trajectories (Mens 2002). Branches make the version history nonlinear, introducing a temporal logic that supports experimentation, contingency, and multiplicity. As branching became central to development workflows, it also reshaped how developers conceptualize time, authorship, and the evolving structure of a project (Estublier 2000; Spinellis 2005; Kalliamvakou et al. 2014). These branches can either be merged back into the main work or left as separate ideas, serving as abandoned experiments or new creative paths. This functionality offers observers valuable insights into a project's development and provides current and future collaborators with tools to engage with ideas in their various stages of evolution.

Merging is the complementary operation to branching, enabling divergent development paths to be reconciled into a unified commit (Mens 2002). Early support for merging in second-generation VCS introduced the ability to integrate changes made on separate branches, though this often required manual intervention and was prone to error (Tichy 1985; Estublier 2000; Pilato, Collins-Sussman, and Fitzpatrick 2008). Over time, merging became a core operation across many third-generation systems, supported by increasingly sophisticated tools for detecting and resolving conflicts between parallel edits (Mens 2002; Bird et al. 2008). Crucially—and consistent with a central tenet of version control—merges record the convergence of multiple histories without overwriting them, maintaining traceability and accountability across revisions (Spinellis 2005).

One of the first systems to propose a distributed model of version control was BitKeeper, developed by Larry McVoy in 2000 (BitMover Inc. 2000). It introduced a hybrid architecture that allowed developers to work on local copies of repositories and later synchronize changes with a central server, and supported distributed branching and merging. BitKeeper

was briefly adopted by the Linux kernel project due to its performance and suitability for managing large, distributed development workflows. However, BitKeeper's proprietary licensing and limited flexibility exposed the need for a fully open, decentralized alternative.

Git (Torvalds, Hamano, et al. 2005) emerged in response to the practical challenges of coordinating development in large, distributed projects like the Linux kernel, where changes had long been shared as emailed patch files.[1] Linus Torvalds designed Git in 2005 to build on that decentralized workflow while addressing its limitations—formalizing a model that supported fast branching, decentralized history, a storage system based on SHA-1 hashes, and a DAG (Torvalds, Hamano, et al. 2005; Loeliger and McCullough 2012).

2.2.4 Fourth Generation: Social Networks and Real-Time Versioning

Building on the established historical framing of VCS as evolving through three generations—local file-based systems, centralized project-based systems, and distributed version control (Estublier 2000; Spinellis 2005) — I identify a fourth generation, characterized by shifts in how version control operates as an interface for collaboration, visibility, and authorship. This generation comprises two distinct and, in some ways, opposing trajectories: one rooted in the social and platform-based infrastructures built around distributed VCS, exemplified by GitHub (Dabbish et al. 2012; Tsay, Dabbish, and James Herbsleb 2014); the other in the development of real-time collaborative systems based on operational transforms (OT) and conflict-free replicated data types (CRDTs) (Sun and Ellis 1998; Kleppmann and Beresford 2017). While both of these trajectories have been active for several decades—GitHub since 2008, OTs since the 1990s—viewing them together as a distinct phase allows us to account for how version control has come to shape, and be shaped by, broader cultural, temporal, and economic forces (Kleppmann, A. Wiggins, et al. 2019).

Socially Networked VCS

The first trajectory involves the rise of socially networked VCS, particularly through platforms like GitHub (GitHub Inc. 2008), GitLab (GitLab Inc. 2011), and SourceHut (DeVault 2018), which are built atop the distributed architecture of Git (Chacon and Straub 2014; Loeliger and McCullough 2012). These platforms retain Git's core mechanics but layer them with mechanisms of coordination, review, and governance (Dabbish et al. 2012; Tsay, Dabbish, and James Herbsleb 2014). Version control becomes embedded within web interfaces that go beyond the management of code to include but issues threads, discussions, contribution workflows, and project visibility (Zagalsky et al. 2015). In these environments,

participation is not limited to those with repository write access. Instead, collaboration is structured through a series of highly visible, formalized interactions (Marlow, Dabbish, and Jim Herbsleb 2013).

Central to this model are platform-specific features like cloning, forking, and the pull request. Cloning allows any user to copy and inspect the full contents and history of a public repository without requiring permission or affiliation. Forking builds on this by creating a personal, modifiable copy of the repository under the user's own account. The pull request then becomes a means by which that user can propose changes to the contents of the original repository. These mechanisms enable a form of participation where contributors can engage from outside the project's formal boundaries—without prior contact or endorsement from maintainers (Gousios, Pinzger, and Deursen 2014). In doing so, they enact GitHub's broader logic of openness: contributions are technically open to all, but subject to review, curation, and approval (Nafus 2012; Vasilescu, Serebrenik, and Filkov 2015). Pull requests also formalize the integration of these temporal streams, documenting external contributions while preserving the historical context of changes (Treude and Storey 2010). By situating contributions within the artifact's evolving narrative, they make collaboration itself an integral part of the creative process.

As a result, GitHub repositories move beyond tools for collaboration to function as public spaces of authorship, recognition, and social capital (Eghbal 2016; Dabbish et al. 2012). Metrics like stars, forks, contribution graphs, and merged pull requests operate as proxies for value and influence (Marlow, Dabbish, and Jim Herbsleb 2013; Vasilescu, Serebrenik, and Filkov 2015). Contributors accrue visibility by participating in highly watched projects; maintainers gain status by overseeing successful or widely-used codebases (Zagalsky et al. 2015). In this way, version control becomes a form of platform labor—one that ties participation to recognition, algorithmic visibility, and social capital (Kelty 2008). The repository now becomes an interface for signaling identity, credibility, and alignment within open-source and professional ecosystems (Nafus 2012).

Real-Time VCS

The second trajectory of fourth-generation VCS centers on real-time collaboration: a mode of version control in which users simultaneously co-edit shared documents or codebases, with changes merged continuously in the background. Systems built on Operational Tranforms (OT) (such as with Google Docs) or conflict-free replicated datatypes (CRDT) (as implemented in frameworks like Automerge (Automerge 2017) or Yjs (Jahns 2017) abandon

the linear sequence of commits in favor of a fluid, causally consistent shared state (Sun and Ellis 1998; Kleppmann and Beresford 2017). They automatically perform the equivalent of diff, merge, and rebase operations in the background, foregrounding the current shared state rather than past revisions. While these systems do preserve internal histories of operations, they typically abstract them away from the user interface. Unlike earlier VCS, which foreground a sequential record of changes, these tools prioritize concurrency: handling edits from multiple users at the same time and quietly resolving them into a unified shared state (Kleppmann, A. Wiggins, et al. 2019).

An essential consistency requirement in OT systems is causality preservation—the guarantee that operations are executed in the same causal order across all users. (Sun and D. Chen 2002). If *operation A* causally precedes *operation B* (for example, *B* modifies the content created by *A*), then all users must see and apply *A* before *B*. This ensures that the logical relationships between edits are maintained, avoiding incoherent or conflicting document states. This principle draws on Lamport's happened-before relation (Massachusetts Computer Associates, Inc. and Lamport 2019) and is foundational to preserving consistency in real-time collaborative environments.

The emphasis on real-time presence and shared state in OT- and CRDT-based systems has also inspired a broader movement toward local-first software design. Frameworks like Automerge, which implements CRDTs for JSON-like data structures, enable peer-to-peer synchronization without server dependency—supporting offline editing and real-time collaboration, with eventual consistency across distributed clients.

Azurite is a selective undo plugin for the Eclipse IDE that brings fine-grained, interactive history management into the programmer's live editing environment (Yoon and B. A. Myers 2015; Yoon, B. A. Myers, and Koo 2013). Whereas traditional version control systems (VCS) such as Git are optimized for committing logical units of work, supporting collaboration, and managing branches across longer timelines, they typically operate only once code has reached a stable checkpoint. By contrast, Azurite intervenes at the micro-level of the creative process, capturing every edit in real time and enabling selective undo, semantic zooming, and contextual history visualization. This approach reflects a broader trend in research on creative version control systems (CVCS), which emphasizes: (1) greater granularity, shifting from coarse commits toward capturing fine-grained edits and ideas; (2) selectivity, allowing practitioners to revert or reuse specific changes without rolling back the entire project state; and (3) visual navigation, presenting history as a navigable, explorable space rather than a linear log. Such features suggest the potential for extending VCS design into other creative

domains—writing, music composition, visual art—where practitioners similarly require the ability to explore, revert, and recombine micro-level variations without compromising the integrity of the larger work.

Kleppmann, A. Wiggins, et al. (2019) expand on these principles through the framework of local-first software, which seeks to ensure that users retain access to their work even when network connectivity is unreliable or unavailable. While the language of local-first design often emphasizes individual autonomy and data ownership, its relevance to collaborative music systems lies in its promise of resilience: the ability for performers to remain connected to the evolving musical state—and to each other—even in the face of latency, dropouts, or infrastructural failure. In this context, VCS becomes a site of infrastructural adaptation, reimagining the relationship between users, data, and networks in ways that prioritize continuity, trust, and shared presence during live performance.

As VCS have evolved from file-based tracking to project-wide commits to fully distributed architectures—and now toward real-time collaboration and platform-based participation—they have enabled more complex forms of software collaboration and introduced new ways of structuring the record of change over time. Their ability to capture divergent paths, embed reflection, and preserve historical context makes them uniquely suited to practices that extend beyond software engineering.

Each commit, branch, or merge traces an alternate possibility, a lattice of co-existing futures that can be revisited, combined, or ignored. Before turning to concrete musical applications, it is helpful to examine version control in terms of the temporal imagination it makes tangible. The next section moves from software history to cultural theory, reading versioning through Borges's iconic labyrinth of simultaneous outcomes and other accounts of nonlinear time to explore how branching infrastructures can underwrite experimental music-making, research-creation, and other forms of distributed authorship.

2.3 Version Control and the Temporal Structure of Creative Practice

Jorge Luis Borges' short story The Garden of Forking Paths Borges (1964, p. 19) has been widely cited in literary theory (McHale 2012), philosophy (Deleuze 1994) and digital media studies (Aarseth 1995; Hayles 2002) for its early articulation of nonlinear temporality and multiplicity. The story introduces a fictional novel that operates as a labyrinth in time, where all possible outcomes of the protagonist Ts'ui Pên's decision coexist in parallel, diverging

and converging paths. Rather than presenting time as a singular, linear progression, Borges writes: "In all fiction, when [someone][1] is faced with alternatives [they] choose one at the expense of the others. In the almost unfathomable Ts'ui Pên, he chooses—simultaneously—all of them" (Borges 1964, p. 26). This has since been taken up as a metaphor for hypertextual narrative structures, branching logics in digital systems, and the aesthetics of multivalence in media art. By foregrounding the conceptual and formal possibilities of simultaneous futures, Borges offers a powerful literary precedent for thinking about distributed creativity, speculative design, and choice in temporal media systems.

VCS allow creators to navigate their work nonlinearly, but their architecture diverges sharply from Deleuze and Guattari's concept of the rhizome (Deleuze and Guattari 1987). Each commit—like a page in Borges's Garden of Forking Paths—advances a new possible future without overwriting the past. The resulting structure is a rooted, directed acyclic graph (DAG): most commits point to a single parent, while merge commits may reference two, briefly entwining separate lines of development. Yet every path remains anchored to a common root and unfolds in a single temporal direction; branches are typically merged back into a dominant lineage. VCS thus preserve hierarchy and linear authorship in modular form, supporting divergence only within an ordered system of inheritance—distinct from the acentred, any-to-any proliferation that defines rhizomatic structures.

The *use* of a version graph, however, can approach rhizomatic behavior. Since any commit can be revisited, branched, or merged into another point in history, authors may create lateral linkages that the underlying hierarchy does not enforce. In practice, this enables a kind of operational web, where "any point connects to any other" (Deleuze and Guattari 1987, p. 5), enacted not in the structure itself but in the performance of forking, recombining, and traversing versions. Rhizomatic behaviour, then, emerges from the user's creative activity within—and sometimes against—the constraints of the DAG.

Still, while the rhizome helps account for how users may hop sideways among commits, it does not explain how the graph itself accrues a broader temporal coherence. For that, a richer understanding emerges through Alfred Gell's concept of the oeuvre.

Gell (1998) introduces the oeuvre as a temporal framework for understanding the development of an artist's practice over their lifetime. Gell explicitly draws on Husserl's model of time-consciousnes (Husserl et al. 1964)—particularly the interplay of retention (the

[1]Square brackets have been used to replace gendered language from the original translation of Borges' The Garden of Forking Paths with gender-neutral pronouns.

just-past) and protention (the about-to-happen)—but applies it at a vastly different scale. Whereas Husserl theorizes time as a continuous flow of lived experience, Gell translates this into the segmented, material register of artworks unfolding over a life. Rather than viewing each artwork in isolation, Gell sees the oeuvre as a distributed object, where individual works function as indices that condition other works—whether earlier (retention) or later (protention). These indices establish a relational network that spans across the artist's career, offering a way to trace stylistic evolution, thematic recurrence, and technical experimentation.

His model foregrounds how temporal structures can be mediated and externalized through artifacts, while also marking a shift from phenomenological continuity to a more punctuated, indexical mode of time. Importantly, Gell's focus is not on the artist's biography or interiority, but on how artworks operate socially—as material agents that index other moments in time and communicate a sense of agency to their audience.

Gell's analysis of Marcel Duchamp's *Réseaux des stoppages* (Network of Stoppages)[2] (Duchamp 1914) provides a striking analogy for thinking about the temporal structure of creative work. The piece layers multiple sketches and conceptual drafts, including a rough version of *Le Grand Verre* (The Large Glass) (Duchamp 1923), and introduces arcs and "stops" branching out from a common origin (see figure 2.1). According to Gell,

> the Network of Stoppages can be read as painterly autobiography. . . . [and] implies that each painting, each phase in the construction, through many preliminary studies, of a major work such as the Large Glass—which itself can be seen as a 'study' for Duchamp's final masterpiece Given the Waterfall and the Illuminating Gas (1948)—can in fact be seen as a 'stop'. . . . It is as if the forms we see in this painting have been temporarily captured and stabilized in their glassy medium like very long-delayed insects trapped in amber, which might still resume their flight one day. (Gell 2015, p. 100)

A stop in Duchamp's *Réseaux des Stoppages* is what Gell describes as a moment of "frozen chance"—an inscription that emerges from a process structured by both indeterminacy and formal constraint. These marks are not necessarily reflective choices, but rather procedural arrests, capturing the path of falling threads and solidifying them into visual

[2]Gell refers to the work by its English translation, *Network of Stoppages*, though Duchamp originally titled it *Réseaux des stoppages*.

form. Their function is to mark a point within an ongoing logic of experimentation and redirection.

Commits in VCS share a similar ambiguity of authorship. While they can reflect moments of deliberate intervention, such as a save point chosen to capture a meaningful state, they can also be arbitrary or incidental, dictated by habit, context, or automation. In OT systems or tools like Automerge, (Automerge 2017) changes are stored in the version history without user intent at all. Like Duchamp's stops, commits operate within a tension between spontaneity and structure, but their mutability and procedural logic differ: commits encode both a state and a logic for revisiting, merging, or adapting. In this sense, both stops and commits act as partial inscriptions, marks of process rather than declarations of finality, but they differ in their degree of intentional authorship and their future affordances.

Gell suggests that Duchamp's stops retain potential, but they are not reactivated in any literal sense. They exist as indices: capturing a form, suggesting a direction, or anchoring an idea that may later influence new work. Their presence on the canvas is final, even as their meaning remains open-ended. In contrast, commits in VCS are structured explicitly for reanimation. Though immutable in themselves, they can be checked out, branched, merged, or even used to generate entirely new trajectories of development.

This distinction highlights a key difference: where stops preserve a frozen moment for reflection or inspiration, commits also encode a mechanism for operational return. They do not merely hint at a possible future, they can be used to make one. This computational reproducibility allows commits to serve as more than static markers, where they function as structurally embedded memory, enabling earlier states to be reintroduced into the workflow without disrupting the continuity of the system.

In the *Réseaux des Stoppages*, every stop is spatially visible and embedded within the overall visual field. The viewer encounters them all at once, distributed across a flattened but conceptually layered surface. Their presence is inseparable from the work's material composition. In VCS, by contrast, commits are hidden from view by default. They must be summoned through log commands or visualizations—abstracted from the working interface, accessed through retrospection rather than coexistence.

This difference introduces a distinct kind of temporality. While Duchamp's stops are ever-present, commits reside in a kind of procedural memory, accessible but recessed. This opacity is not necessarily a drawback—it underscores the idea that memory in computational systems is not continuous but indexed, called into being as needed. Rather than

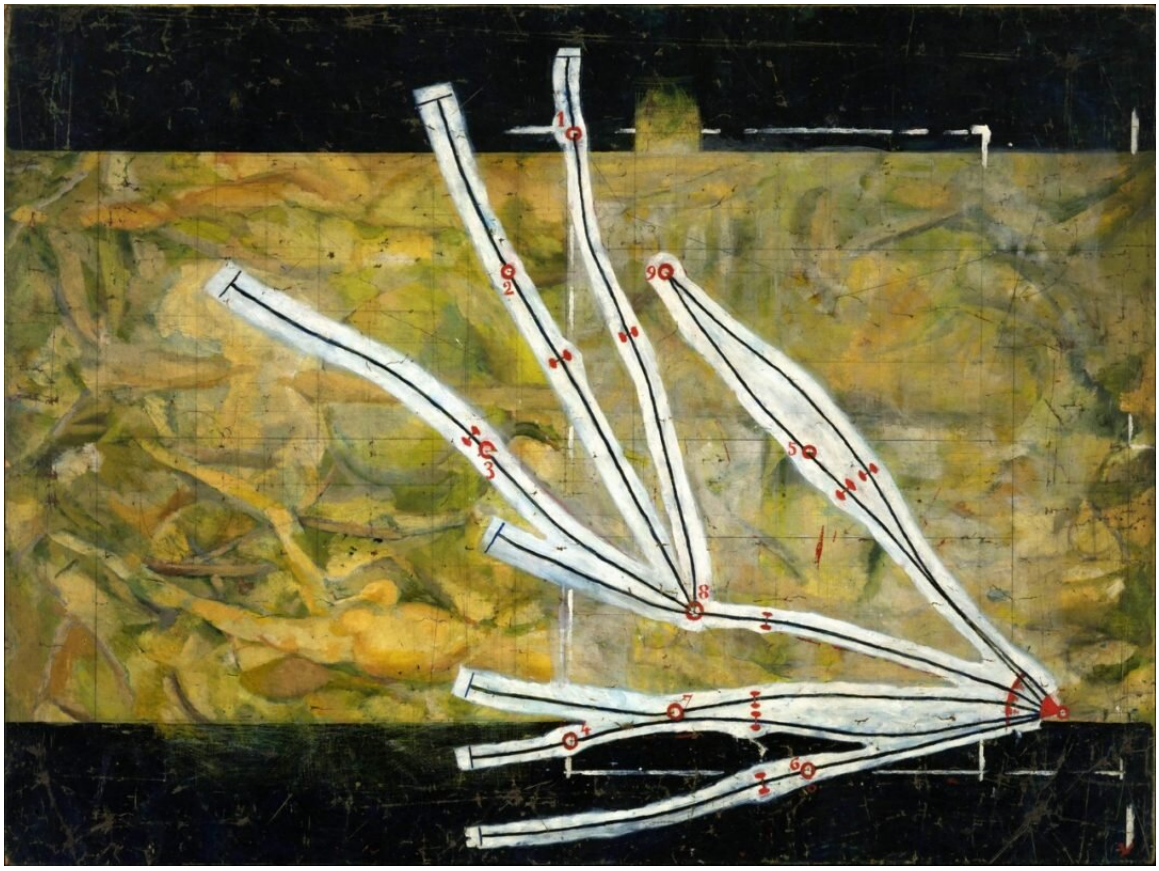


Figure 2.1: Marcel Duchamp, Réseaux des Stoppages, Duchamp (1914). Oil and pencil on canvas, 105.2 × 99.4 cm. Philadelphia Museum of Art. Used under fair dealing for purposes of research and criticism.

being persistently visible, commits require an act of recontextualization to be reengaged, foregrounding the selective and situated nature of recall within VCS.

Duchamp's network can be read as a DAG: it has nodes, branches, and merges, and presents a nonlinear visual structure that hints at multiple timelines or paths of development (see Figure 2.2). But this structure is ultimately frozen in place—visible, interpretable, but not executable. In contrast, VCS put this branching logic to practical use. They allow users to create, navigate, and modify multiple parallel versions of a project, making it possible to revisit or combine different paths over time. This structural difference is crucial. Duchamp's painting both reflects and enacts a logic of multiplicity and divergence, but once completed, it offers no built-in means for further transformation or traceable revision. Any change would overwrite rather than extend its structure. VCS, by contrast, are designed to support

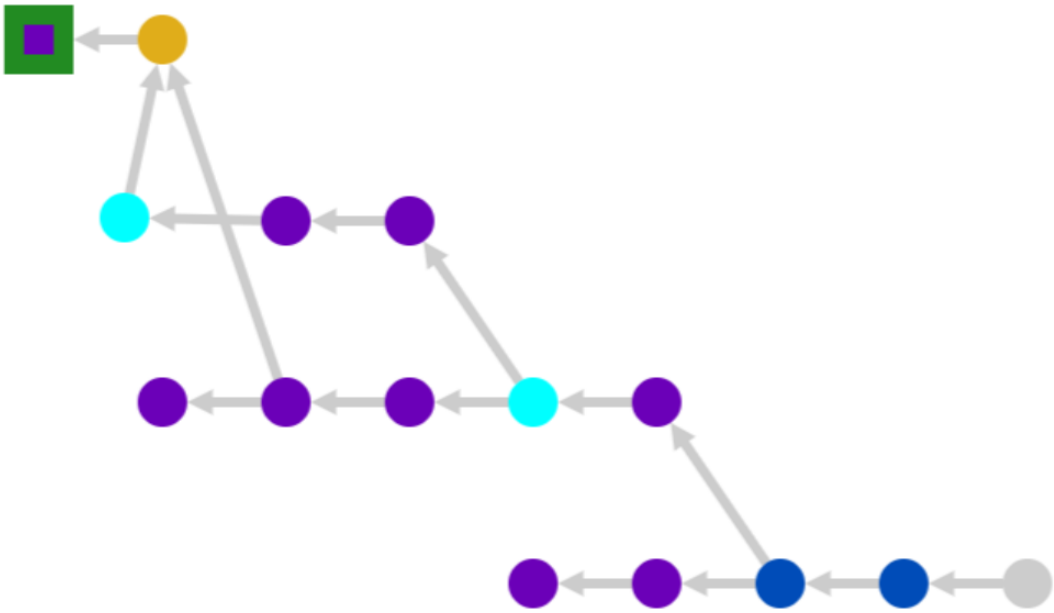


Figure 2.2: A Directed Acyclic Graph (DAG) of a VCS commit history. Although time is conventionally represented as moving from left to right in VCS DAGs, the direction has been reversed here to more closely resemble the structure of Duchamp's *Réseaux des Stoppages*

ongoing divergence, allowing branching processes to unfold across time and collaboration as part of the creative workflow itself.

Some of the projects explored in the following chapter embed VCS within the artwork itself. In these cases, VCS becomes part of the aesthetic and conceptual structure: the artwork tracks its own development and invites transformation.

Extending Gell's insights, Born (2005) frames the oeuvre through a social and institutional lens. She emphasizes that artistic works are both spatiotemporally distributed and socially embedded, shaped by relations between people and things. For Born, creative works are assemblages: their form and meaning evolve through interaction, mediation, and institutional entanglement.

As Gell notes, an artwork "changes not only via its changing interpretation in performance or reception, but it can change even in its very physical form" (Born 2005, p. 16). VCS supports precisely this: changes not just in interpretation, but in material structure—offering the means to revise, reconfigure, and reinterpret a work as it travels through time and hands.

Haworth (2015) extends the oeuvre further by analyzing how techniques, software environments, and compositional strategies propagate through creative communities—particularly in electroacoustic music. His work highlights how the reuse of synthesis techniques create a lineage of creative decisions that connects otherwise separate works and composers. Even in the absence of direct documentation, Haworth demonstrates that artistic genealogy can be inferred through shared technical structures.

This has important implications for preservation and authorship. As works become increasingly tied to custom tools, code dependencies, and performance environments, the creative process itself—including drafts, failures, annotations, and forks—must be preserved alongside the final output (Boutard and Guastavino 2012; McPherson and Kim 2012).

Taken together, these perspectives offer a basis for understanding VCS as cultural infrastructures for the shaping of creative time. They help us to see VCS as structuring the evolving careers of works (Born), the genealogies of technique (Haworth), and the latent potential of suspended creative states (Gell). VCS preserve what might otherwise disappear: the momentary decisions, speculative branches, and relational contexts that make up the distributed, recursive life of creative artifacts.

This dissertation extends these discussions by proposing the repository of a VCS as a kind of protentional-retentional schema—not at the scale of internal time-consciousness, nor the long arc of an artist's career, but at the level of an individual project. Heidegger's concept of positionality (Gestell) refers to the way modern technology discloses the world as a system of resources—framing things, people, and processes in terms of their availability, utility, and readiness for use. It is a mode of revealing that positions everything as standing reserve (*Bestand*) (Heidegger and Mitchell 2012, p. 61). A software repository can be understood in these terms as a structured reserve of possibilities: a system that stores, organizes, and renders past states accessible for future activation or reinterpretation. Just as a carpenter's table "concernfully approaches" through the act of its making (Heidegger and Mitchell 2012, p. 25), each commit in a version history is both a trace of prior use and a latent material for further action.

Kery, Horvath, and B. Myers (2017) investigated how data scientists manage code during exploratory programming, a process marked by non-linear iteration and uncertain goals. Through semi-structured interviews and surveys, they documented a heavy reliance on informal versioning practices such as commenting out code, duplicating files or snippets, and retaining unused code for potential reuse. These ad-hoc strategies supported rapid branching and comparison at arbitrary levels of granularity, such as lines, snippets, or

functions—without leaving the code editor. However, they also imposed significant cognitive burdens, as users had to maintain strong mental models of their evolving code, leading to errors, inconsistencies, and difficulties in revisiting earlier work. Even among participants with formal computer science training, conventional systems like Git were rarely used in exploratory contexts, due to their perceived overhead and poor fit with fragment-level experimentation. In response, the authors developed Variolite, an editor extension that supports lightweight, in-place versioning through "variant boxes," enabling users to create, switch, and nest multiple alternatives of any code segment. The design rationale behind Variolite highlights the value of embedding branching and history management directly within a creative workspace: arbitrary-granularity versioning, immediate comparability of alternatives, and lightweight interaction.

Sterman, Nicholas, and Paulos (2022) examine how creative practitioners across fields such as software engineering, creative coding, weaving, violin making, performance, and industrial design engage with version histories, adapting or rejecting paradigms from software version control systems (VCS). Based on 18 semi-structured interviews, they identify four thematic uses of versions: as a palette of materials for recombination, as a source of confidence and freedom to explore by reducing risk, as low-fidelity captures that foster spontaneity, and as resources for long-timescale reuse across projects and years. Situating these practices within literature on history management, VCS, and creativity support tools, they argue for the development of Creative Version Control Systems (CVCS) that foreground creative values. Their design recommendations include enabling parallel access to versions, supporting dynamic fidelity levels, and facilitating reflection across timescales, emphasizing that while software VCS excel in efficiency and precision, creative practices often prioritize flexibility, material engagement, and process sensitivity.

The integration of version control and state management is a critical step for collaborative musical software. Miranda and Rutz (2010) demonstrate VCS tailored to musical composition processes, supporting varied historical trajectories and interactive installations. Dekker (2018) emphasizes conservation methodologies integrating versioning for dynamic net-based artworks. Hallam and Ingold (2007) define creativity as "the production of novelty through the recombination of already extant elements" (Hallam and Ingold 2007, p. 11), a process that resonates strongly with how a merge operation reconciles the divergent histories of two branches, combining their differences into a new, unified state, while simultaneously preserving the paths that led to this divergence.

Olga Goriunova emphasizes the aesthetic and conceptual significance of creative works that remain incomplete or unrealized. These works, she writes, "retain the scintillating unattainability of art in the scale of their audacity in the creation of the novel material versions of the future" (Goriunova 2014, p. 262). Stalled or abandoned trajectories offer unique insight into artistic process and speculative potential, and they embody a mode of creativity that resists closure, pointing instead to a horizon of possibilities that never fully crystallize. In this view, the value of the unfinished lies in what it makes possible for others who encounter it. VCS make this kind of open-endedness practical. Even if a direction wasn't finished or didn't work out, it stays part of the project's history and can be picked up again later. Instead of deleting unfinished work, version control keeps it available for future use.

A persistent challenge in digital and electroacoustic music is the difficulty of transmitting what Haworth (2015) calls the "hidden software ecology" of composition: the undocumented code and tacit knowledge that shape creative practice but resist reuse or interpretation. A good example of the challenges this poses is what McPherson and Kim (2012) describe as "the problem of the second performer," where the reproduction of a software-based work is hindered by the absence of standardized interfaces or documentation, or shared design language.

While reproducibility is a key technical challenge, equally important is access to the creative process itself—in the case of a modular synthesizer patch, how it is constructed and modified through time and interaction. This kind of visibility matters for how creative work is taught, shared, and credited. Pedagogically, drafts and patch notes can offer students and fellow artists crucial insight into how works are made. Relationally, in collaborative music-making, authorship is often difficult to untangle, and without a record of process, contributions can be obscured. This concern connects to broader critiques of how creativity is framed and rewarded. As Battersby (1989) notes, the "modern male genius" myth detaches artistic output from the collaborative, material, and institutional conditions that enable it. Scholars like Ahmed (2012) and Harney and Moten (2013) further show how institutions often reward individualism while erasing the distributed labor that supports it.

At the same time, it is important to not replace one myth with another. While collaborative and process-based approaches can surface previously obscured forms of participation, they should not be romanticized to the point that solitary creation is diminished or dismissed. The desire to make something on one's own carries real cultural and emotional weight, and can represent an important form of agency and expression. In that light, the challenge is not

to reject singular authorship outright, but to design systems that make space for multiple modes of creative work, both collective and individual. In modular synthesis, this means recognizing the value of collective processes while also honoring the focused, private acts of patching that many artists rely on.

Commits introduce a layer of discretion and subjectivity into what might otherwise seem like a purely procedural history. As Burlet and Hindle (2015) observe in their study of version control usage among computer musicians working with Max/MSP, the frequency and granularity of commits vary significantly from standard software engineering practices: these users made fewer commits overall, committed less frequently, and often outside of conventional work hours—yet showed similar patterns in collaboration and bug tracking. Their findings highlight that VCS reflect the often idiosyncratic rhythms of creative labor.

While atomic commits and clear messages are considered best practices, these conventions are often inconsistently applied—subject to fatigue, haste, or personal style as much as to discipline or domain (Singer et al. 2013; Petre and G. Wilson 2014). This human tendency toward infrequent or irregular documentation has clear parallels in modular synthesis, where patch states are typically preserved only at chosen moments—through saved files, screenshots, or hand-drawn diagrams. The decision to record a commit is a discretionary act, often resulting in a fragmented or lossy account of how a patch came to be. This raises the need for systems that can reduce this burden by automating granular documentation—capturing creative decisions as they happen, rather than relying on users to decide when and how to record their work.

Distributed VCS may appear to counter the solitary genius myth by making revision, divergence, and process structurally visible. However, these systems are not ideologically neutral. Platforms like GitHub reinforce metrics of visibility, reputation, and authorship that continue to privilege individual contribution, often obscuring forms of collaborative labor that do not translate into commits or pull requests (Dabbish et al. 2012; Liang, Zimmermann, and Ford 2022). While VCS introduces nonlinearity into the temporal structure of projects, it still encodes assumptions about accountability, ownership, and productivity rooted in software engineering culture (Nafus 2012). In this sense, it both resists and reproduces the ideologies it appears to challenge—offering a critical site for examining how infrastructures shape who gets seen, credited, and remembered.

As previously noted, Gitelman and Drucker draw attention to how documentation systems and interfaces are never neutral, shaping what becomes visible, knowable, or legible as creative work. Building on this, Rodgers (2020) critiques the masculinized discourse

of electronic music technologies, where tools often encode assumptions about authorship, mastery, and legitimacy, and about who gets to claim those privileges. Cowan and Rault (2018) similarly foreground the importance of networked support systems for feminist and queer creative labor—emphasizing practices of co-presence, maintenance, and mutual care that rarely surface in formal accounts of authorship. Together, these authors show that infrastructures actively shape how creative work is recognised, circulated, and remembered. This is especially relevant to VCS, which play a central role in the system proposed in this dissertation. While VCS can support collaborative, distributed, and incremental forms of music-making, they may also reinforce assumptions about authorship, control, and technical expertise—especially when embedded in software interfaces that appear neutral or universal. Local-first systems may mitigate some concerns by decentralizing ownership and allowing users greater control over their data and processes. However, they are not exempt from broader design ideologies and still demand critical scrutiny. Acknowledging these risks is essential in order to foreground the values and affordances that should guide implementations of VCS within participatory and improvisational music systems, like Forking Paths.

Beyond authorship and power, version control tools also shape how creative time is organized and experienced. Olga Goriunova's work on software art and digital aesthetics provides a clear lens for examining the temporal dynamics of VCS in creative practice. She identifies how digital signal processing and software art offer new modes of temporal experience, actively modulating its perception and use. These systems, she argues, enable artists to work within layered and anticipatory temporal structures, where the present moment is entangled with unrealized futures and reconfigurable pasts (Goriunova 2014, p. 254).

This dissertation begins from the proposition that such temporal structures can be embodied through the integration of VCS into software-based modular synthesizers. In Forking Paths, this principle is carried forward through the design of tools that treat musical time not as a branching system of states, gestures, and transformations, and drawing with Goriunova's account of the avant-garde as a field of possible futures configured in the present.

Annet Dekker turns our attention to the archival value of VCS , where her work on the conservation of net art considers versioning as intrinsic to the ontology of digital and process-based works. She argues that traditional conservation practices, which focus on fixing artworks in time and preserving singular, canonical forms, are ill-suited to the fluidity and mutability of software-based art. Instead, Dekker calls for "a new understanding

of conservation theory that embraces versioning and process as inherent qualities of the artform" (Dekker 2018, p. 13). This insight positions VCS as essential infrastructure for maintaining and meaningfully preserving digital works that captures the unfolding identity of an artwork over time. Dekker argues that documentation should be treated as a central practice of artistic and historical continuity, that capturing the creative and contextual scaffolding of a work becomes as important as maintaining its media artifacts.

As the next section will explore, software and hardware modular synthesis makes these dynamics especially vivid. Patchable systems foreground process over outcome, improvisation over fixity, and structure over surface. By viewing these systems as infrastructures that evolve over time, we can better understand how instrument design becomes a form of distributed authorship, and how versioning might support the speculative, processual nature of experimental sound practices.

2.4 Modular Synthesis

The preceding discussion has shown how version-controlled environments and software infrastructures condition the possibilities of authorship, collaboration, and preservation. These dynamics are also central to modular synthesis, both as a musical practice and as a sociotechnical assemblage.

Modular synthesizers, whether hardware-based or virtual, offer a revealing case study for exploring how modular infrastructures shape the creative process itself. A modular synthesizer is composed of discrete signal processing units called modules, with each designed to perform one or more specific functions. These modules are patched together manually, exposing parts of their internal circuitry and enabling performers to create bespoke instrument configurations. As Palumbo and Wakefield (2024) note, this modular structure affords a high degree of plasticity, where the instrument's topology itself becomes an expressive parameter. In hardware systems, replicating exact module arrangements or knob settings is often impractical, and is a constraint that some performers embrace as a source of creative freedom, while others lament its limitations. The authors discuss how this impermanence becomes part of the compositional method. By contrast, software-based modular environments allow patches and module arrangements to be saved and reloaded, enabling forms of composition and preservation that are otherwise unavailable—but this comes at a sacrifice of tactility and embodiment.

This logic extends across both hardware and software implementations. VCV Rack, for instance, offers a highly extensible, open-source software simulation of Eurorack hardware, allowing players to experiment with modular systems without physical components (Belt 2017). Similarly, *BEAP* (Davidson 2013), which is an educational modular synthesis library built within Max/MSP (Cycling '74 2024), has been designed to introduce students to modular thinking through an intuitive patching interface (Mazzola et al. 2018)(Mazzola 2018). These tools reflect an increasing interest in and accessibility of modular synthesis, while also raising questions about how interface and visual metaphor shape learning and interaction.

Lindh (2018) examines this dynamic through a study of subtractive synthesizer interface design, contrasting skeuomorphic and flat UI paradigms. Lindh argues that the design of software synthesis interfaces directly influences intuitive interaction, pedagogy, and aesthetic perception. These decisions are not merely stylistic, as they shape how synthesis is learned, used, and imagined.

The pedagogical and collaborative dimensions of modular synthesis are explored by Kevin Austin and Ryan Gaston. Austin (2016) offers foundational definitions of modular synthesis, tracing its roots in voltage control and outlining core concepts such as signal routing, module classification, and patch behavior. His work provides a bridge between the technical and artistic dimensions of hardware-based synthesis. Meanwhile, Gaston (2016b) focuses on the integration of modular synthesizers into ensemble performance contexts, identifying several barriers to collaborative engagement. He highlights the lack of standard pedagogical frameworks and notational systems, as well as features of modular synths—such as their default-on state, generative autonomy, and spatial detachment—that can challenge shared musical intuition and real-time coordination (Gaston 2016a, p. 4).

These collaborative frictions underscore the social and infrastructural complexity of modular synthesis. As Randell and Rietveld (Randell and Rietveld 2024) argue, modular performance often involves systems of interdependence and serendipity, where meaning emerges in relation to dynamic networks of signal, interface, and gesture.

2.5 Visual Programming Languages

Visual programming languages (VPLs) share important structural and conceptual similarities with modular synthesis environments, particularly in how they visualize signal flow and modular relationships. Though VPLs emerged from a broader history of graphical sys-

tem design, such as with *LabVIEW* (Kodosky 2020)), tools like Max/MSP and Pure Data (Puckette et al. 1996) have been especially influential in musical contexts, enabling players to construct real-time audio processes by visually linking functional modules of code. These tools blur the boundaries between composition, performance, and system building, offering an interface model where musical thinking is structured through spatial metaphors of patching, flow, and transformation. This logic is similar to modular synthesis, but VPLs differ in key ways: they typically prioritize programmatic flexibility and extensibility over performative immediacy, and often integrate data processing or control logic not native to most modular synth interfaces. For this reason, they raise unique challenges when integrating with VCS, particularly due to the graphical nature of patches, which complicates diffing, merging, and tracking fine-grained changes when compared to text-based code.

A recent project that vividly illustrates this overlap is *Mischmasch*. In their study, Wakefield, Palumbo, and Zonta (2020) investigated how the physicality, modularity, and improvisational flexibility of hardware modular synthesis (MS) could be meaningfully translated into room-scale, multi-user virtual reality (VR). Rather than attempting a literal simulation of MS in VR, they posed a more generative question: *What if modular synthesizers had been invented after VR?* This thought experiment guided the design of *Mischmasch*, a collaborative VR environment that combined the embodied cognition and spatial memory central to hardware MS while adding the flexibility and dynamic-editing capabilities of software-based systems and visual programming languages. Notably, the system employed a graph-based Operational Transform protocol to coordinate concurrent edits and dynamically recompiled *gen~* patchers in real time.

Fasciani and Rahman (2018) propose tangible patch cords that reintroduce physicality into virtual environments, while Paris (2018) demonstrates collaborative remote patching in *Kiwi*, a graphical programming environment for sound and music that extends the patching paradigm of environments like Max/MSP and Pure Data into a real-time collaborative context. Developed as part of the *MUSICOLL* project (Sèdes et al. 2017), Kiwi enables multiple users to simultaneously edit the same audio patch hosted online, and introduces important architectural considerations for shared modular environments, including modes for editing and performance, visual patch inspection, and support for multiple simultaneous views. These projects emphasize the evolving sociotechnical role of patching as a shared and networked form of interaction.

VPLs often differ from hardware modular systems in significant ways. For instance, most graphical environments disallow suspended cables, that is, connections made to only

one module. In hardware synthesis, dangling cables can indicate abandoned directions, unresolved intentions, or be held in reserve for later use.

Phenotropic programming proposes an alternative model of computation, in which modules communicate through perceptual pattern recognition rather than rigid, predefined interfaces (Lanier 2003). Inspired by biological systems and human cognition, phenotropic systems privilege adaptability, ambiguity, and sensory responsiveness over strict formalism. This paradigm offers a provocative contrast to the symbolic logic that governs most programming environments, including many visual programming languages. In the context of digital musical instruments and modular synthesis, Lanier's concept encourages us to imagine tools that engage performers not through strict sequencing or discrete states, but through evolving, perceptual dialogue. It aligns with a broader interest in interface design that supports emergence, reflexivity, and co-agency—values increasingly foregrounded in both experimental instrument design and speculative computational art. Where traditional VPLs tend to prioritize clarity, hierarchy, and operational closure, phenotropic approaches suggest an aesthetics of open-endedness, allowing systems to act less like instruction sets and more like responsive creative partners. For modular synthesis and VPLs, this ethos suggests patches, cables, and interfaces could act as dynamic, evolving ecologies where provisional connections, suspended ideas, and half-finished gestures remain accessible for future transformation.

One challenge in visual programming environments like Max/MSP, as Wakefield noted to me, is the lack of a built-in mechanism for "commenting out" code blocks (Wakefield and Palumbo 2019). In text-based languages, commenting is a convention that removes a line from execution while preserving it for human reference—a practice that favors iterative development by allowing old or speculative ideas to coexist alongside active code. In Max/MSP, by contrast, disconnected objects remain active within the compiled patch unless explicitly encapsulated or removed. As a workaround, some users—including the author—group inactive sections inside sub-patchers with notes describing their function or intent. These informal techniques allow for reflection and experimentation, but they also highlight how the conventions of visual programming shape the kinds of developmental traces that persist. While VPLs offer strengths in spatial reasoning, layout, and visual structure, they may require more intentional design to support practices of revision, speculation, and documentation that emerge more naturally in text-based environments.

At the time of writing, VPLs—such as Max/MSP and Pure Data—do not offer integrated version control. While users may manually save successive versions (often with filenames

like *patchv2.maxpat* or *final-final-edit.maxpat*) or rely on external tools like Git, these practices remain informal and decoupled from the design of the interface itself. This is not unique to VPLs, as most text-based programming environments also treat version control as an external layer, often requiring manual configuration and some degree of technical fluency. However, VPLs have been slower to adopt these tools, likely due in part to a deeper mismatch between their data structures and the assumptions of traditional VCS.

The diffing tools that underlie most VCS have evolved with a textual bias, designed to operate on streams of characters rather than structured data. As a result, changes to graphical patches saved in structured formats (like Max/MSP's JSON-like `.maxpat` files) can generate large, unreadable diffs, even when the actual change is semantically trivial. This problem is not unique to VPLs—similar issues arise when diffing JSON in many modern software projects—but it becomes especially visible in patch-based environments where graphical layout and object ordering in the `.maxpat` are often incidental to function. Tools like Ableton's *maxdiff* (Ableton 2024) and Diemo Schwarz's *diff-for-max* (Schwarz 2025) attempt to address this by offering more structured patch comparisons, but they remain external workarounds rather than integrated solutions.

2.6 Live-Coding and Improvisation Practices

Live coding represents a distinct lineage of real-time musical system design, one in which code becomes both instrument and performance. This section considers how live coding foregrounds temporal engagement and iterative transformation, complicating the relationship between compositional planning and performative spontaneity.

Magnusson (2014) positions algorithmic modification during performance as central to authentic live-coding practices. Nilson (2007) highlights the cognitive challenges inherent in live-coding, balancing real-time code manipulation with musical outcomes. McLean and G. Wiggins (2010) describe live coding languages like TidalCycles as temporal pattern systems that foreground the sequencing and transformation of musical structure in time. A. R. Brown and Sorensen (2007) describe live coding environments as "just-in-time" compositional systems, where code is both the medium and the method of improvisation.

This emphasis on immediacy and structure has led live coders to reimagine core concepts of authorship, iteration, and infrastructure. The TOPLAP manifesto calls for visible, modifiable, and shared code, framing live coding as a practice of radical openness and continuous transformation (TOPLAP 2004). Rohrhuber and De Campo (2009) argue that live

coding distributes authorship across time and across collaborators, producing systems that are both performative and reflexive. Browser-based live coding environments such as Tidal Cycles (McLean 2014) and Estuary (Ogborn and Beverley 2017) support collaborative or networked improvisation. Scholars such as Armitage and Thornham (2021) emphasize the politics of representation and inclusivity within live coding communities, advocating for experimental systems that are as socially and pedagogically open as they are technically responsive.

Wakefield, Charlie Roberts, et al. (2014) presented Alive, a browser-based live coding environment designed for real-time, collaborative improvisation in immersive settings. The system used Share.js[3] to manage simultaneous edits, employing operational transforms (OTs) to resolve conflicts across multiple performers. Alive demonstrated the potential of decentralized performance platforms that combine the flexibility of live coding with the collaborative affordances of the web.

The SHARP system (Manesh, Bowman Jr, and Lee 2024) represents a significant advancement in embedding real-time version control into live coding environments. Designed as a plugin for Tidal Cycles, SHARP creates interactive version trees for individual patterns, allowing performers to navigate, retrieve, and recombine code variants on the fly. Through a user study, the authors demonstrate how this lightweight VCS supports sensemaking, musical form, and high-level improvisation, positioning version history as a core musical resource rather than only a development aid. SHARP offers valuable insights for broader inquiries into versioning, temporality, and creative reuse in computational music tools. Toward the end of this dissertation's research, the closely related SHARP project was published. While both systems explore version control as a creative resource in live performance, Forking Paths differs in that its versioning infrastructure is extended beyond conventional VCS functions to support exploratory tools like *patch history* sequencing, gesture reuse, and branching as a compositional structure. These distinctions will be discussed in more detail in the chapters that follow.

Together, these accounts position live coding as a domain in which musical improvisation and software infrastructure are deeply entwined. The development of systems like SHARP suggests environments where version control invites performers to treat revision, divergence, and return as live materials. This combination of algorithmic improvisation with temporal

[3] At the time, Alive used Share.js to manage simultaneous edits; the project has since been renamed and is now maintained as ShareDB — https://github.com/share/sharedb

tracing leads to the question: how can instruments scaffold reflective, co-authored, and temporally dynamic musical practice?

Live coding's relationship with modular synthesis has been explored by C. C. Hutchins (2015), who compares the embodied, tactile interaction of hardware synthesizers to the abstract nature of text-based live coding. While their interfaces differ—one prioritizing tactile control, the other abstract code—both practices involve live reconfiguration of the instrument, foregrounding the act of shaping musical structure in real time. Hutchins suggests that this interplay between physical manipulation and abstract decision-making offers a nuanced model of musical agency, one that foregrounds both immediacy and structural change.

2.7 Networked Musical Creativity

Literature on telematic and networked performance explores how digital connectivity reshapes authorship, liveness, and embodiment in collaborative contexts. From Pauline Oliveros's Deep Listening (Oliveros 2005) and The Telematic Circle (Oliveros 2009) to Doug Van Nort's work on distributed improvisation and responsive systems (Van Nort 2023), networked performance has evolved into a site of relational listening and co-authored musical agency. The network—once seen mainly as a logistical hurdle due to latency and synchronisation issues—is now embraced as an expressive medium for musical interaction (Tanaka 2000; Chafe, S. Wilson, and Walling 2002). Rather than compensating for distance, telepresence becomes an aesthetic and ethical condition—enabling new models of shared intention, attentional attunement, and temporally distributed creativity.

Networked musical performance literature spans decades of inquiry into how digital connectivity reshapes collaborative music making. Foundational perspectives by Gresham-Lancaster (1998) and Tanaka (2006) trace the evolution of interactive technologies in communal performance settings, highlighting their impact on agency, liveness, and the distribution of authorship. Their work situates the network as a performative space where musicians negotiate control, autonomy, and co-presence.

Barbosa (2003) surveys the evolution of networked music systems, categorizing them into four overlapping types: local interconnected networks for co-located performance, asynchronous composition support systems, remote performance systems constrained by latency, and shared sonic environments for improvisatory participation. The review situates these practices within acoustic communication theory (Truax 1996) and sonic art tradi-

tions, emphasizing how factors such as delay, topology, and participant expertise shape design. Hybrid models, such as FMOL (Jordà 2002), blur distinctions between synchronous and asynchronous collaboration. By documenting architectures, interaction paradigms, and participation scales, Barbosa provides a foundational framework for understanding how network technologies reconfigure collaboration and acoustic community in digital contexts.

Weinberg (2005) develops a historical and theoretical framework for interconnected musical networks, tracing their evolution from acoustic interdependence in traditional ensembles to electronically mediated systems where performers influence each other's output in real time. He classifies such systems into approaches—the Server, Bridge, Shaper, and Construction Kit—and analyzes them through social organization models (centralized, decentralized, democratic, monarchic) and network architectures (e.g., synchronous "flowers," sequential "wheelbarrows," and hybrids). By distinguishing process-centered networks, which foreground participants' creative and social experiences, from structure-centered networks, which prioritize coherent musical products, Weinberg links system design choices—including topology, control mapping, and use of technologies such as personal computing, the Internet, and alternative controllers—to the balance between individual autonomy and collective coherence. His framework positions musical networks as socio-technical ecosystems where performer interaction, technological affordances, and aesthetic priorities co-produce emergent musical forms.

Building on this lineage, Mayton et al. (2012) explore how remote performers can manipulate hardware modular synthesizers over the web, underscoring new paradigms of virtual agency and control across physical distances.

Çakmak, Çamci, and Forbes (2016) introduced Monad, a multimedia instrument that foregrounds shared virtual objects as sites of interaction between performers. Their study emphasizes the importance of embodied interaction in networked music systems, especially when traditional gestural or visual cues are diminished. To navigate network latency and preserve expressivity, the authors suggest prioritizing sonic parameters like timbre, spectrum, and dynamics over precise temporal coordination—a tactic that echoes broader strategies in telematic improvisation. They also note the performers' desire for legible self-representation in virtual space, an ongoing challenge for distributed interfaces.

Lee and Essl (2014) review existing practices in networked live coding, situating them within established taxonomies of network music (Barbosa 2003; Weinberg 2005) and identifying opportunities for future development. They classify collaborative systems by the types of data shared, highlighting trade-offs between centralized and decentralized models.

Centralized architectures simplify synchronization but risk conflicts in a single shared state, while decentralized models allow multiple independent states but require explicit strategies for code, state, and timing consistency. The authors also explore "disparate collaboration," where live coding outputs are sequentially or hierarchically mediated by other performers or systems, and "remote collaboration," which, while less common, raises unique challenges for latency management and symbolic-versus-audio data transmission. They identify underexplored areas such as asynchronous collaboration supported by notation systems, scalability to large or crowd-based ensembles, and diversified performance topologies that combine multiple modes of code distribution and sound generation, arguing for designs that extend beyond traditional centralized/decentralized binaries to embrace hybrid, distributed, and audience-participatory models.

2.8 Web-Based Audio and Visual Tools

With native APIs for audio, real-time graphics, and network communication, browser environments have become powerful platforms for musical and visual experimentation, performance and collaboration. Researchers and artists have leveraged these capabilities to build systems that emphasize immediacy, distribution, and accessibility—redefining what it means to synthesize, patch, and perform within the constraints and affordances of the web. In contrast to desktop-based systems that require installation, configuration, and specific hardware access, browser-based environments offer a frictionless entry point for experimentation and dissemination, aligning with broader values of openness and accessibility in creative coding.

Complementary research by McKinney (2014) presents Lich.js, a browser-based environment for collaborative audio-visual live coding. McKinney highlights the practical benefits of browser-based tools: widespread compatibility, low-latency text transmission, and resilience to jitter make code-sharing especially well-suited to musical collaboration online. His subsequent work situates these developments within a broader history of networked performance, tracing a lineage from John Cage's *Imaginary Landscape No. 4* (Cage 1951) to early internet-based concerts. (McKinney 2016) He emphasizes the importance of bespoke interfaces that mediate interaction not only between musicians but also between performers and audiences—interfaces that reflect the values of experimentalism, adaptability, and communal authorship.

Other systems such as Estuary (Ogborn and Beverley 2017) further extend this paradigm by enabling multiplayer, browser-based live coding with real-time visual feedback and compositional control, creating a model for co-located and remote musical interaction.

Charles Roberts, Wakefield, and Wright (2013) present browser-based digital signal processing libraries, notably exploring the constraints and potentials of the Web Audio API. In exploring the web browser as an audio-visual platform, they introduce *Gibberish.js*, a JavaScript library for efficient DSP synthesis in the browser. Their work addresses limitations of the Web Audio API, such as the lack of sample-accurate timing, difficulties constructing feedback networks, and challenges in supporting self-patching and dynamic routing, all especially relevant for modular or generative synthesis. As a solution, they describe a system of code-generation that pre-compiles synthesis graphs into audio callbacks, enabling optimization of both structure and runtime performance.

Recent developments, including the introduction of AudioWorklets (Google 2024) and WebAssembly (W3C WebAssembly Working Group 2022), have begun to address limitations of the Web Audio API, opening possibilities for faster DSP routines, fine-grained control, and integration with shared memory architectures. These advances position the browser an increasingly sophisticated environment for modular instrument design and collaborative experimentation.

Taken together, these projects reflect an evolving ecology of browser-based music systems that foreground accessibility, collaboration, and experimentation. As the technical capabilities of web audio continue to mature, they not only support increasingly complex synthesis and interaction models but also form the infrastructural backbone for new modes of networked musical collaboration.

Chapter 3

Practice-Based Experiments

Over the course of several years, I developed a series of research-creation projects that explore how version control systems (VCS) might be adapted to support distributed creativity and experimental practices in electronic music. These projects, spanning several years, include *Using Git to Write a Paper About Using Git* (Palumbo 2017b), *Git Show* (Palumbo and Van Nort 2020), *Alicenode Editor* (Palumbo and Wakefield 2018a), *Cards* (Palumbo and Wakefield 2018b), *Graph Operational Transform* (Wakefield and Palumbo 2020), *Modular Synthesis in VR (MSVR)* (Palumbo, Wakefield, and Zonta 2022), and *Mischmasch* (Palumbo, Wakefield, and Zonta 2020; Palumbo and Wakefield 2024). Each one approached a different facet of this inquiry into distributed creativity in musical software environments: from collaborative authorship and projectional editing to real-time patching in immersive environments. Throughout this dissertation I use the term experiment to refer to practice-based explorations and prototypes.

Together, these experiments—and ultimately, the Forking Paths system—form the practice-based backbone of this dissertation's research. This chapter outlines the working process behind each project, including the design decisions, technical constraints, and conceptual insights that shaped them. While these works differ in scope and interface, they collectively reflect an evolving interest in treating software versioning not merely as a tool for ensuring software stability or managing multi-author contributions—as it is often deployed in software engineering and large-scale technical documentation—but as a creative medium for improvisation, musical structure, and ensemble co-creation. The final system documented in this dissertation, Forking Paths, emerged directly from this lineage. Its design and implementation are discussed in full in the next chapter.

3.1 Recursive Writing

Prior to starting the PhD, my research-creation practice was already grounded in bespoke software development and interdisciplinary performance, exploring collective creativity, redistribution of control, and temporal layering. I entered the PhD as a musician, artist, and creative coder for whom inquiry was inseparable from building custom software systems.

A project that predates this dissertation is *Recursive Writing* (Palumbo 2015), which explored similar concerns through machinima, retro game emulation, and live performance. Audience members were invited to sit with a Super Nintendo controller and a microphone; they first watched a machinima sequence derived from The Legend of Zelda, in which a previous participant's recorded gameplay and narration gave the characters new plotlines and motivations (for example, recasting the hero as a gardener tending vegetables). At the end of the recorded clip, the game transitioned seamlessly into live play, allowing the new participant to continue the story from the same game state where the machinima had ended. They were then asked to add their own spoken narration, captured through the microphone alongside a recording of their gameplay. Each time a new participant sat down, a machinima video was randomly selected from the database. This meant that subsequent participants encountered different fragments, each shaped by earlier contributions, so that the piece accumulated into a collective, non-linear narrative across successive turns.

Recursive Writing foreshadowed concerns that became an important focus of this dissertation: collective authorship and the interplay of past, present, and future contributions. This continuity grounds the dissertation's methodological orientation and situates Forking Paths as the most recent articulation of a research-creation practice where building custom software is inseparable from making and thinking through art.

3.2 Using Git to Write a Paper About Using Git

My experimentation with software version control began with academic writing. In 2017, during graduate coursework, I adopted a workflow that involved using the Unix-based text editor Nano (Allegretta and Schulenberg 1999) alongside the VCS Git to write and revise academic papers. All of my course notes, drafts, and edits were written entirely in Nano and committed regularly to a Git repository. What began as a pragmatic experiment quickly became a methodological provocation that led to several experimental projects.

The first of these projects, an article titled *Software Version Control Systems as Tools for Research-Creation in the Fine Arts (or: Using Git to Write a Paper About Using Git)*

(Palumbo 2017b), explored this question explicitly. I used Git to manage versions of the text[1], and to analyze its development by mining the repository as a dataset and reflect on the structure of my own revisions. The project was fully self-referential: the paper itself was about version control, written using version control, and analyzed with the tools of version control.

I continued this line of inquiry with a paper titled *A Technicity of Distributed Version Control Through Accelerationism* (Palumbo 2017a), also written entirely within a Git repository[2]. This project was submitted for a seminar on accelerationism and technological mediation. While the abstract framed the work in relation to productivity platforms and distributed collaboration, it didn't fully express the conceptual leap I was attempting to make: to think of the Git commit through Heidegger's concept of Gestell, or enframing (Heidegger and Mitchell 2012). I was drawn in particular to the notion of the standing reserve—the idea that stored elements, once brought into a system, are held in readiness for future use. Commits, I thought, became a reserve of ideas but suspended in a structure where they could be reinterpreted or activated anew.

Over time, my commit messages grew longer than some of the edits they described. After drafting in Nano, I would use the commit message to reflect, extend, and sometimes redirect the idea, turning each log entry into a running commentary that was still anchored to the changes in the text. Gradually, the VCS began to feel like a prosthetic layer for memory, revision, and sense-making.

The paper situated Git within the broader philosophical context of a graduate seminar I took titled *Accelerating Technicity*, drawing on Heidegger, Simondon, and Marcuse to reflect on how VCS do more than coordinate work—they also structure cognition, mediation, and creative agency. I explored how commits, branches, and diffs enact a logic of deferred potential by drawing from Simondon's concept of technical individuation (Simondon, Malaspina, and Rogove 2017, p. 11), and how the user's relationship to these systems echoed Marcuse's notion of self-administration within technological infrastructures (Marcuse 1964). While the paper was still in an exploratory phase, it helped clarify a line of inquiry that would continue in my later projects: version control not just as a technical layer, but as a compositional and epistemological one.

[1]The repository is publicly available at https://github.com/michaelpalumbo/self-referent

[2]https://github.com/michaelpalumbo/spth6155

Although these projects were textual, they anticipated many of the questions that would later guide my work on musical systems, and how VCS could be co-opted for research-creation. In my experiments, by using version control to track and reflect on the writing process about version control, I began to leverage the technical functions of commits and diffs, as recursive compositional forms in their own right.

3.3 Git Show

Git Show (Palumbo and Van Nort 2020) was an experimental project which leveraged version control to explore how distributed creativity could be supported in collaborative musical composition and instrument design. By emphasizing the retention of creative histories and the protention of future possibilities, (Husserl et al. 1964; Gell 2015; Haworth 2015) *Git Show* served as a formative experiment towards later developments of real-time version control such as with *Forking Paths* and *Mischmasch*.

I served as the principal investigator on the project, which I designed and led as part of a research assistantship under the supervision of Dr. Doug Van Nort. The work was developed within the Distributed Performance and Sensorial Immersion Lab[3], directed by Van Nort, who also served as faculty advisor for my research, and it was implemented within Van Nort's graduate-level course Vertical Studio Lab II at York University. *Git Show* became a central activity within the class, with weekly sessions in which students acted as research participants, engaging with the evolving instrument, score, and recordings through a series of structured contributions.

Initially each of the 13 participants was assigned their own GitHub repository for their creative development. Each week, a script was used to reassign repository ownership among the participants, ensuring that each participant inherited a different one from the previous cycle. This redistributed authorship of each repository's contents—including the instrument, score, and version history—on a weekly basis. Each cycle involved iterative development of three artifacts:

- A digital instrument written in Max/MSP. Participants had freedom to modify the instrument and were required to commit a functional version by the end of the week. Max/MSP was chosen for its widespread use in electroacoustic music and relatively low barrier to entry.

[3]https://dispersionlab.org

- A text-based score composed in *README.md* files using Markdown. This format was chosen as a Git-friendly and easily rendered medium for documenting compositional changes.

- A recording made while performing the score on the digital instrument, with file sizes capped at 100MB to comply with GitHub's storage restrictions.

Each week began with participants listening to the previous owner's recording to foster the relationship with the evolving auditory identity of the instrument. By engaging with prior contributions, participants were encouraged to reinterpret and extend the creative potential of the artifacts.

The authorship reassignment script determined repository transfers using criteria such as the number of Max/MSP objects or the modularity of designs.

At the conclusion of each week, the artifacts were tagged in Git with the cycle number and marked as official releases to distinguish them from drafts. Release tagging encourages participants to commit frequently and leave unrealized ideas accessible for future exploration.

The project also incorporated innovative approaches to analyzing creative changes. Using the UNIX diff algorithm, the similarities and differences between recordings from successive cycles were extracted and converted into sound files. This process turned change data into aural material, offering a new perspective on how artifacts evolved over time.

Promising directions for future research include leveraging techniques of mining software repositories (MSR)—such as analyzing Max/MSP projects for recurring design patterns—to inform the reassignment of authorship, to deepen the relationship between iterative processes and musicological and computational studies.

Weekly improvisation sessions offered a social context for participants to test their instruments and engage in collective experimentation. These sessions were logged in a shared project wiki,[4] complementing the software's public homepage.[5]

Git Show adhered to open data and open science principles by conducting all activities through public GitHub issues, threads, and wikis. Open data refers to the practice of making data freely available to anyone for use, redistribution, and modification, typically under minimal restrictions, while open science emphasizes the democratization of scientific

[4]https://github.com/dispersionlab/gitshow/wiki

[5]https://dispersionlab.github.io/gitshow

knowledge through transparency, reproducibility, and public engagement in the research process. Together, these principles aim to foster collaboration, accelerate innovation, and break down barriers to knowledge-sharing across disciplines (McKiernan et al. 2016).

Git Show demonstrated how open data and open science principles can be applied to experimental research-creation projects in the arts. This drew on Dekker's Dekker (2018) argument that for net art and software-based works, "versioning and process" are not auxiliary but intrinsic to the work itself.

While *Git Show* exhibited some potential in employing VCS towards supporting collaborative creativity, it also revealed several limitations inherent in using Git for computational music projects. One significant issue was the low resolution and inconsistent frequency of commits. Since participants were often new to version control, many did not commit changes frequently, with some making only a single commit at the end of the week before submitting their work for the next cycle. This meant that intermediate steps and incremental decisions were not recorded.

While Git is technically capable of handling frequent commits—there is no hard limit preventing updates every millisecond—its architecture and intended use patterns make it ill-suited for capturing real-time interactions in collaborative performance contexts. Git assumes a relatively deliberate act of saving state, where changes are staged, committed, and often accompanied by a message. This workflow prioritizes intentional versioning over continuous, automatic state capture. In practice, attempting to use Git to record every subtle gesture or interaction in a group improvisation—such as turning a knob or connecting a patch cable—would require wrapping every interaction in an explicit commit operation, leading to bloated histories, unreadable diffs, and significant performance overhead. More importantly, Git lacks built-in mechanisms for synchronizing concurrent changes across multiple users in real time. It was designed for asynchronous, distributed workflows where users commit changes independently and merge them later. In live collaboration, this can cause race conditions, merge conflicts, and conceptual mismatches between Git's tree-based, file-oriented model and the granular, often ephemeral nature of real-time media interactions. These limitations prompted me to explore alternative approaches to version control that better accommodate the fluidity and simultaneity of live performance, such as conflict-free replicated data types (CRDTs) and real-time state synchronization systems used in later iterations of this project.

3.4 Alicenode Editor

In 2018, I co-developed the Alicenode Editor[6] (Palumbo and Wakefield 2018a) with Dr. Wakefield through a research assistantship in the Alice Lab at York University. The editor was designed to support *Artificial Nature: Insuperposition* (Ji and Wakefield 2018), an artwork by Dr. Haru Ji and Dr. Graham Wakefield, which exhibited at the Daejeon Museum of Art in Korea that same year. A key technical feature of *Insuperposition* was its ability to hotload code—updating the simulation at runtime without needing to restart the system. The Alicenode Editor provided a web-based interface to interact with this functionality, enabling remote updates to the artwork's codebase during installation and exhibition. The software ran on a PC mounted in the gallery ceiling, which was remotely accessed and updated throughout the exhibition period (see Figure 3.1).

The Alicenode Editor presented users with two simultaneous views of a file: one representing the current version in use by the *Insuperposition* simulation, and another drawn from a recalled commit. The text editor interface was built using CodeMirror (Haverbeke 2024a), a browser-based code editor that supports syntax highlighting, inline editing, and plugin extensions. Its dual-pane "merge view" enabled visual diffing between file versions and allowed users to easily copy lines or sections from one version into the other (see Figure 3.2).

This front-end functionality was made possible through Git worktrees[7], which is a lesser-known feature that allows multiple working directories to exist simultaneously within a single repository. Unlike typical Git workflows, where only one repository version may be accessed at a time, worktrees enable separate branches or commits to be checked out in parallel. This made it possible to load two distinct file versions side by side in the editor, each mapped to an independent working directory on the server, while preserving Git's ability to track and update both versions.

Users could then select either the "Version In Use" or a "Recalled Version" and send it to the server, triggering a hotload into the running simulation. If a new version introduced instability, the prior version could be quickly reloaded to restore functionality. By leveraging worktrees, the editor supported concurrent development by multiple collaborators, who could dip into each other's branches without conflicts—a concept inspired by "Worlds:

[6]https://github.com/worldmaking/alicenode/tree/master/client

[7]Documentation on the Git worktree can be found at https://git-scm.com/docs/git-worktree

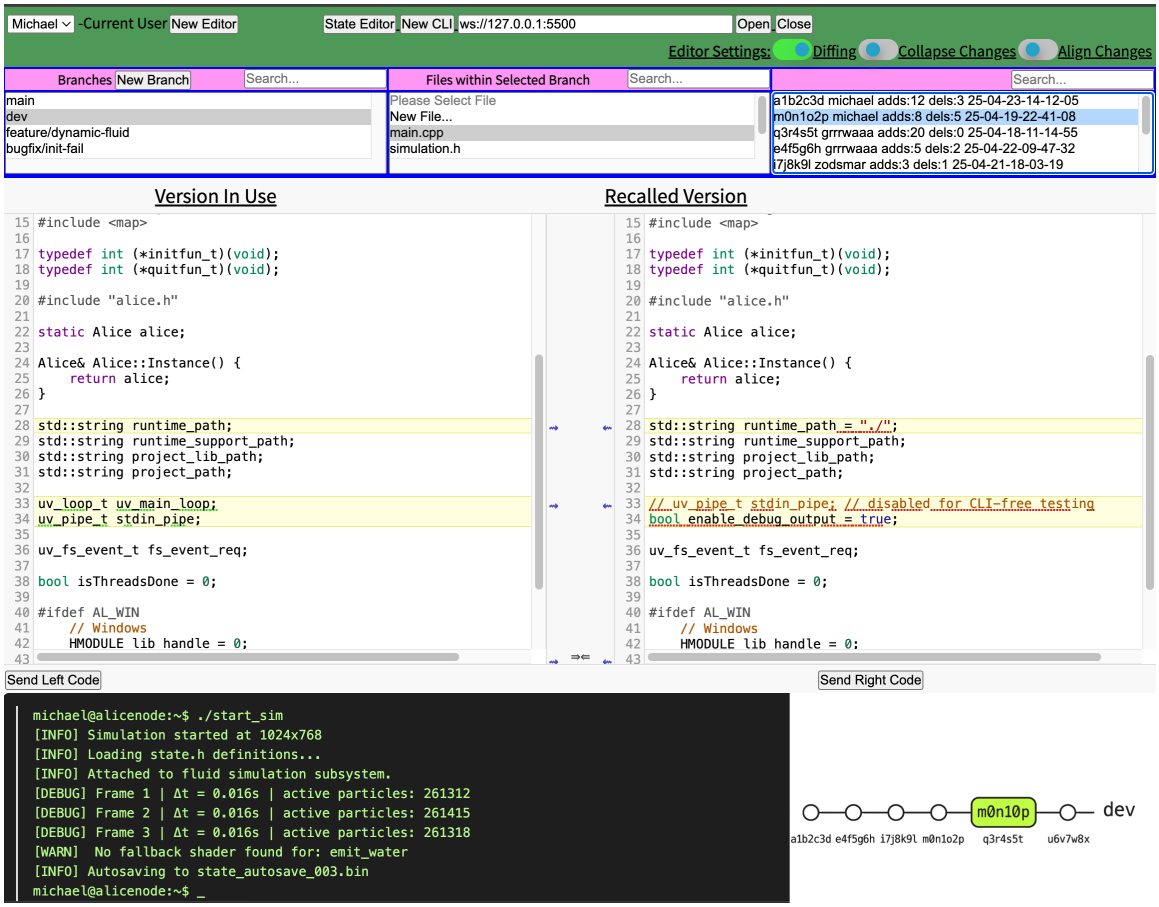


Figure 3.1: The Alicenode Editor interface, which includes searchable branch, file, and commit selectors; two simultaneous and editable code views with diff highlighting and change folding; an integrated terminal connected to the host PC; and an interactive version graph for retrieving or displaying the current branch and commit. The terminal, powered by xterm.js provided flexible access for tasks not yet handled through the graphical interface.

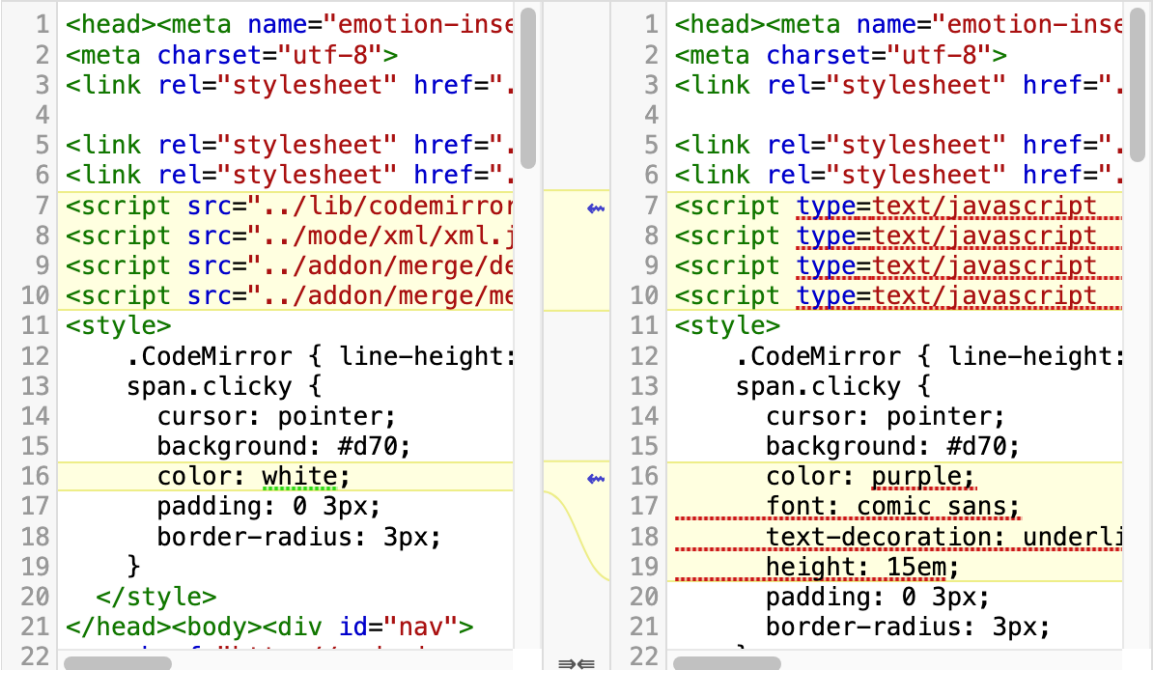


Figure 3.2: Screenshot of the MergeView interface in CodeMirror (Haverbeke 2024b)

Controlling the Scope of Side Effects" (Warth et al. 2011), and by Dr. Wakefield's broader vision of enabling code to be authored within immersive VR environments. A terminal emulator was integrated into the app using xterm.js (Kasidiaris 2013), enabling us to control the PC through the command line in the same interface.

Another notable part of the app was the Alicenode State Editor, an interface for visualizing and modifying the internal variables of the real-time simulation (see Figure 3.3). It exposed the live state struct, offering editable access to simulation parameters such as positions, velocities, colors, and field values. The interface remained synced with the live `state.bin` data structure (a remote memory-mapped file), presenting parameters in a scrollable, categorized table where values could be adjusted in real time using text fields. The UI was generated automatically based on the C++ type definitions in the file `state.h`, streamlining the process by exposing internal simulation parameters.

A key feature of the editor made use of another Codemirror instance which displayed the source code of `state.h` and allowed for direct editing. This enabled us to test changes interactively through the UI and then commit them to state.h, ensuring those edits would persist in future simulation runs. Modified parameters were highlighted in the code, making it easy to trace changes back to their definitions and maintain a clear link between live state and source. Commits were automatically generated with our own GitHub usernames.

Figure 3.3: The Alicenode State Editor interface, showing synchronized views of live simulation parameters and source code. The left panel displays the state.h file using CodeMirror with syntax highlighting and active line highlights for edited constants. The right panels show the corresponding parameter table, where variables such as LAND_DIM and FUNGUS_DIM can be modified in real time. Color highlights visually link edited rows to their source definitions, supporting fast, intuitive parameter tuning within a live simulation environment. Not all parameters are shown in view, in order to prioritize screen legibility.

The development of Alicenode Editor, which enabled remote and real-time code modifications with a particular focus on interacting with version control branches, laid some of the groundwork for Forking Paths. Techniques such as Git worktrees for branch management informed the approach to real-time versioning while having access to multiple branches simultaneously. Once the team returned from Korea, the ideas developed for Alicenode Editor spurred research into projectional editors. The team explored how 3D space could represent multiple running instances of code from different worktrees and how to edit a program's abstract syntax tree (AST) directly. These explorations eventually evolved into a new project called Cards.

3.5 Cards

3.5.1 Background

The Cards project began in 2018 as an attempt to rethink how code is authored, visualized, and transformed, especially in the context of collaborative, runtime, and immersive environments. Initiated within Alice Lab under Dr. Wakefield's supervision, and influenced by ongoing work on simulation-based artworks, Cards was set out with three interrelated goals: to explore version control as a creative medium, to prototype a collaborative code editing interface suitable for VR, and to investigate how Intentional Programming (Simonyi, Christerson, and Clifford 2006) might inform new approaches to authoring and navigating program logic.

While traditional programming tools often enforce a narrow focus on source text and sequential logic, Cards proposed a multimodal, projectional editing environment in which the same code could be viewed and modified through abstract syntax trees (ASTs), semantic graphs (ASGs), and textual representations. These views were designed to be structurally aware, navigable, and, in some cases, editable in real time. The aim was to enable fluid movement across levels of abstraction, from the conceptual to the operational, while supporting collaborative and branch-aware workflows as found in a VCS.

Although the project did not reach full implementation, it produced several functional components and a set of design insights that would go on to influence future work on runtime systems and modular synthesis in immersive environments.

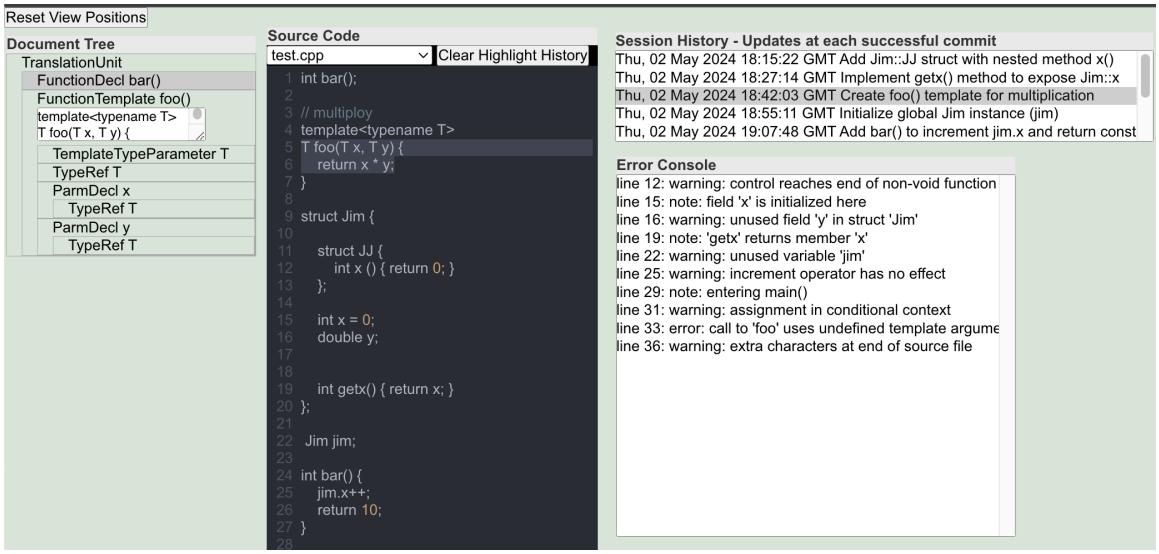


Figure 3.4: This view of the Cards editor visualizes and edits a parsed C++ codebase using a draggable, multi-pane layout. Users can browse the AST in the document tree, view and edit source files with syntax highlighting, track commit history, inspect past versions, and review compile-time errors—all connected to a live backend that responds to updates in real time.

3.5.2 Implemented Features and Technical Foundations

Several components of the Cards system were implemented, forming the basis of a projectional C++ editor designed for collaborative, runtime composition. This included a functioning abstract syntax tree (AST) editor, built using a custom NodeJS script called `cpp2json.js`[8] that parsed C++ source files into structured JSON representations suitable for browser-based display. These ASTs were displayed as hierarchical trees in the browser interface, where each node could be expanded or collapsed, and in many cases edited directly via inline text fields. This enabled structured editing of C++ code while preserving a visual overview of program flow (see Figure 3.4).

The AST view was integrated with a CodeMirror-based text editor, allowing users to highlight the source location of any selected AST node. This dual-view system provided a fluid bridge between structural representations and raw code.

Cards inherited a client-server architecture from the Alicenode Editor, enabling edits made in the browser to be transmitted to a persistent simulation process on the server. Although full runtime patching was not achieved, the infrastructure supported near–just-in-

[8]cpp2json archived at https://github.com/worldmaking/cards/blob/master/cpp2json/cpp2json.cpp

time code updates. Git integration—with per-author commit metadata—provided traceable versioning for every contributor. Worktree support and branch-aware views allowed collaborators to explore different code states in parallel editor windows.

While the system remained in an experimental phase, its client-server model, dual-view interface, and Git-integrated collaboration tools demonstrated the viability of structure-aware editing environments tailored for creative coding workflows.

3.5.3 Conceptual Features and Partially Completed Work

Several speculative features shaped the design of Cards but remained either partially prototyped or conceptual. These ideas are noted here for continuity, as many were developed further in later projects:

- Interactive cards: Inspired by patcher environments like Max/MSP, Cards aimed to introduce draggable, modular interface blocks representing discrete functions. Though partially mocked up, they were not integrated into a full editing loop.
- Reusable function library: A language-agnostic code repository was envisioned to store user-submitted functions, code fragments, and documentation for creative reuse across projects.
- Multi-layered code navigation: Drawing from Intentional Programming, Cards aimed to allow editing across textual, syntactic (AST), and semantic (ASG) layers, supporting interaction at the level of user intent.
- Abstract Semantic Graph (ASG): Proposed as a more flexible alternative to ASTs, the ASG would allow speculative or unconnected code structures to remain spatially represented and contextually navigable.
- Round-trip transformation: Synchronization across these layers (e.g., AST $\leftrightarrow$ ASG $\leftrightarrow$ text) proved complex, leading to a rethinking of the data model to support structural consistency across views.

3.5.4 Graph Operational Transform

Cards was also motivated by earlier experiments like Git Show, which explored how version control might function as a compositional tool. That project highlighted the creative value of making changes legible and replayable as part of the expressive process itself. It also

revealed the limitations of third-generation VCS tools when applied to real-time, collaborative environments. Drawing on these insights, we began developing a fourth-generation VCS for Cards that would support continuous, fine-grained versioning and real-time concurrency—features necessary to reflect the collaborative and processual nature of research-creation.

Operational Transformation (OT) (Sun and Ellis 1998) addresses these limitations effectively by continuously tracking every incremental edit as a delta, thus preserving the exact order and granularity of each change. This approach makes OT particularly suitable for real-time concurrency management. A primary example of OT's use is in online shared document editors such as Google Docs, where multiple users may simultaneously edit the same text. OT ensures all changes are recorded and resolves conflicts automatically through a process known as rebasing. Each user's edits are applied immediately to their local document instance for responsiveness and are subsequently validated against the authoritative shared version on a server. The server then propagates these validated changes to all other users, ensuring seamless synchronization without manual intervention.

Since Cards used a graph as its core data structure, adapting Operational Transform (OT) techniques required rethinking how edits could be represented and merged. Traditional OT systems were developed for text documents, where operations reference two-dimensional positions: line number and character offset. This model assumes a linear sequence of symbols and is well-suited for plain-text editing, but it does not generalize easily to topologies where relationships between elements are nonlinear and multidirectional.

Graph-based systems like visual programming environments or modular synthesizers require a different approach. Patches are composed of interconnected nodes and edges whose structure cannot be described with two-dimensional positional coordinates. The primary innovation of Graph Operational Transform (GOT) was to design an OT around nodes and edges rather than text positions—defining atomic operations such as adding, deleting, or re-routing nodes in a way that could be rebased and merged across concurrent edits. The proposed schema for these operations was documented in the project's wiki[9] and drew from other established javascript-based OT systems like JOT (Tauberer 2025) and quilljs/delta (J. Chen 2014).

The GOT also followed the OT's replicated architecture, in which changes are applied immediately at the local site and then propagated across the system, so that its effects

[9]https://github.com/worldmaking/cards/wiki/List-of-Operational-Transforms

are immediately visible and audible within the virtual environment. This design prioritizes responsiveness and preserves the fluidity of user interaction, which is essential in a real-time audiovisual environment.

Local changes are simultaneously sent to a central server, where they are negotiated against concurrent edits from other users. If necessary, a rebase operation is performed, and the original client receives a corrected set of deltas to ensure global consistency. While the effect of this is similar to the replicated approach in terms of immediacy, it ultimately maintains server authority by ensuring that only validated, rebased updates are accepted into the canonical scene graph. This hybrid model balances the need for local responsiveness with the integrity and consistency provided by centralized coordination. In doing so, GOT addresses the key inconsistency challenges outlined by Sun and Chen (2002), including causality preservation, ensuring robust synchronization in collaborative, real-time editing environments.

The operations in the GOT system can be categorized based on their effects on the graph. Structural operations, such as `newnode`, `delnode`, `repath`, `connect`, and `disconnect`, directly modify the graph's topology by adding or removing nodes and edges or altering their connections. In contrast, property-change operations, such as `propchange`, modify the attributes of nodes or edges without altering the underlying structure of the graph. For instance, a property-change operation might adjust the weight of an edge or update a node's label. These distinctions between structural and non-structural operations are crucial for maintaining consistency and enabling precise control within the system.

Together, these principles, drawn from OT theory and adapted to the unique demands of graph-based editing, formed the foundation of the GOT system. GOT represented a critical conceptual advancement: a model for fine-grained, causally consistent collaboration on nonlinear data structures. GOT laid the groundwork for subsequent projects in this dissertation, tools capable of supporting real-time, multi-user composition across complex representational layers.

3.6 Modular Synthesis in Virtual Reality

Cards was initially explored as a potential framework for immersive code editing in VR, inspired in part by Jaron Lanier's early live-coding experiments in which collaborators could remove, modify, and reinsert code while the system continued running (Lanier 2003). This notion of runtime flexibility and immersive authoring aligned with Cards' broader design

goals, even though the VR component was never realized. Unlike Alicenode, which was created to address a specific installation need, Cards lacked a defined application at the outset. One idea was to use it as an editor for the Rhino SDK, but this was eventually set aside due to its overwhelming scope.

Instead, experiments with modular synthesis and visual programming shifted the focus toward creating a modular synthesizer environment in VR. This direction reflected ongoing interests in collaborative improvisation, branching version histories, and the author's own growing engagement with hardware modular synthesis. The analogy between modular patching and code editing became increasingly central, where components can be freely connected, disconnected, or replaced without disrupting the overall system. We wanted coding to feel more like communal music-making, emphasizing immediacy, improvisation, and flexibility in creative workflows.

In 2019, this trajectory led to the development of Modular Synthesis in Virtual Reality (MSVR) (Palumbo, Zonta, and Wakefield 2020), a collaborative project between the present author, Dr. Graham Wakefield, and undergraduate students Alexander Zonta and Andrew Sidsworth. Initially built entirely in Max/MSP, MSVR prototyped the potential of combining modular synthesis with multiplayer VR environments and raised new questions about how version control might be visualized and navigated in three-dimensional space (see Figure 3.5).

While software-based modular synthesis environments have proliferated on desktop and mobile platforms (Belt 2017; Davidson 2013; Eriksson 2017) and offer conveniences like rapid instantiation, patch recall, and algorithmic extensibility, they often come at the cost of embodied interaction. These systems typically compress musical engagement into flat screens and mouse-based inputs, limiting the spatial and tactile qualities central to analog modular practice (Holden 2015). With Mischmasch, we set out to reintroduce and expand these physical dynamics by leveraging room-scale VR. By drawing on VR's affordances for movement, immersion, and spatial feedback (Andersson, Erkut, and Serafin 2019), the project aimed to merge the open-ended flexibility of digital synthesis with the situated, bodily interaction that defines hardware systems, bridging the gap between virtual plasticity and physical presence.

After early prototyping entirely within Max/MSP, the team settled on a distributed architecture in which VR rendering and input control were handled through a browser-based application built with Three.js (Cabello 2010) and the WebVR API (*WebVR* 2016).

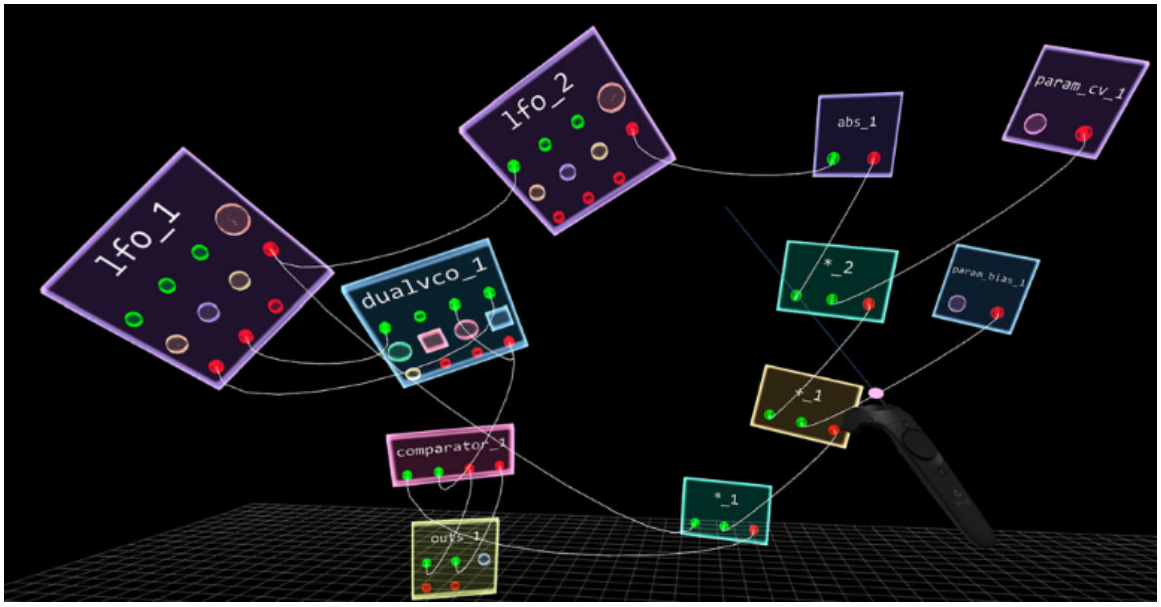


Figure 3.5: A screenshot of a modular synthesis patch created within MSVR

Player interactions within the VR scene were recorded as GOT deltas on a shared graph document, which was continuously updated and fed into the animation loop to reflect changes in real time. These deltas also triggered the generation of corresponding efficient audio synthesis code using *gen~* in Max/MSP through patcher scripting techniques, allowing player interactions to be mapped directly onto sonic changes.

Peer-to-peer connections were brokered using a signaling server called Tea Party (Palumbo 2019), a reference to Alice in Wonderland (Carroll 1865), the namesake of Dr. Wakefield's research lab, the Alice Lab for Computational Worldmaking. Each instance of the application included both a `server.js` and a `client.js` script, authored in Node.js. When a player connected to the Tea Party server, the first participant to enter a room would launch the `server.js` script, assuming the role of session host. All subsequent players launched the `client.js` script and connected to the host for synchronization.

The session host's server supports bidirectional communication and hosts the GOT system to synchronize edits among collaborators. Incoming GOT deltas are applied to the host's authoritative version of the graph and then propagated to all connected clients to be reflected in their local scenes. If a player went offline, their app would start a local server to continue managing the state of their graph.

We integrated GOT in MSVR with several key goals. One primary objective was minimizing latency, which is crucial in motion-tracked, room-scale virtual reality (VR)

environments to maintain immersion. Low latency ensures that interactions feel natural and responsive, which is an essential factor for sustaining player engagement and avoiding disorientation.

Another core goal was to support concurrent editing of the scene graph by multiple players, a challenge explored by Sun and D. Chen (2002) in the context of real-time collaborative systems. A key concept in their work is the intention violation: a situation in which a player's operation is disrupted by a concurrent action, resulting in an outcome that no longer reflects the player's original intent. To address this, their framework emphasizes intention preservation, which ensures that each operation has the same effect as it would in a single-player scenario, even amid simultaneous edits.

In immersive VR environments, being able to see other collaborators' actions, like reaching for a knob or moving across the space, helps establish a shared sense of presence and timing. Visual cues can act as subtle invitations: in improvised music, for example, a glance, a nod, or infamous stank face—a term used by musicians to describe the exaggerated grimace of appreciation when someone plays something particularly good—can communicate affirmation. These nonverbal cues help shape the flow of interaction, enabling performers to respond in real time to each other's gestures, ideas, and affective states. Similarly, when one player sees another manipulating a control in a certain way, they might feel compelled to join in. These moments of visible interaction invite responsiveness, helping participants co-shape the texture and direction of the performance in real time.

MSVR was tested both in lab environments and at public events, including at Expo '74 (Zicarelli 2018), a conference for Max software users held at Mass MoCA in April 2019. Overall, participant feedback was highly positive.

Experts in modular synthesizers were also invited to evaluate the system. Players praised the layout, noting that the angled positioning of modules made them easy to see and interact with. Across sessions, players generally arranged modules in curved, body-oriented configurations rather than on flat planes or rigid grids. This spatial flexibility supported intuitive, immersive interaction. The way modules were placed over time also conveyed a kind of embodied memory, what Booth, R. Brown, and Muggs (2015) refer to as the "psychic residue" of creative environments.

Participants frequently remarked on the gestural controls used to manipulate modules. The "reeling in" gesture achieved by pushing the joystick along its y-axis to move a module along the z-plane toward the player was described as both effective and satisfying. Players could also rotate modules by moving the joystick along its x-axis: positive values rotated

the module to the right, while negative values rotated it to the left. This interaction was complemented by the inverse action, a "shoving" gesture, which was used to push modules away. Based on player feedback, an additional method for patching was implemented that allowed players to create a connection by pulling a module toward a hanging cable tip.

Participants remained comfortable during long sessions, with minimal reports of motion sickness, even during occasional system glitches. Contrary to earlier findings, the absence of a full-body avatar did not disrupt the experience[10].

Some interaction challenges were observed. Participants sometimes struggled with cable insertion due to the lack of tactile guidance. Players repeatedly asked for richer in-world visual feedback. Some requested an oscilloscope module to help with precise parameter control, while others wanted a way to trace a signal path from a given output back to its source. This latter feature was considered a natural extension of the existing graph-coloring system, and was flagged for future implementation.

Participants frequently reported that module names and parameter labels alone were not enough to communicate a module's functionality. In response, plans were made to integrate editable documentation into the VR environment.

In a final phase of player testing, we began recording players' GOTs. This allowed participants to explore the environment without being directly observed, with interviews conducted later based on session playback. This approach helped preserve natural engagement while still enabling detailed post-hoc evaluation.

3.7 Mischmasch

The development of MSVR exposed significant limitations in the underlying software platforms, including the deprecation of WebVR, which ultimately made the implementation unsustainable. In response, the project was re-planned from the ground up in Node.js, integrating graphics, sound, player input, and backend systems. This transition marked a major shift in direction, leading to the launch of a new version of the system under the name *Mischmasch*[11] (Wakefield, Palumbo, and Zonta 2020) (see Figure 3.6).

[10]Several participants reported that the visual representation of hand-held controllers—including their laser-pointer tips—provided sufficient feedback for interaction. Although Serafin et al. (2016) suggest body representation is essential for immersion, our players emphasized task-specific visibility over full embodiment.

[11]See https://www.youtube.com/watch?v=lpxfm5mGfG8 for a video demo of Mischmasch, included as part of our NIME2020 conference presentation.

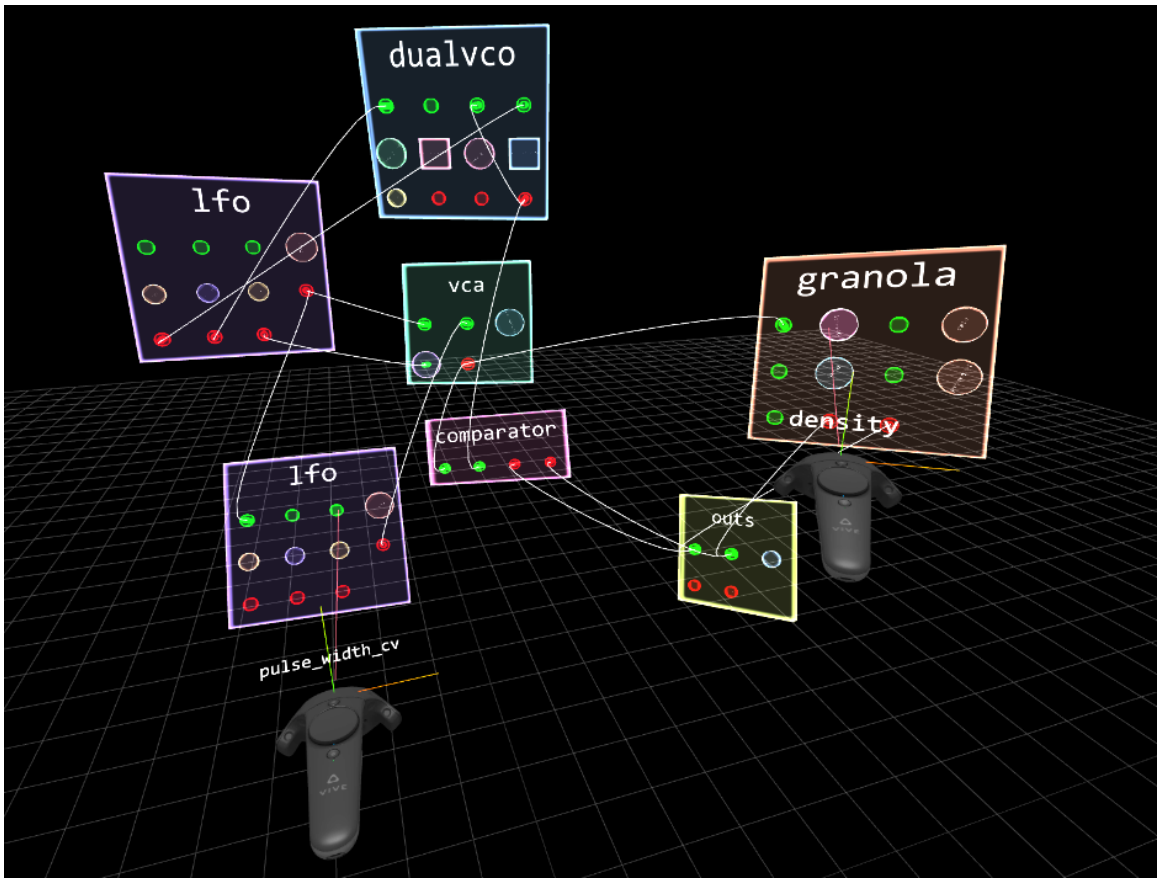


Figure 3.6: Player-side view in Mischmasch: when a player hovers over any element inside a module, its name appears for quick reference.

Graphics rendering was achieved using *node-gles3*[12] (Wakefield 2019), a native WebGL2 emulator for Node.js that utilizes OpenGL ES 3.0 through native windows on macOS, Windows, and Linux platforms. This library enables the execution of WebGL2 code in a native desktop context, facilitating integration with non-browser features and addressing bandwidth issues between C/C++ simulations and sensor code by running them in native Node.js modules alongside WebGL rendering. The new system retained the foundational goals of MSVR while addressing its technical limitations through a redesigned repository and architecture.

Genish.js[13] (Charlie Roberts 2016), a WebAudio library inspired by *gen~* [14], offered a solution to also migrate audio processing into a Javascript environment. Genish.js allowed for seamless audio graph changes during runtime and its support for offloading audio processing to audio worklets avoided disruptions due to visual rendering, making it suitable for the evolving Mischmasch system (see Figure 3.7.

3.7.1 Design Goals

The design of Mischmasch was guided by the goal of translating, rather than merely simulating, the experience of modular synthesis into a virtual reality context. Rather than replicating a physical rack-and-patch-cable setup, the project asked what modular synthesis might look like if it had originated in a virtual environment from the outset. This speculative orientation enabled a rethinking of traditional constraints. For instance, modules in Mischmasch are not tied to gravity or planar surfaces—they can be freely instantiated, repositioned, and arranged around the performer's body in three-dimensional space, encouraging personalized, ergonomic configurations that are impossible with physical systems.

Signal routing was similarly reimagined: all knobs double as modulation jacks, allowing for highly flexible control schemes, and cables are dynamic entities that stretch, snap to targets, and support stacking without signal degradation.

To reinforce the modularity of Mischmasch, every knob in the environment was designed to accept signal modulation directly: rather than assigning separate modulation inputs, we allowed cables to be plugged straight into the knobs themselves. This effectively flattened the

[12]Node-gles3 has since been renamed and expanded as anode_gl, which continues to be developed and maintained by Dr. Wakefield. For more information, see: https://github.com/grrrwaaa/anode_gl

[13]See http://www.charlie-roberts.com/genish/

[14]https://docs.cycling74.com/legacy/max8/refpages/gen

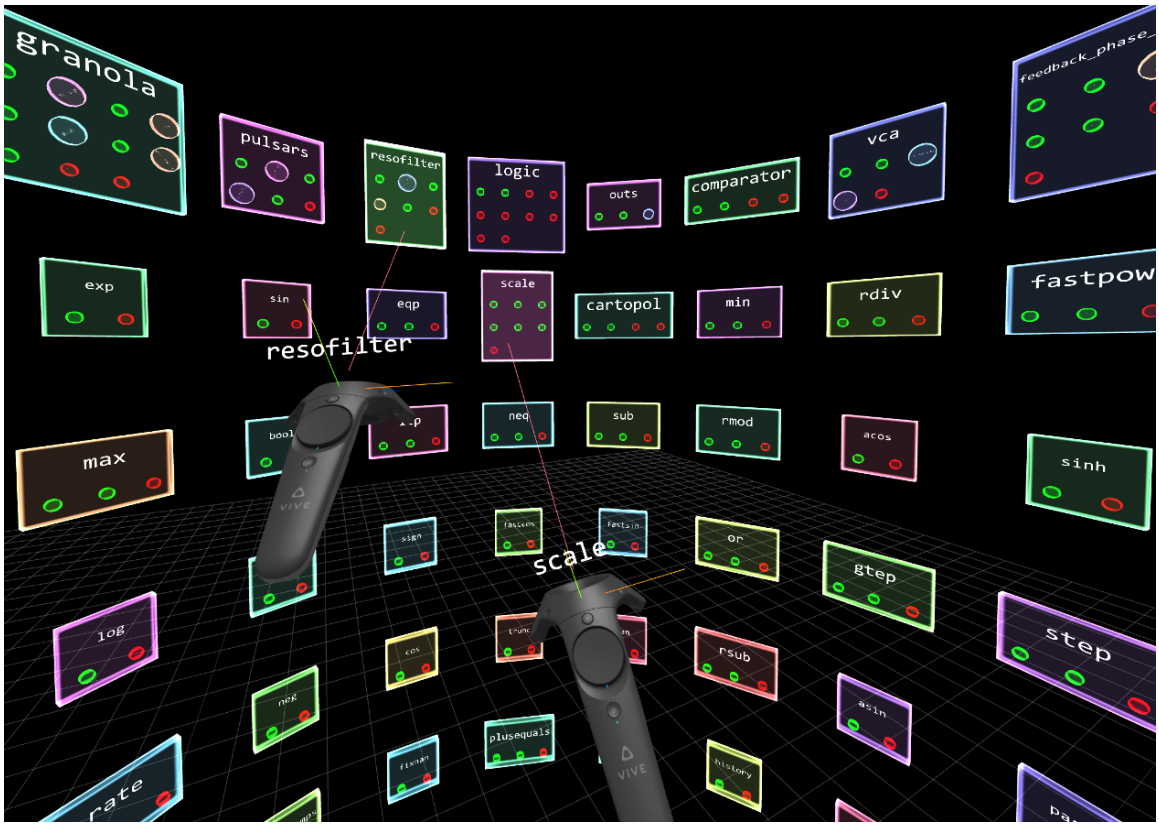


Figure 3.7: The Mischmasch *Palette* allows players to browse and add modules into the VR scene; every operator available in the genish.js library can be instantiated as a module.

interface's control schema: any parameter adjustable by hand could also be modulated via a signal, treating patch cables like virtual tendrils for gestural automation. This design echoes Jaron Lanier's concept of phenotropic software (Lanier 2003), in which modules influence each other not through rigid APIs but by simulating tactile manipulation, as if by virtual hands. Lanier proposed this model to improve the learnability, playability, and longevity of software instruments—qualities directly inspired by physical musical practice. In keeping with conventions from hardware modular synthesis, modulated knobs in Mischmasch don't animate or rotate visually; instead, the knob functions as an attenuator for the incoming signal, scaling its influence rather than being visually overwritten.

These design choices emphasized immediacy and embodied interaction, leveraging VR's unique spatial affordances to extend the expressive range of modular synthesis.

3.7.2 Transition to Automerge

As Mischmasch evolved, the team pivoted from the custom GOT system to Automerge, a mature and actively maintained implementation of Conflict-Free Replicated Data Types (CRDTs) (Kleppmann and Beresford 2017). While the GOT prototype successfully demonstrated collaborative graph editing, several delta operations—especially those involving fine-grained node movement and edge reconfiguration—had not yet been formalized or tested under concurrent editing. Completing the system would have required substantial additional development, particularly around conflict resolution and offline support.

Automerge provided a tested alternative that addressed these limitations. It offered causally consistent merging, support for nested data structures, and a peer-to-peer architecture that eliminated the need for a central server. This shift allowed the team to refocus on instrument design rather than protocol implementation, leveraging an existing CRDT framework to better support real-time collaboration.

Crucially, Automerge's local-first design ensured that each participant could continue editing even while offline, with changes automatically merged once connectivity was restored. This was particularly important in the context of modular synthesis, where co-located or distributed performers need uninterrupted access to the instrument—and where every interaction, even those made asynchronously, contributes to the evolving creative process. By adopting Automerge, Mischmasch moved closer to a system that supports multiplayer performance while preserving the continuity and legibility of patch history.

In Mischmasch, the Teaparty signaling server evolved into a proper peer-to-peer broker, facilitating WebRTC (W3C WebRTC Working Group 2021) connections between collab-

orators. Once a connection was established, WebRTC data channels were used to transmit changes made to the shared Automerge documents. This allowed two or more players to inhabit the same VR scene and collaboratively build or modify modular patches in real time, with edits synchronized directly between peers without relying on a centralized server.

Despite its visual and sonic richness, Mischmasch ultimately proved too expansive for a doctoral dissertation. Over the course of five years, the system underwent six full codebase rewrites—each iteration grappling with the compounded complexity of managing immersive graphics, networked multiplayer interactions, spatial audio, and real-time synthesis.

Since immersive VR was not essential to the central research problem of systems for capturing, transforming and re-performing musical processes over time, the next phase of the project set aside the VR complexities of Mischmasch in order to refocus on a more specific research goal: the need for modular synthesizer software that supports real-time improvisational version control.

This became the foundation for Forking Paths, a new system that retains the expressive and improvisational goals of its predecessor while stripping away excess complexity. Unlike Mischmasch, Forking Paths is lightweight, extensible, and accessible through the web. Forking Paths enabled a focus on the design of patch histories, the manipulation of versioned edits, and the musicality of branching and merging processes in a live setting.

3.8 Methodological Reflection

The methodological arc traced in this chapter—from collaborative Git repositories and projectional editors to VR patching environments—gradually distilled into a focused inquiry: how might the principles of version control serve as a compositional and improvisational interface in modular synthesis? One concept that emerged from this process is the patch history: a mechanism for capturing, navigating, and reinterpreting the lineage of a musical patch over time. In the final system developed for this dissertation, Forking Paths, patch histories function analogously to the Command design pattern (Gamma et al. 1993): each patch edit is stored as a discrete, timestamped action that can be replayed, reversed, or sequenced. Unlike Git's model of deltas embedded within linear or branching timelines, the patch history treats edits more like free-floating events—individually addressable and recombinable outside a strict lineage. This opens up not only undo/redo functionality but also branching, collaborative revision, and the re-performance of electronic music through historical states.

A practice-based, research-creation approach was the right fit for this project because the kind of instrument I sought to investigate did not exist. The only way to explore what real-time branching in modular synthesis might feel like was to build and perform with prototypes. This approach also aligns with my background as an artist, musician, and software developer, where creative inquiry has long proceeded through making. Prototyping and performing were not just ways of demonstrating ideas but ways of generating them.

This method also has clear limitations. It does not produce rigorous benchmarking, formal proofs, or controlled studies, and the knowledge generated is necessarily situated and sometimes idiosyncratic. Open-endedness made scope difficult to contain: projects like Mischmasch underwent repeated substantial rewrites, leaving some artefacts preserved only as partial repositories or documentation. These gaps reflect the improvisatory nature of the research, but they also mark its limits.

3.9 Comparative Analysis

The earliest projects, Using Git to Write a Paper About Using Git and Git Show, treated version control as a compositional device and emphasized the visibility of process. Forking Paths inherits these concerns directly: its patch-history architecture records every parameter adjustment, cable connection, or gesture as a versioned event, and allows players to reuse these traces as playable material to extend Git Show's conceit into performance.

Other projects, such as Alicenode Editor and Cards, were concerned with interfaces for repository navigation and code annotation. Alicenode introduced projectional editing and graph-based representations of structured states, a logic reappearing in Forking Paths' DAG visualization and node-based branching as integral parts of the instrument. Cards, meanwhile, foregrounded annotation as patch notes inscribed directly onto the synth surface.

The technical challenges confronted in MSVR and Mischmasch shaped the design choices of Forking Paths. MSVR stalled when Max/MSP's audio scheduler could not remain stable while synchronizing new or past states, while Mischmasch ran into limitations with other audio libraries. To address these issues, Forking Paths required a custom DSP engine tailored to the system's needs. Where MSVR attempted custom Graph OT, Mischmasch pivoted to Automerge, a mature CRDT framework that enabled offline editing, peer-to-peer synchronization, and robust conflict resolution. Yet Mischmasch also attempted too much—VR graphics, spatial audio, multiplayer, and synthesis—making it unsustainable despite six rewrites. Its key contribution was not immersion but the adoption of Automerge,

clarifying that VR was inessential. Forking Paths inherits Automerge's stability while narrowing its scope to improvisational version control as the central research problem. The following chapter examines Forking Paths in detail, outlining its architecture, features, and creative implications.

Chapter 4

Forking Paths

Forking Paths is the culmination of several years of research-creation exploring how VCS might reshape digital musical instrument design. As demonstrated across the projects described in the previous chapter—*Using Git To Write a Paper About Using Git*, *Git Show*, Alicenode Editor, Cards, MSVR, and Mischmasch—this work investigates how VCS might support improvisation, trace creative process, and scaffold collaborative authorship. This chapter introduces Forking Paths as the practical and conceptual synthesis of that lineage. It presents an overview of the interface, technical architecture, and conceptual foundation.

Forking Paths is a browser-based modular synthesizer that integrates software version control into the core of its design[1][2]. It enables musicians to patch, perform, and revisit every interaction as a material part of the composition process. Unlike conventional synthesizers, where the ephemeral changes within a performance are difficult to retrace, Forking Paths was designed to capture each modification—parameter adjustments, module connections, gesture movements—as a versioned event within a persistent and navigable timeline. This contributes a novel approach to musical instrument design by addressing a notable gap in modular synthesis: the absence of tools that support branching, merging, and sequencing patch histories.

4.1 Design Goals

The system's design was guided by three central goals: first, to foreground versioning as a creative interaction in its own right; second, to provide a patch history sequencer that made

[1]Forking Paths can be accessed on the web at https://forkingpaths.onrender.com

[2]The source code for Forking Paths can be acessed at https://github.com/michaelpalumbo/forkingpaths

navigation of past and future states a core musical technique rather than a peripheral feature; and third, to support free play and experimentation without the need for manual saving. These goals framed the specific technical decisions described below.

4.2 A Tour Of The Interface

The user interface of the app is organized across four distinct browser tabs, each of which is described in the subsections that follow:

1. **Lobby**
2. **Synth View**
3. **Patch History Window**
4. **Synth Designer**

4.2.1 Lobby

When players first open the app, they are presented with the Lobby page, which lists all active sessions, with player names and join buttons for each session (see Figure 4.1). The page also provides a very brief description of Forking Paths and a quick start guide.

4.2.2 Synth View

After joining a room, the Synth View opens in a new tab. The Synth View contains a modular synthesizer, a drawing tool, system settings, buttons to open other Forking Paths windows, access to the synth browser, and room status updates (see Figure 4.2).

The synthesizer interface is rendered as a directed graph rendered using the Cytoscape.js graph library (Franz et al. 2016), with compound parent nodes representing modules, child nodes representing jacks and parameters, and edges representing cable connections. Input jacks are shown as orange triangles, and output jacks as blue squares.

Cables

Cables can be created by clicking and dragging from either an input or an output jack, allowing flexible patching gestures during improvisation. However, once a connection is made, the system interprets all signal flow as moving from output to input, regardless of how

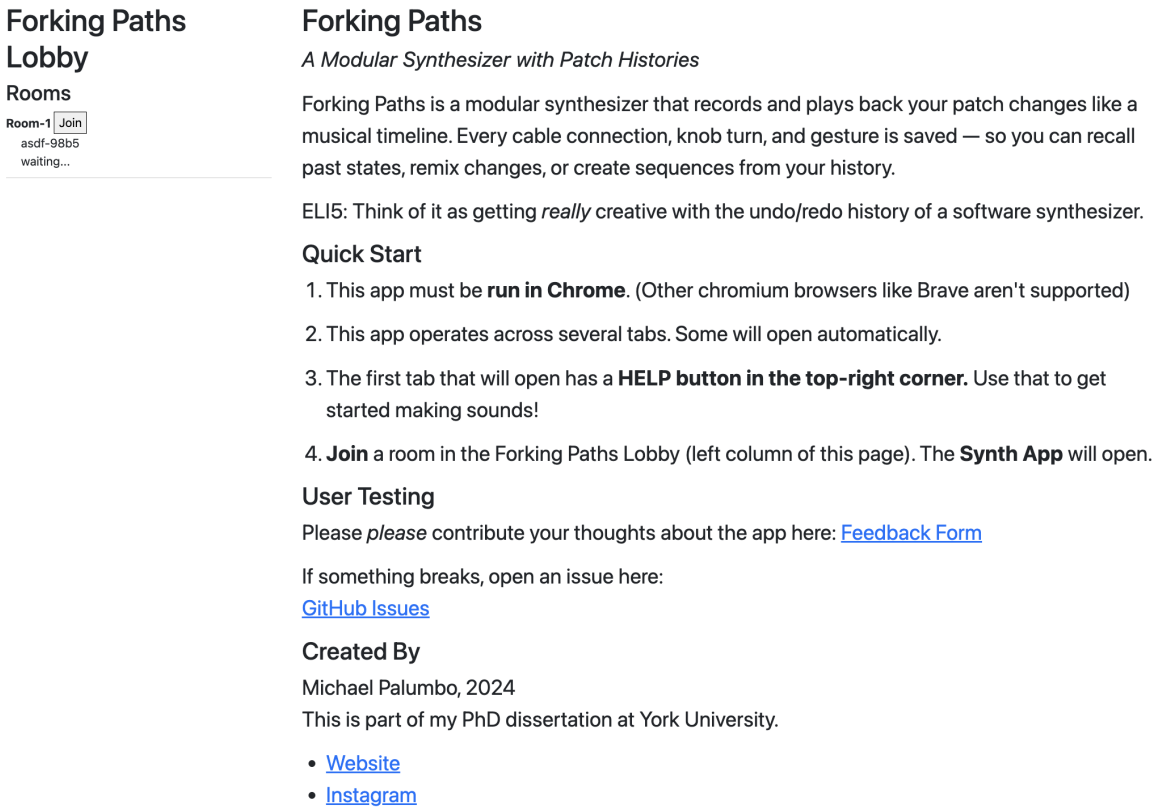


Figure 4.1: Forking Paths Lobby page

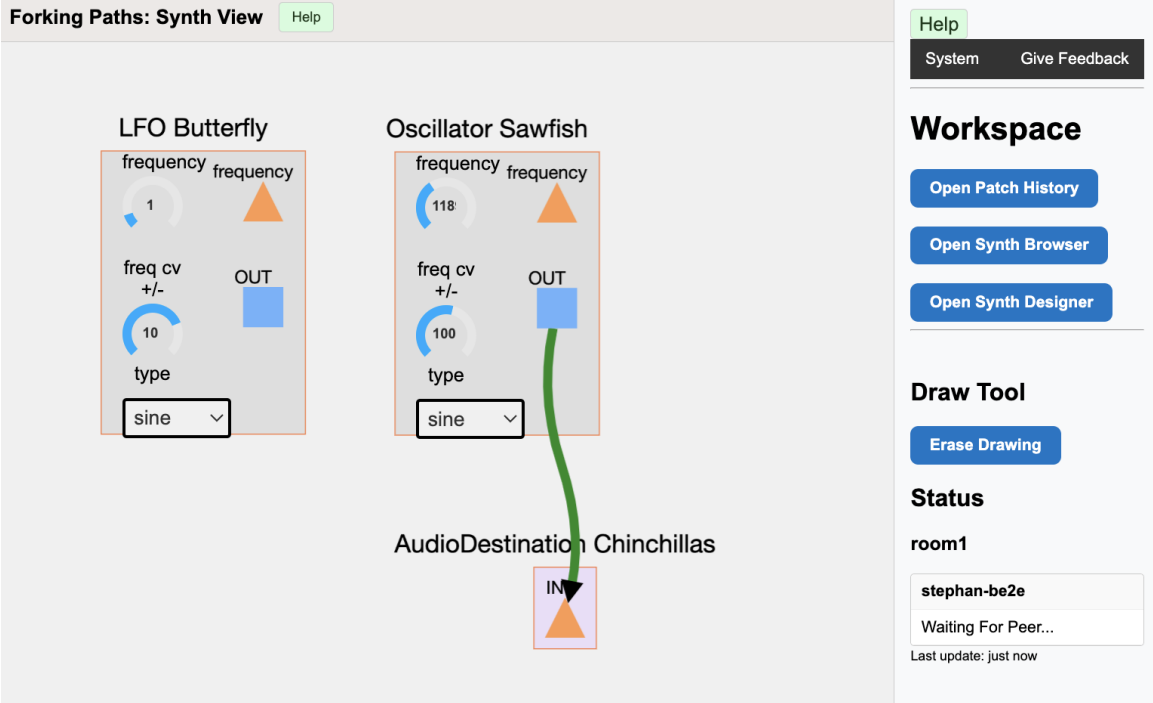


Figure 4.2: Forking Paths synthesizer interface.

the cable was drawn. To maintain clarity and consistency, tooltips and overlays describe connections using standardized terms: Output Module, Output Jack, Input Module, and Input Jack. This naming convention aligns with the underlying signal flow of the synthesizer, helping players quickly interpret patch structures without needing to recall the direction in which a cable was originally created. Cable colours are set randomly by default, but players can set them using keys 1 through 5 during their creation or modify them afterward.

Cables can be removed by first selecting them and then pressing the DELETE key. To repatch an existing cable, players can drag either of its endpoints away from the current jack and drop it onto another valid destination jack. This action removes the original connection from both the visual graph and the audio engine, ensuring that any signal previously routed through that cable is silenced. During the drag, the system renders a dotted line cable in place of the original, visually indicating the temporary disconnection. If the player abandons the action or attempts to connect to an invalid jack, the dotted line cable disappears.

Parameter Changes

Module parameters are represented by one of two UI elements: knobs or menus. Players control these parameters using the mouse. Menus open on click and allow selection from

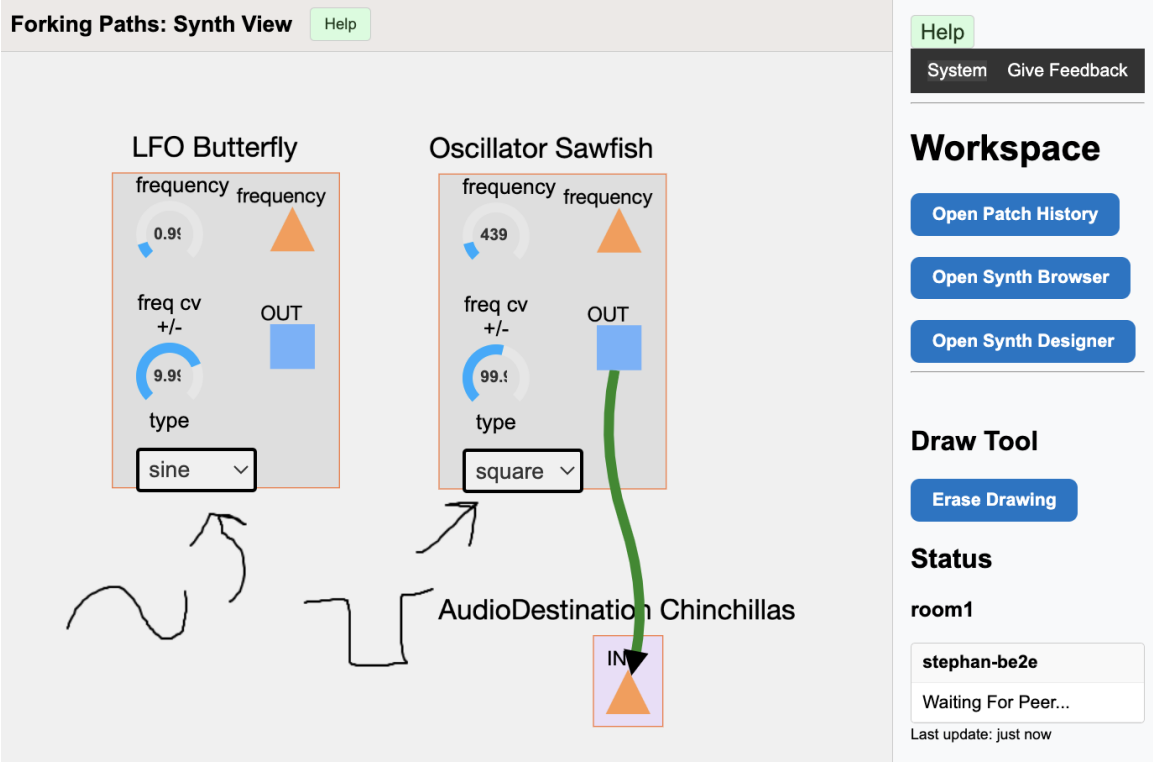


Figure 4.3: Making patch notes within the Synth View

a list of options. Knobs have two interaction modes: clicking anywhere within the knob's radius immediately jumps the parameter to that value, while clicking and dragging the knob's indicator adjusts the value continuously based on the vertical position of the mouse.

Patch Notes

Players can make patch notes directly on the screen by clicking and dragging anywhere within the grey area of the Synth View (see Figure 4.3). The drawing tool introduces marginalia within Forking Paths, offering players a way to annotate, reflect upon, and communicate within their work.

Players can load other synths using the Synth Browser, which lets them access `.fpsynth` files saved locally from the Synth Designer (see Section 4.2.4), or browse and load synths created by themselves and other players from the Forking Paths cloud database.

Synth Browser

Players can load different synthesizer layouts using the *Synth Browser*, which offers two options: loading a `.fpsynth` file from the local filesystem or browsing the cloud-based

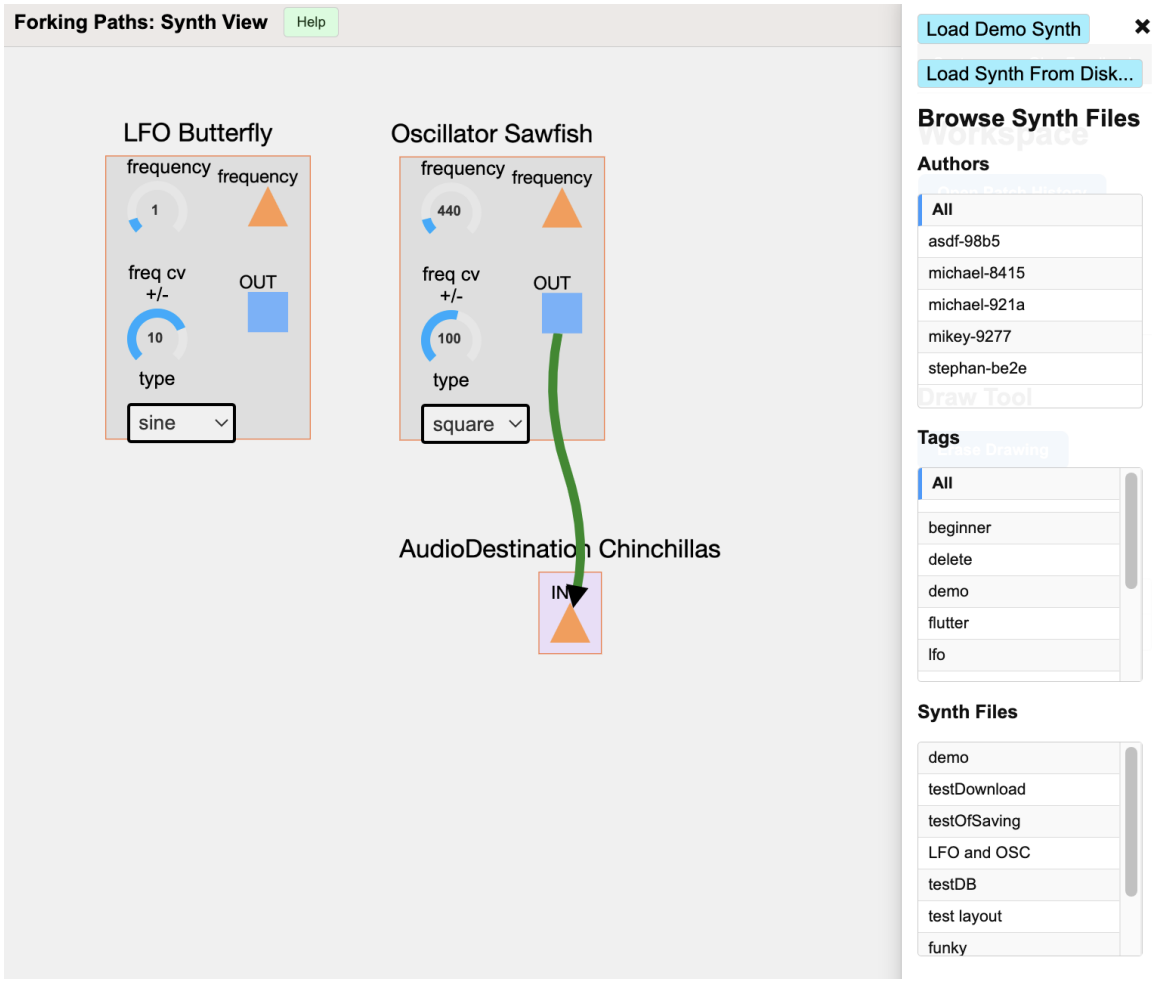


Figure 4.4: The Synth Browser allows players to load `.fpsynth` files from their local device or from the shared synth database, with filtering by author and tags.

Forking Paths synth database (see Figure 4.4). The database view allows players to filter files by author name or tags. All synth files are created using the Synth Designer Window (see Section 4.2.4).

4.2.3 Patch History Window

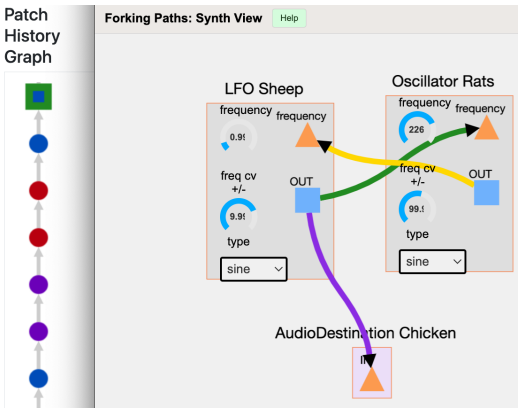
While the Synth View provides players with an interface for creating patches and manipulating synthesizer parameters, the Patch History Window offers a space to explore, organize, and reinterpret those changes. Every modification made in the Synth View is recorded as a discrete `changeNode`, which is stored in the patch history.

The Patch History Window (see Figure 4.5) presents these `changeNodes` in a directed acyclic graph (DAG) that visualizes the evolving structure of the player's creative process. Players can recall, branch, and merge past states; filter the graph to show specific types of changes; insert selected changes into a step sequencer for looped playback; and play and edit recorded knob turns. Patch histories can also be saved to disk, loaded from disk, or cleared to begin a new session.

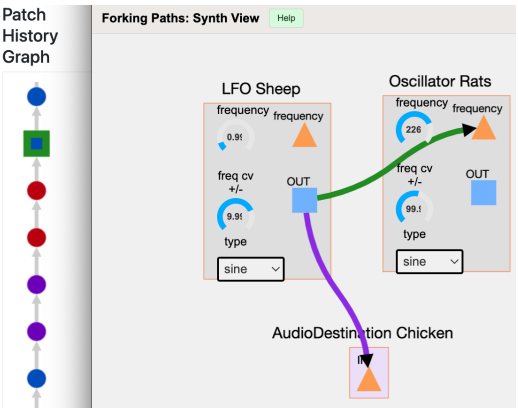
Figure 4.5: The Patch History Window. The left panel includes buttons for saving, loading, and creating new patch histories, as well as the patch history query tool for filtering `changeNodes` in the graph. The patch history DAG is displayed in the centre, with the timeline arranged from bottom to top. The patch history sequencer is located at the top-right corner, and the Gesture Editor is in the bottom-right. The currently selected `changeNodes` is indicated by a green square, and as it is a gesture `changeNodes`, the gesture data points are being displayed in the Gesture Editor.

Patch History Graph

The patch history DAG is displayed in the centre column of the window. Players can click on any `changeNode` to load a historical version, fork from that point, experiment recorded gestures, and view the results of merges between branches. Each node in the graph represents a discrete change, while the edges reflect the flow of time and authorship within the patch's evolving history (see Figure 4.6).



(a) Synthesizer state for the selected version



(b) Synthesizer state for the selected version

Figure 4.6: Two different patch states loaded from the patch history. Note the yellow cable is what changed from version *a* to version *b*

Node colors and styles are dynamically assigned to differentiate types of `changeNodes` (see Table 4.1). For example, Forking Paths distinguishes between two types of parameter-related `changeNodes`: `paramUpdate` and `gesture`. A `paramUpdate`, represented by a purple `changeNode`, represents a single, discrete change, where a player has clicked once within a knob's radius to set a new value. In contrast, a `gesture`, represented by a light blue `changeNode`, records a continuous movement, capturing a series of parameter changes made while the left mouse button is held down during a knob turn.

To keep the graph visualization clean and minimal, no labels in displayed alongside the `changeNodes`. Players can hover over a node to view a tooltip with more detailed information (see Figure 4.7).

Players can also navigate the patch history graph using MIDI CC messages, with values from 0–127 mapped to a chronological traversal of the version graph (see Listing C.3 in Appendix C).

Merges are achieved by dragging a `changeNode` over another `changeNode`, at which point the target `changeNode` is highlighted by a triangle outline (see Figure 4.8). Releasing

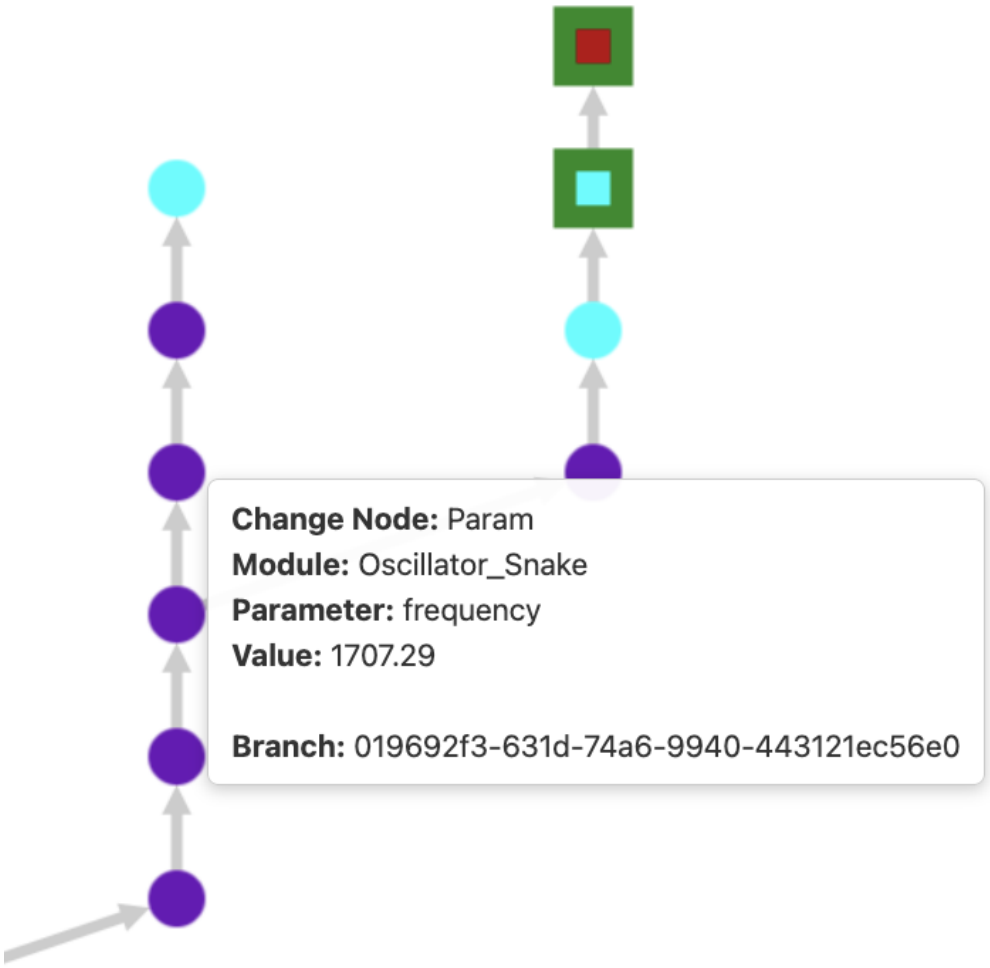


Figure 4.7: A `changeNode` tooltip provides players with information about each node, and is accessed by hovering over any `changeNodes` in the History Graph.

ChangeNode	Description	Colour
connect	Patch connection made between two jacks	Dark Blue
disconnect	Deleted patch connection	Red
paramUpdate	Discrete knob value change (single-click on a knob)	Purple
gesture	Continuous knob value change (2 or more values while mouse is held down)	Light Blue
blank_patch	A `.fpsynth` Synth file was loaded	Light Grey
merge	The result of merging two `changeNodes` together	Orange
sequence	Sequencer table data	Dark Green
draw	Single pen strokes	Light Brown

Table 4.1: `changeNodes` types with descriptions and colour codes

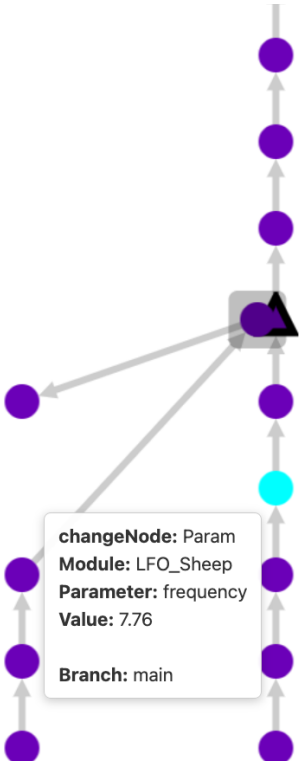
the mouse while two `changeNodes` are intersecting will result in a third `changeNode` being added to the graph, on a new branch, and whose parents are the original two `changeNodes`.

History Query Tool

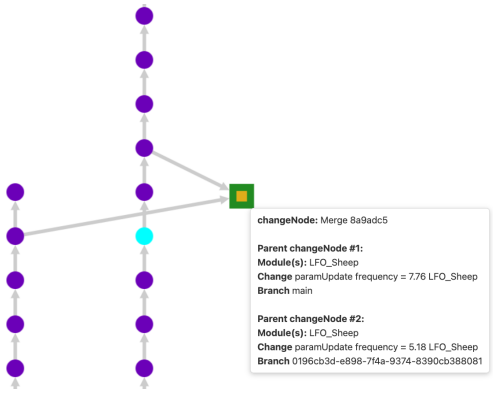
Players can query the patch history to filter out specific types of changes. This includes both structural graph-based filters—such as identifying leaf nodes, which are the newest `changeNodes` on any given branch—and content-based queries for parameter changes, recorded gestures, cable connections, and merge events. Additionally, players can isolate all changes associated with a particular module, offering a focused view of its evolution across time (see Figure 4.9).

Gesture Editor

The Gesture Editor allows players to view, edit, and reassign recorded parameter gestures—such as knob turns—within the modular synthesizer interface (see Figure 4.10). Gestures appear as timelines of points (value over time) and can be looped, reshaped, or applied to new parameters. Unlike the Synth View, changes made within the Gesture Editor are not automatically committed; instead, gestures are versioned only when the player chooses to save them. When a gesture is saved, it inherits its parent directly from the source gesture it was derived from, preserving the visual and structural lineage of transformations within the history graph.



(a) Dragging one node over another initiates a merge action. Releasing the mouse completes the merge. The target node is visually marked with a triangular border during hover.



(b) A completed merge, with the new merge `changeNodes` shown in orange and highlighted by the green square.

Figure 4.8: Merging two patch history states

The gesture is displayed in a breadth-first layout using the Cytoscape.js graph library with each parameter value in the gesture rendered as a node, connected by directional arrows moving left to right. The y position of each node reflects the parameter value, while its point in time is indicated by its position along the x axis. Along the left edge of the editor, the minimum and maximum values of the gesture are shown as labels, while the total length of the gesture is displayed to the right of the last node (in milliseconds or seconds, depending on length). When players hover over a node, its precise value is displayed in the editor toolbar. Gesture playback is triggered using the `Play` button (see Listing C.1 in Appendix C). As the gesture plays back, each node is highlighted in red in sequence, while unplayed nodes remain purple. Playback can be looped using the loop button.

Players can adjust gesture points by vertically dragging their position. The updated value is immediately sent to the audio engine and UI, allowing for live auditioning of these edits without writing them to the history.

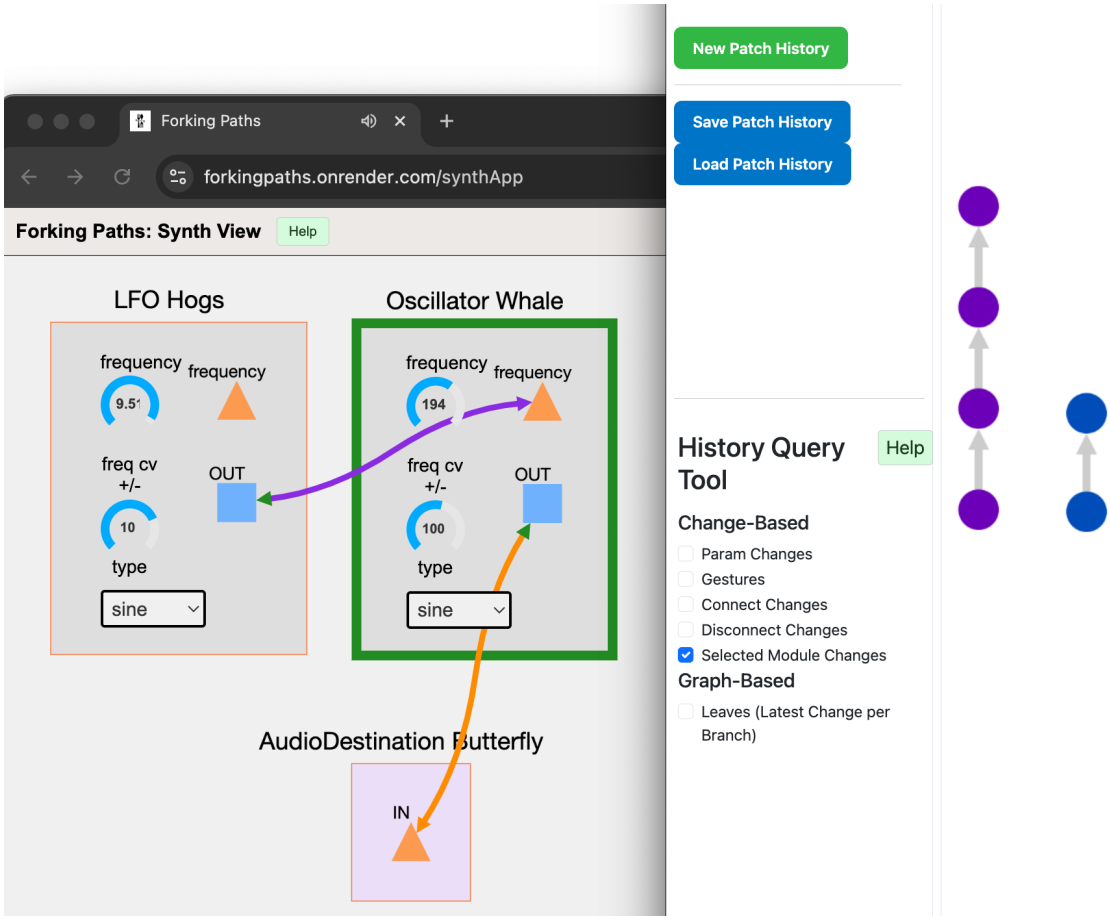


Figure 4.9: The History Query Tool. Here the player has chosen to isolate all `changeNodes`associated with a selected module (Oscillator Whale).

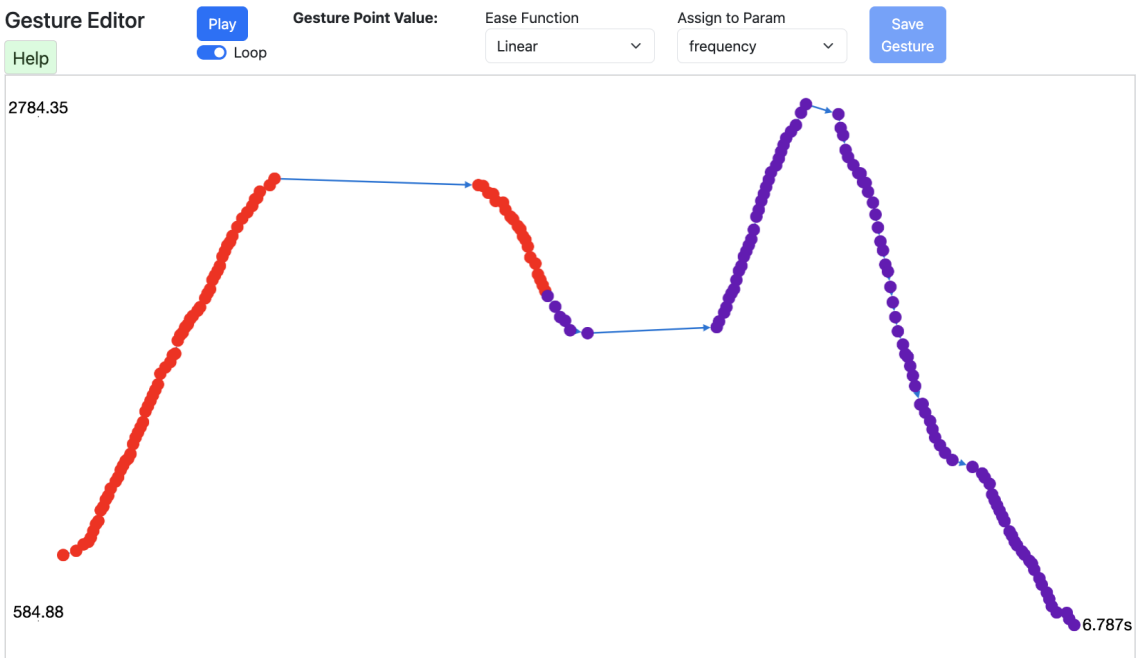


Figure 4.10: The Gesture Editor allows players to load, preview, and modify recorded gestures by editing their shape, applying easing, or reassigning them to new parameters. Each variation is saved as a new fork in the patch history. In this figure, the gesture's playback is being looped, and the red nodes indicate which data points have been played.

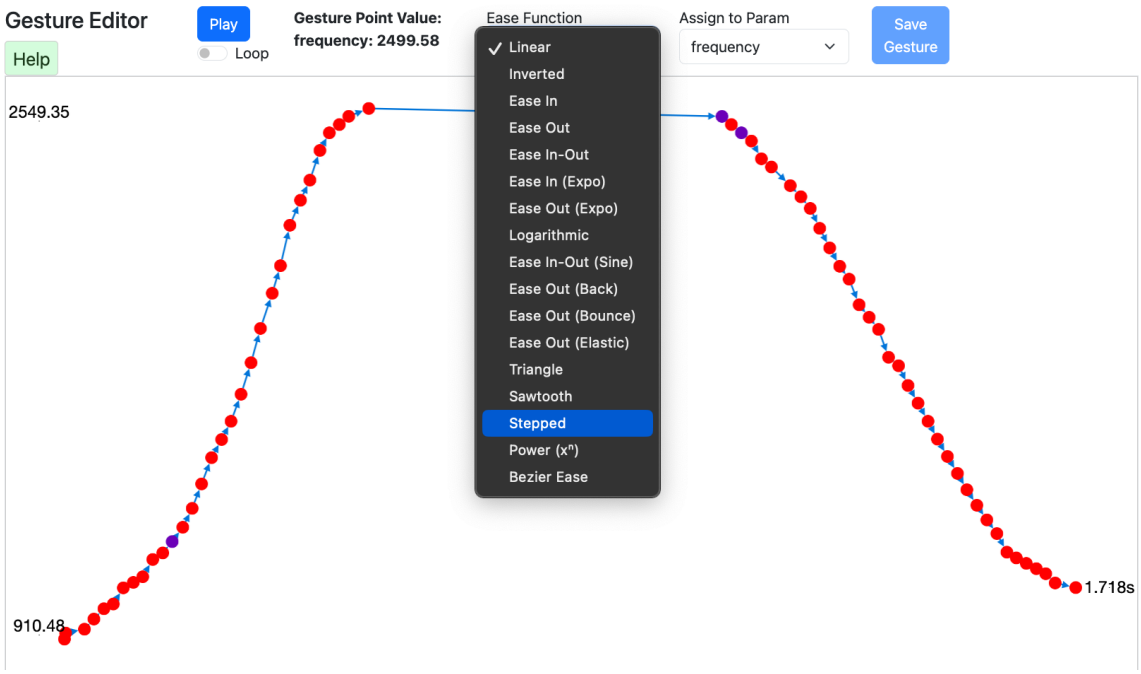
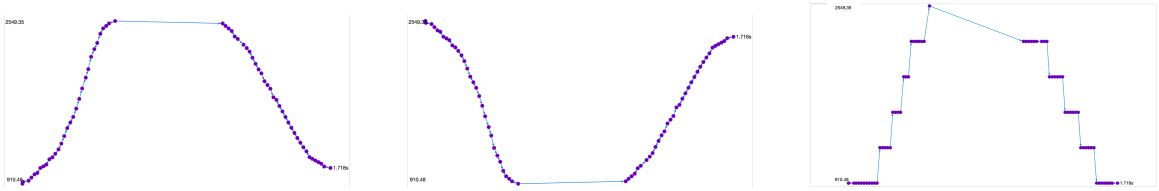


Figure 4.11: The available Ease functions in the Gesture Editor

Easing functions can be applied within the Gesture Editor (see Figure 4.11). These curves shape the timing and dynamics of the gesture, allowing players to smooth out abrupt knob turns or introduce intentional delays and accelerations, with each variation being able to be saved as a new gesture in the patch history (see Figure 4.12).



(a) Original (b) Inverted (c) Stepped

Figure 4.12: Three variations of one knob turn: the original, its inversion, and run through a stepped function.

Players can reassign a gesture to control a different parameter via a dropdown menu which by default displays the parameter associated with the loaded gesture. When opened, this menu lists all available modules (greyed out for visual grouping) and their associated parameters. Any parameter, whether continuous or discrete, can have a gesture assigned to it. Upon selecting a new target parameter, the original gesture is linearly scaled to the range of this new parameter (see Figure 4.13). For example, a gesture originally ranging from

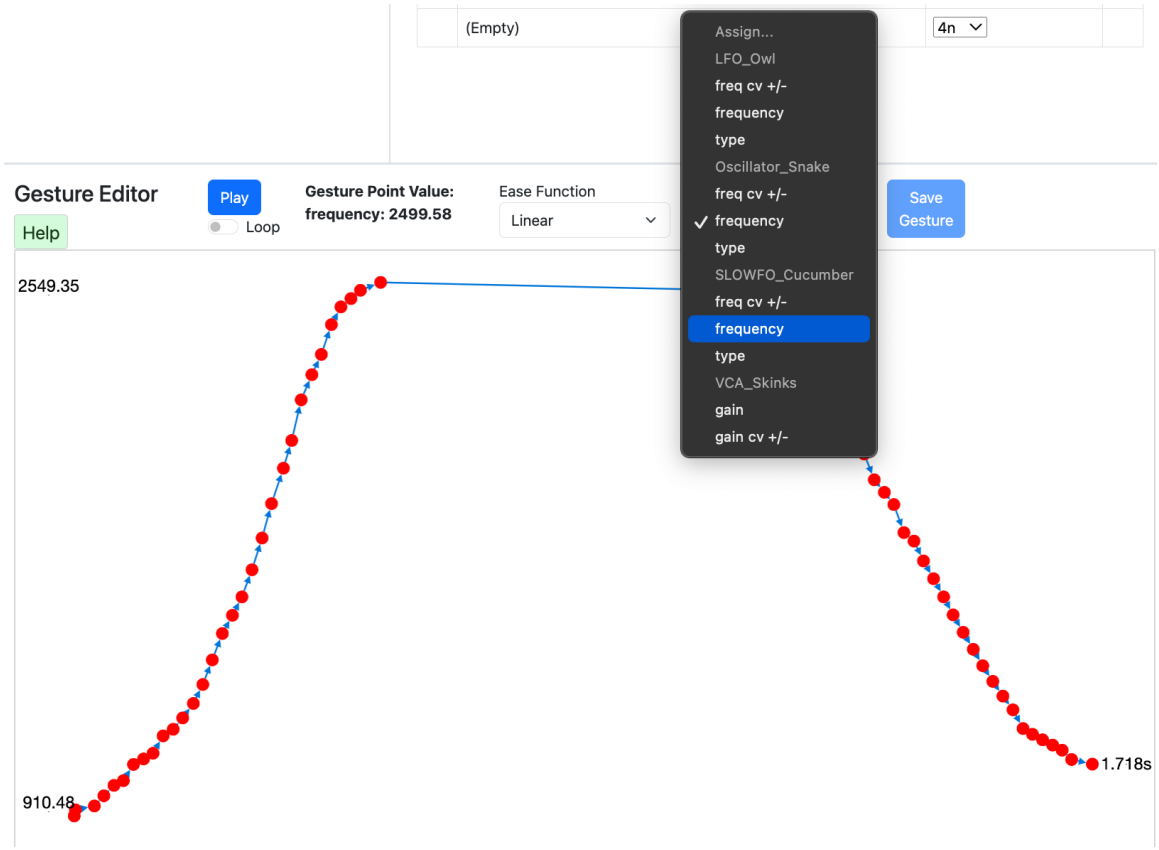


Figure 4.13: Gesture reassignment tool. Here a knob turn from the frequency parameter of the Oscillator with a range of 60-3000 is about to be assigned to the frequency parameter of the SLOWFO with a range of 0.01-10.

0–1000 can be mapped to a knob with a range of 0.1-20. Scaling is one-directional and doesn't overwrite the original.

Players can assign continuous control gestures, such as those derived from knob input, to menu-based parameters. The gesture is linearly interpolated across the discrete options of the menu. For instance, mapping a gesture controlling an oscillator's frequency onto its waveform selector causes the waveform to change over time based on the Y-range of the gesture. The minimum and maximum values of the gesture correspond to the first and last options.

Patch History Sequencer

The Patch History Window includes a sequencer that allows past patch states and parameter gestures to be treated as playable timeline events. Any change node can be added as a step

in the sequencer (see Figure 4.14). Step durations are set to quarter notes by default, but can be set by the player or be determined algorithmically based on the topology of the version graph. For instance, Euclidean distance between nodes in the DAG is used to interpolate step durations, mapping spatial distance to rhythmic values ranging from thirty-second notes to whole notes. Other metrics such as closeness centrality may also be employed to influence sequencing behavior, enabling structurally responsive playback of historical versions.

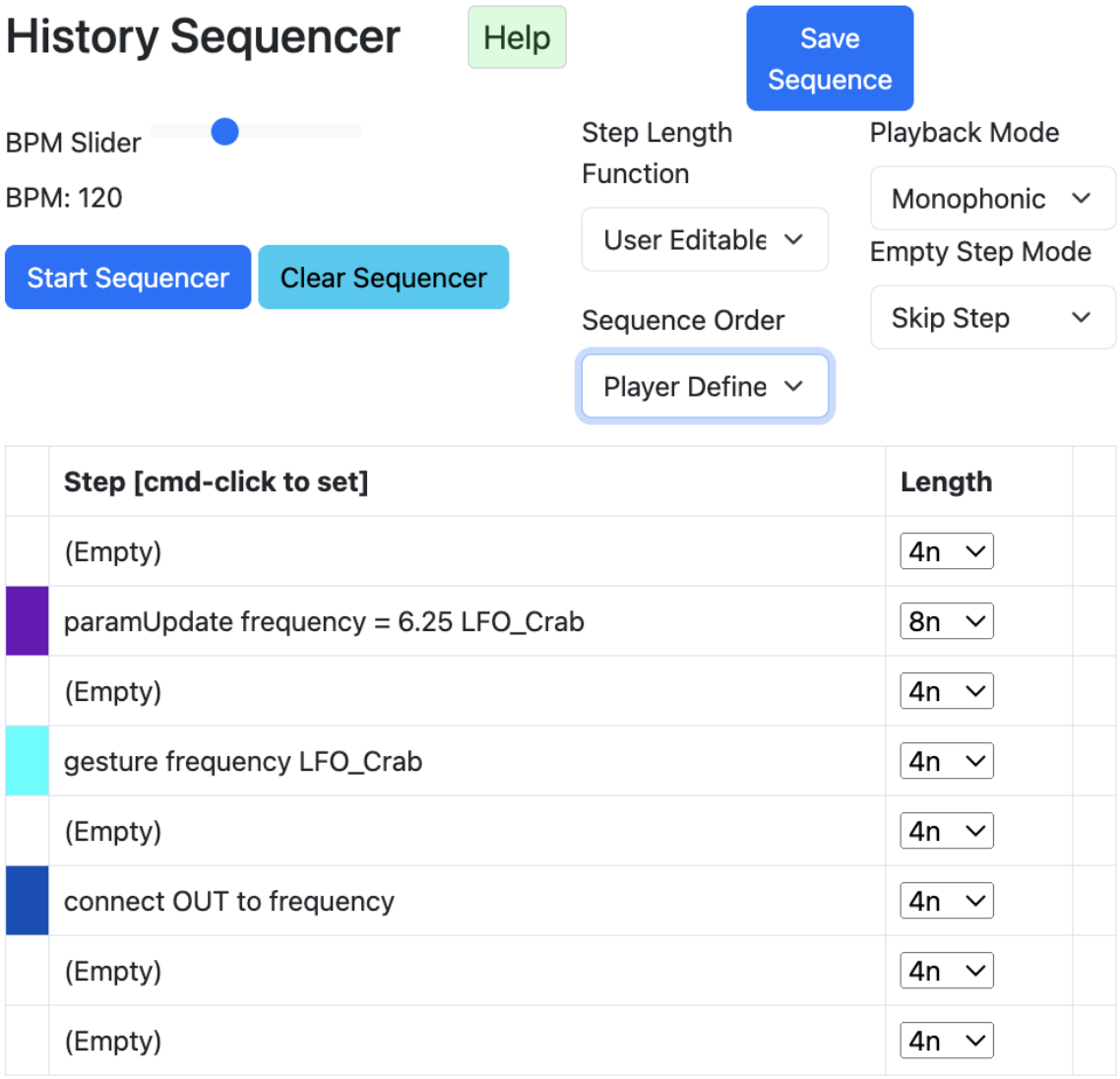


Figure 4.14: The Patch History Sequencer. Here the player has inserted three `changeNodes` into steps 2, 4, and 6, respectively.

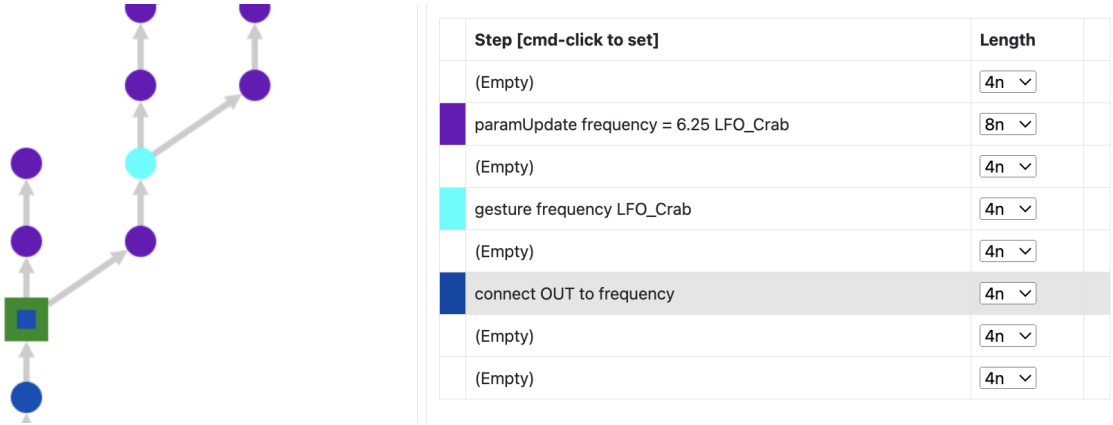


Figure 4.15: During sequencer playback, the current step is highlighted in grey, while its associated `changeNodes` is visually highlighted in the history graph, allowing players to follow the real-time traversal of patch history.

Gestures can be added as a sequencer step. Each gesture's original duration is displayed in the editor, and when assigned to a sequencer step, the entire gesture is quantized to fit within that step's length. This ensures that the full shape of the gesture is preserved, but played back at a faster or slower rate depending on the current BPM and the step's duration. This allows performers to repurpose expressive gestures rhythmically, creating dynamic modulations that can be scaled across different musical contexts. Individual gesture points can also be selected and added as a sequencer step.

Sequencer playback does not result in new changes added to the history, as the `changeNodes` are simply recalled when their step is triggered (see Figure 4.15). However, sequences can be saved to the history graph by clicking the *Save Sequence* button, allowing players to load different sequences back into the sequencer.

4.2.4 Synth Designer

The Synth Designer Window is an interface that players can use to create new modular synthesizer layouts which can then be loaded into the Forking Paths Synth View (see Figure 4.16). It comes with 11 modules that can each be added and arranged spatially. The layouts can then be saved to the player's computer with a custom file extension of `.fpsynth`, as well as stored in a database hosted by the Forking Paths server.

Edits made to the synthesizer were initially intended to be part of the main application, where adding, removing, or repositioning modules would register as `changeNodes` within

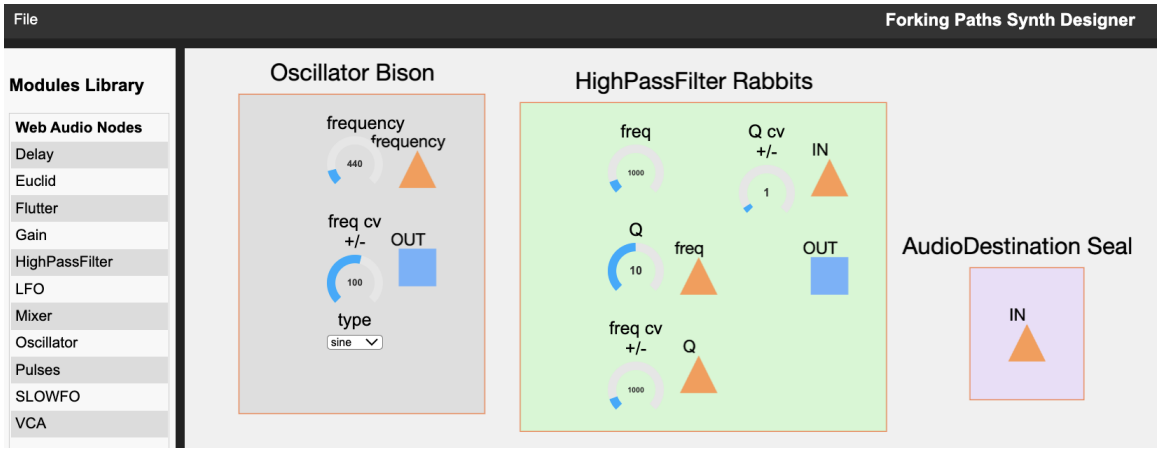


Figure 4.16: The Synth Designer Window.

the patch history. However, several technical constraints led to its separation into a standalone interface, as discussed in the Technical Architecture section below.

4.2.5 Help Overlays

Each of the aforementioned interfaces contain help overlays that are accessible via dedicated *Help* buttons. When activated, a panel appears over the side of the window that is opposite to the associated help section, allowing players to keep their attention focused on the interface while referencing instructions or explanations (see Figure 4.17).

4.3 Forking Paths VCS

4.3.1 Automerge Documents

An Automerge document is a data structure that tracks changes to JSON-like objects using conflict-free replicated data types (CRDTs) (Automerge 2017). Forking Paths maintains two active Automerge documents at all times: `patchHistory` and `currentBranch`[3].

[3]To support real-time collaboration and flexible history tracking in Forking Paths, the author initially experimented with *automerge-repo* (https://github.com/automerge/automerge-repo), a high-level wrapper around the core Automerge library designed to simplify peer-to-peer synchronization and document sharing. However, automerge-repo does not support branch management and did not expose fine-grained control over document synchronization, both of which were essential for the research goals of this project. As a result, it was replaced with the lower-level @automerge/automerge library, which provided direct access to patching, diffing, and sync logic. A custom branch management system was subsequently developed. The original implementation

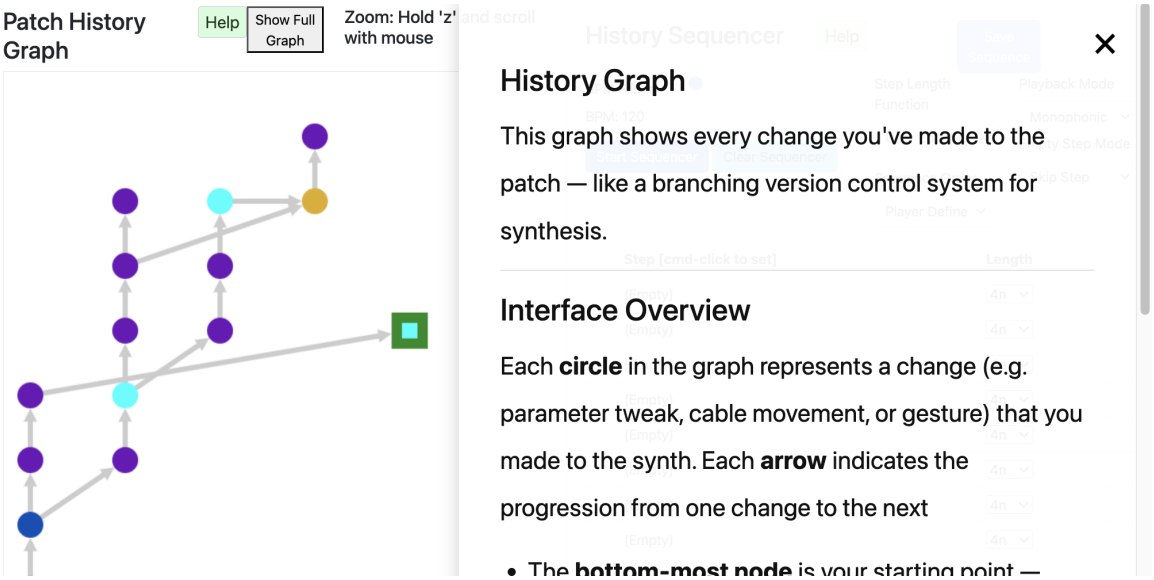


Figure 4.17: Each of the aforementioned interfaces includes a help overlay, accessible via a dedicated "Help" button. When activated, a panel appears on the side of the window opposite the relevant interface section, allowing players to stay focused on their interaction while referencing instructions or explanations.

patchHistory

The `patchHistory` document functionally operates like a VCS repository: it stores a copy of all branches, maintains parent-child relationships between those branches, and contains player and session metadata.

The `patchHistory` document contains the following properties:

- `patchHistory.branches`: maintains a record of all branch names, and how those branches connect to each other.
- `patchHistory.docs`: stores binary-encoded Automerge documents keyed by branch name.
- `patchHistory.head`: tracks the current branch name and the UUID hash for its most recent `changeNode`.
- `patchHistory.userSettings`: holds global preferences for the player's interface and behavior.

using automerge-repo is archived at
https://github.com/michaelpalumbo/forkingpaths/tree/legacy/automerge-repo

- `patchHistory.synthFile`: stores the `.fpsynth` file currently loaded into Forking Paths.

currentBranch

The `currentBranch` document represents the branch that the player is actively engaged with, and it is both read from and written to patchHistory.docs. Any changes made to the synthesizer—such as adding modules, modifying connections, or adjusting parameters—are recorded as new `changeNodes`within the `currentBranch` document.

Document Initialization

Upon loading the Synth View, the `startAutomerge()` function handles the initalization of the two documents (see Listing B.1 in Appendix B). If the player is joining an active multiplayer room, a new `patchHistory` document is initialized and kept ready for synchronization with peers—for an in-depth explanation of multiplayer initialization, see Section 4.10.1. Otherwise, the system attempts to load a previously saved `patchHistory` document from an IndexedDB instance. If no saved patch history exists, it creates a new `patchHistory` object with default metadata and structure, and then saves it in IndexedDB.

If the `patchHistory` contains any `changeNodes`, then its most-recent branch is loaded into currentBranch, and the synth graph and visual interface are rehydrated accordingly. If no `changeNodes` exist, a new `currentBranch` is initialized and the Forking Paths demo synth demo.fpsynth is added as the first `changeNode` in the history.

This staged initialization ensures Forking Paths can either resume previous sessions, enter collaborative editing contexts, or launch new projects from the demo synth template.

4.3.2 Creating New Patch Histories

A new patch history can be created in one of three ways: automatically during Automerge document initialization for first-time users, manually by the player via the New Patch History button, or when a different Forking Paths synth file is loaded. In each case, the `createNewPatchHistory()` function is responsible for resetting the system state and initializing a fresh patch history session (see Listing B.2 in Appendix B). It begins by clearing all existing visual, audio, and collaborative data—including the drawing canvas, Cytoscape.js synth graph, DSP state, and patch history sequencer—and removing all dynamically generated UI overlays. It also deletes the current `patchHistory` document from IndexedDB and notifies the server and any connected peers to clear the patch history DAG.

A new `patchHistory` document is then created, incorporating either the previously loaded `.fpsynth` file or, if the user is new, a default `demo.fpsynth` template. A corresponding empty `currentBranch` document is initialized and populated with the contents of the synth template, including its audio graph and Cytoscape.js node structure. This initial state is recorded as the first `changeNode` in the new branch history with the label of `blank_patch`.

The synthesizers visual graph is then reloaded into Cytoscape.js, UI overlays are regenerated, and the audio graph is passed to the DSP engine. Finally, the newly created `patchHistory` document is saved locally and sent to the server, which renders the patch history DAG for display in the Patch History Window.

4.3.3 Applying Changes to the Current Branch Tip

When a player modifies a synth patch, such as adjusting a module parameter (see Listing B.3 in Appendix B), the edit is applied to the `currentBranch` document using the `applyChange()` function (see lines 4-59 of Listing B.4 in Appendix B). For the purpose of introducing this concept, this section assumes the player is working at the tip of an existing branch—that is, at the most recent `changeNode` in that branch's linear history.

The function captures the hash of the previous `changeNode` on `currentBranch`, which serves as the parent identifier for the new change. An Automerge change message is then generated and passed to Automerge.change(), which applies the actual mutation to the `currentBranch` document. Once the document has changed, a new hash is computed, representing the new `changeNode`.

The function next encodes the updated `currentBranch` as a binary blob and stores the result in `patchHistory.docs`. Metadata such as the current hash and timestamp are also updated in `patchHistory.head`.

Finally, a callback is triggered to sync the updated state with any connected peers, refresh the synth visual state, pass the change to the DSP engine, and redraw the patch history graph, as can bee seen in Listing 4.1 below.

```
1 // the onChange Callback
2 onChange = () => {
3     // send to peer(s)
4     sendSyncMessage()
5     // set docUpdated so that indexedDB will save it
6     docUpdated = true
7     // update the historyGraph in the server
```

```
8       reDrawHistoryGraph()
9       // set audioDirty flag
10      if(audioGraphDirty){
11          audioGraphDirty = false
12  }
```

Listing 4.1: The onChange callback is triggered after a successful `patchHistory` document update within the function `applyChange()`. It handles peer synchronization, flags the document for saving to IndexedDB, updates the server-rendered history graph, and clears the audio dirty flag if needed.

4.3.4 Recalling Versions

When a player clicks on a `changeNode` in the patch history DAG, its branch name and hash are passed to the `loadVersion()` function, which then loads the corresponding state from the patch history and restores the associated synth configuration and/or patch history sequencer state (see Listing B.5 in Appendix B).

When `loadVersion()` is called, it uses `Automerge.view()` to reconstruct the patch state at the given `changeNode`, producing a read-only snapshot called `historicalView`. This snapshot contains everything necessary to rehydrate the synth, including the audio graph, visual node layout, and pen tool drawing (if present).

Next, the audio graph from the loaded version is then passed to the DSP engine via the `updateSynthWorklet()` function (see Listing B.6 in Appendix B). The handling of these messages within the DSP engine is covered later in Section 4.8. The visual layout is loaded into the Cytoscape.js interface using `updateCytoscapeFromDocument()` (see Listing B.7 in Appendix B).

4.3.5 Automatic Branching

Because edits in Forking Paths are only ever added to the tip of a new or existing branch, loading a previous version requires the system to anticipate the possibility that the player will make a new change from that historical point. If they do, their changes must not overwrite the existing history but instead create a new, divergent path.

The function then updates the `patchHistory.head` pointer to reflect the current position in the patch history. If the selected version is the latest `changeNode` in its branch, then no additional branching behavior is required. In this case, `automergeDocuments.newClone` is set to `false`, indicating that any subsequent changes will continue to extend the current

branch. When this condition is met, the `applyChange()` method described in Subsection 4.3.3 can proceed by appending a new `changeNode` to the current branch without forking.

However, if the player selects a version that is not the latest `changeNode`—whether from the current branch or from a different one—then `automergeDocuments.newClone` is set to `true`. This flag signals that the system should prepare to fork a new branch if the player makes a change after loading the historical state. In the case of editing a past commit on the same branch, the function also clones the `historicalView` using `Automerge.clone()`, ensuring that any subsequent edits do not overwrite existing history but instead generate a new branch from that point.

4.3.6 Making Changes After a Recall

If a player begins editing after loading a `changeNode` that is not the tip of a branch, then Forking Paths must create a new branch to preserve the integrity of the existing patch history. In this scenario, the `applyChange()` function detects that `automergeDocuments.newClone` is `true` and initiates a branching workflow (see lines 60-125 of Listing B.4 in Appendix B).

The system clones the previously viewed document using `Automerge.clone()`, ensuring that any changes are applied to a fresh copy rather than overwriting the past. A new branch name is then generated using a UUID hash, and the initial change is applied to the cloned `currentBranch` using `Automerge.change()`. A new hash is generated for this change, and a timestamped `changeNode` is created.

This new `changeNode` is stored as the head of the newly created branch in the object `patchHistory.branches`. The new branch entry includes the current hash, a pointer to its parent hash, and an array containing the initial history entry.

The `currentBranch` document is then stored as a binary blob in `patchHistory.docs`, and the `patchHistory.head` pointer is updated to reflect the new branch and its head. The branch name is also added to `patchHistory.branchOrder`, which preserves the insertion order of branches for use in the patch history DAG.

Finally, `onChangeCallback()` is called to trigger peer synchronization, UI updates, and a redraw of the patch history graph. The `automergeDocuments.newClone` flag is reset to `false`, allowing subsequent edits to continue extending the new branch linearly.

4.3.7 Command-Driven Patch History

While this branching infrastructure bears a strong resemblance to Git, the patch history in Forking Paths also draws from the Command design pattern, where each player action is stored as a discrete, reversible operation. Historically used to implement undo/redo systems, the Command pattern treats each change as a standalone instruction that can be replayed or reversed. Unlike Git or OT, Forking Paths does not rely on computing or inverting deltas in order to navigate version history. In Git and OT, moving from one version to another—such as from a newer state *A* to an older state *B*—requires applying the inverse of all deltas that occurred between those points, as shown in Figure 4.18 and Table 4.2.

Original Deltas: $A \rightarrow B \rightarrow C$

Figure 4.18: Forward sequence of original patch deltas

Step	Action (Original Delta)	Inverse Delta
1	Add cable from Osc → Filter	Disconnect cable Osc → Filter
2	Add cable from Filter → Output	Disconnect cable Filter → Output
3	Disconnect cable LFO → Osc	Add cable LFO → Osc

Table 4.2: Patching deltas

In systems based on deltas, reverting to an earlier version would require applying the inverse actions in reverse order, as shown in Figure 4.19

Inverse Actions to Undo: $C^{-1} \rightarrow B^{-1} \rightarrow A^{-1}$

Figure 4.19: Inversion sequence to revert patch to earlier state

Forking Paths, by contrast, stores a complete Automerge document at each `changeNode`. This allows the system to rapidly load any historical version without traversing or recomputing intermediary operations. This distinction reinforces the alignment with the Command Pattern, in which the emphasis lies on storing actionable, replayable states rather than chained deltas requiring ordered application and inversion.

4.3.8 Merging

The `createMerge()` function handles the merging of two `changeNodes`as selected by the player (see Listing B.8 in Appendix B). When a merge is triggered, Forking Paths loads the

two documents associated with each `changeNodes` from the `patchHistory`, merges them, and applies an empty change using `Automerge.emptyChange()` to flatten the result into a single new patch state.

4.4 Patch History Visualization and Navigation

This section describes how Forking Paths renders the patch history using an interactive DAG, and outlines the architecture of the Patch History Window and server-side rendering optimizations.

The concepts and operations described in the previous sections are made accessible to players through the Patch History Window—a separate browser window that renders the version history DAG in real time. This window is an interface for navigating, replaying, and branching from past versions of a patch.

History Graph

Forking Paths represents patch history as a DAG, where nodes correspond to individual change events and edges denote either sequential changes or parent-child relationships across branches. The graph is displayed using Cytoscape.js (Franz et al. 2016) and synchronized across clients using both inter-window messaging and WebSockets.

Whenever the `patchHistory` document is updated in the main window, the new DAG is serialized and sent to the Patch History Window, which then offloads the rendering of the graph to the Forking Paths server (see Section 4.9.2).

The use of Cytoscape.js allows for fluid graph rendering and interactive zoom, pan, and selection behaviors. The graph is dynamically re-laid out on updates using a layout algorithm that prioritizes readability.

4.5 Patch History Sequencer

This section describes how historical patch states and gestures become playable musical material through Forking Paths' integrated sequencer. Housed within the Patch History Window, the sequencer enables players to structure nonlinear histories into repeatable, time-based forms of performance and composition.

4.5.1 Sequencer Interface and Step Logic

The Patch History Sequencer is an 8-step sequencer that enables players to build rhythmic structures from the branching patch history. Each step can contain a `changeNodes` from the patch history DAG, including:

- `paramChange`: discrete parameter changes.
- `gesture`: knob turns involving two or more `paramChanges` while the left mouse button is held.
- `connect`: cable connections.
- `disconnect`: cable disconnections.
- `Sequence`: previously saved sequences that can be embedded within a single step.
- `blank_patch`: the first state of the synthesizer after loading the `.fpsynth` file, which effectively produces no audio. Useful for creating rests.

Steps are populated by selecting `changeNodes` in the history graph and assigning them to specific table rows (see Listing C.4 in Appendix C). When a `changeNode` is added, its background colour is also added to the sequencer step. Each row stores metadata about the assigned node (e.g., ID, label, duration), and is considered active once populated. During playback, each step is triggered sequentially using a loop driven by Tone.js (Mann 2014), which is used here exclusively for its transport and scheduling utilities, not for audio synthesis.

As the sequencer runs, it dispatches commands to load the associated patch version, updating both the interface and the underlying audio engine. Step durations are user-definable but can also be algorithmically generated, as detailed below.

In developing the ability to embed one sequencer inside another in Forking Paths, a design oversight became apparent around how gesture data is stored and accessed. Most `changeNodes`—such as parameter updates or cable connections—encode sufficient information in their labels, such as the branch and `changeNodes` ID, to recall the correct version from the synth. Gesture `changeNodes`, however, depend on external gesture point data that isn't included in the sequencer's saved step array. This is partly by design: gestures have no fixed length, and a single 5-second knob turn could contain hundreds or thousands of data points. It would be inefficient,and conceptually inappropriate,to store this level of detail

directly within the Cytoscape.js graph, nor within a given sequencer step. As a result, when a sequence is saved as a `changeNodes` and later embedded as a step, any gesture steps within it lack the data required for playback unless that information is retrieved separately. This limitation prompted a rethinking of how and when gesture data should be accessed during playback.

The solution points toward a strategy known as lazy hydration, which involves deferring the loading of large or complex data structures until the moment they're actually needed. In the context of web development, lazy hydration is used to delay the reactivation of UI components until user interaction requires them, improving performance and reducing initial load time (Oberlehner 2020). In Forking Paths, this concept is adapted to the domain of real-time music systems: when a sequencer `changeNodes` is added to a step, the system can scan its embedded steps for gesture `changeNodes` and request their full gesture point data using only their unique IDs just prior to playback. This approach keeps the sequencer structure lightweight and performant, while still ensuring that all necessary data is available when it matters. Lazy hydration thus allows for a clean separation between the structural representation of a sequence and the high-resolution data required for expressive playback.

4.5.2 Gesture Quantization

In addition to patch states, gesture nodes—representing parameter sweeps like knob turns—can be sequenced directly. Each gesture has an original recorded duration, which is displayed in the gesture editor and automatically quantized to match the duration of its assigned step. See Listing C.2 for how this works.

Quantization is implemented via a linear scaling function, ensuring the full gesture curve is preserved even when played faster or slower. Gesture playback is handled with setTimeout-based scheduling, aligning each parameter update to the BPM and current step duration.

4.5.3 Sequencer Modes

Monophonic and Polyphonic

The sequencer in Forking Paths operates in two distinct modes: monophonic and polyphonic. In monophonic mode, the sequencer advances through its rows in a linear, step-by-step fashion, with each step's duration determined by its assigned note length (see Listing C.5 in Appendix C). In contrast, polyphonic mode treats each row as an independent loop: each

step loops at its own rate, determined by its note length (see Listing C.6 in Appendix C). For example, a step assigned a length of a sixteenth-note will loop four times within each beat, and a step assigned a length of an eighth-note would be triggered twice within a beat. The result is an overlapping, asynchronous stream of multiple recalls of patch states.

In both modes, if a step includes a gesture, that gesture is quantized to the step length, and plays back in its entirety for the duration of the step. In polyphonic mode, a glitchy and sonically rich behavior emerges when one or more gestures are looping alongside discrete changes (such as parameter value changes or cable connections/disconnections). As gesture playback continuously updates a parameter value—even as that same parameter may be targeted by a discrete patch change—the gesture modulates against a potentially shifting background. This creates an interplay where gesture values temporarily override or respond to state changes that are also unfolding within the looped patch history. The result is a complex texture of modulation and state transitions where steps overlap and interfere with each other.

Internally, the sequencer is implemented using Tone.js and a hybrid scheduling system. In monophonic mode, a single Tone.Loop iterates through the steps in order, updating its interval at each iteration based on the next step's stepLength. In polyphonic mode, each row spawns its own Tone.Loop, looping independently at its own rate. These loops access a shared table of step definitions named `storedSequencerTable`, which includes patch state references, fully hydrated gesture data, and step lengths. Gesture playback is handled by a separate modulation engine that reads through the gesture point values to continuously update its parameter values over the length of the step. Because gesture modulation and patch state recalls both operate on the same parameter space, no prioritization exists between them—resulting in a system where gestures can override, or be overridden by, other recalled changes. This lack of separation allows intentional interference, and creates a feedback loop in which the system keeps recontextualizing modulation over dynamically recalled values, especially in polyphonic mode where asynchronous triggers multiply the complexity.

Direction

The playback direction of the sequencer defines how the sequencer advances through the eight available steps:

- **Forward**: Plays steps in fixed order from 1 to 8. This mode mimics traditional step sequencers and is ideal for player-curated patterns.

- **Topological Sort**: Orders the steps based on the position of each step's corresponding change node in the patch history DAG. This mode uses a graph traversal algorithm to determine chronological relationships between `changeNodes`. In other words, the sequence order depends on when a given step's `changeNodes` was made.
- **Random**: Chooses the next step randomly at each iteration.

Step Length Functions

Each step's duration is defined by a step length function, which determines how long that step plays before advancing to the next. This can be set manually by the performer or calculated dynamically using graph-based metrics:

- **Fixed to Quarter Notes**: All step lengths are set to quarter notes (4n).
- **Player-Defined**: Each step's length is chosen via a dropdown menu, allowing for precise rhythmic control using note values such as `"32n"`, `"8n"`, `"4n"`, etc.
- **Closeness Centrality**: Uses Cytoscape.js's implementation of Dijkstra's algorithm to calculate how central a step's graph node is within the DAG (see Listing C.7 in Appendix C). These values are normalized and mapped to a range of note durations between thirty-second notes and half notes. More central nodes may produce longer or shorter durations depending on the scaling strategy.
- **Euclidean Distance**: Takes two consecutive sequencer steps, computes the Euclidean distance between their associated `changeNodes`in the patch history graph, and maps that distance to a musical note length ranging from 32nd notes to quarter notes (see Listing C.8 in Appendix C). This allows step durations in the sequencer to reflect the spatial proximity of nodes in the DAG layout, creating a visually-informed temporal structure.

Empty Step Behavior

This mode determines how empty steps should be treated.

- **Pass-Through**: The sequencer does nothing for the full step duration. The result is that the state from the previous step remains active.
- **Blank Patch**: Loads the very first change node of the patch history — the blank patch. The result is that the synth is silent for the duration of the step.

- **Skip**: Skips ahead to the next step. If, for example, only three steps contained `changeNodes`, then the sequencer becomes a 3-step sequencer while this mode is active.

4.6 Modular Synth Architecture

The front end of the modular synthesizer in Forking Paths is built using Cytoscape.js (Franz et al. 2016), while its audio is rendered within a custom audio engine developed by the author, the full implementation of which can be found in Appendix F. The same underlying graph structure drives both the audio synthesis logic and the visual renderer, ensuring that what the player sees is consistent with the audio that they hear.

Modules are represented as parent nodes in the Cytoscape.js graph, with input and output jacks rendered as child nodes. Parameter controls—such as knobs and dropdown menus—appear in a floating UI layer above the Cytoscape.js viewport, spatially anchored to each module based on the same layout schema used for jack placement.

Earlier versions of the system used a custom patching interface built on an HTML canvas, but it proved too clunky and inflexible for ongoing development. React Flow (Möller and Klack 2019) was later adopted for its intuitive interface; however, limitations in its free version and restricted access to low-level API functions made it difficult to implement necessary custom behaviors. These constraints ultimately led to the adoption of Cytoscape.js, which offered greater flexibility and control over both layout and interaction.

4.6.1 Module Structure

The visual layout follows a consistent left-to-right ordering within each module: parameter knobs first, then input jacks, followed by outputs. This layout reinforces functional groupings—for example, in the SLOWFO module, the frequency knob, its CV input, and the attenuverter appear side-by-side (see Figure 4.20). Module structures are defined in a centralized JSON specification file (see Appendix G), which lists UI components, parameter ranges, CV mappings, and tooltips. These definitions are parsed by the `parentNode_webAudioNode.js` script, which generates both the visual Cytoscape.js elements and corresponding entries in the audio graph. A grid layout system, defined in `forkingPathsConfig.js` (see Appendix I), controls spacing and alignment. Descriptions embedded in the JSON appear in a floating readout when a player hovers over a node.

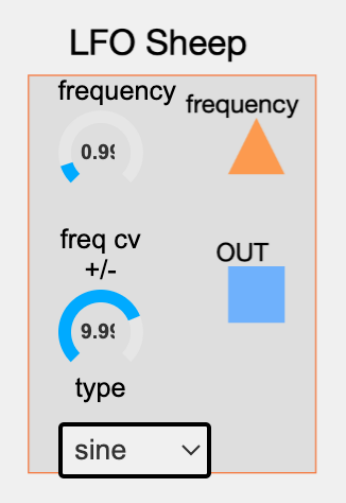


Figure 4.20: A Forking Paths Module

As multiple instances of the same module type can be included in the synthesizer, the system required a method for assigning unique identifiers to each module, both for structuring patch connections and mapping performer interactions to the audio DSP, as well as for tracking changes in the Patch History Window. To achieve this, the system combines a random animal name generator with a UUID hash. To avoid cluttering the user interface, only the animal names are displayed to the player.

Patching Interaction

Players create patch cables using click-and-drag gestures. When initiating a cable from a jack, a temporary "ghost node" appears and follows the cursor (see Figure 4.21). The ghost node visually indicates the required connection type—for instance, an output-shaped ghost node (square) will appear beside the player's mouse pointer if the cable was started at an input. As players hover over potential targets, only compatible jacks are highlighted via a custom Cytoscape.js class named `connectedEdges`. Invalid patch attempts are silently discarded, and the ghost cable subsequently disappears. These real-time visual cues—jack shapes, color coding, directional feedback, and hover-based validation—are meant to support accessible and accurate interaction for players.

Cable highlighting uses the Cytoscape.js `.connectedEdges` class, and mouse collision is detected using functions `getElementsAtPoint()` and isNearEndpoint(). A ghost cable is spawned using the `startCable()` function, see Listing B.9 in Appendix B. Finalized cables trigger a 'connect' event, which stores it in the `patchHistory` document.

Figure 4.21: An incomplete cable. Note that the ghost node at the unconnected end of this particular cable indicates that it can only be connected to an output (a blue square).

Parameter Control Overlays

The renderer for Cytoscape.js does not support native HTML input elements such as knobs or dropdown menus. To address this limitation, Forking Paths implements a custom overlay system. Parameter controls—such as knobs and menus—are rendered as floating HTML elements positioned above the Cytoscape.js viewport and anchored to their predefined positions within their parent node (module). These overlays track their associated nodes and automatically update their positions during zoom or pan events to maintain alignment with the visual graph.

Each floating element is associated with a custom Cytoscape.js child node, named `paramAnchorNode`. A `paramAnchorNode` is positioned within its module's parent node during the execution of `parentNode_WebAudioNode.js`. This placement provides a consistent anchor for each overlay, allowing the interface to determine precise positions for knobs or menus whenever a parameter is added to a module. This strategy decouples UI rendering and player interactions from the Cytoscape.js graph while preserving spatial consistency between interface elements and their corresponding synthesis behaviors.

Overlays are created using the createFloatingOverlay() function, which generates a containing either a knob or a <select> menu. Knobs are enhanced via the jQuery-Knob library (Terrien 2011), while menus are populated dynamically from the parameter specifications stored in `modules.json`. Cleanup is handled by `clearparamContainerDivs()`,

which is triggered whenever a `.fpsynth` file is loaded or a historical state is recalled—operations that cause the Cytoscape.js graph to be redrawn.

This approach emerged after earlier attempts to build parameter controls directly within the Cytoscape.js graph, using multiple nodes to simulate knobs. While somewhat functional, this method was plagued by inconsistent performance and poor interaction fidelity. The custom overlay system resolves these issues by separating the UI rendering layer from the graph itself, while maintaining tight integration through real-time positional synchronization. It ensures that parameter overlays remain responsive, visually coherent, and tightly coupled to both the user interface and the underlying audio engine.

Parameter Resolution

Since parameter ranges in Forking Paths vary significantly—from subtle modulation depths to wide frequency sweeps—the system dynamically adjusts knob resolution to ensure responsive and intuitive control. This is handled by the `determineStepSize()` function (see Listing 4.2) which computes an appropriate step size for each parameter based on its defined range in `modules.json`. Depending on the context, the function may apply linear or logarithmic scaling to maintain fine control across the entire parameter span, particularly for sensitive values like gain or filter cutoff.

```
1  function determineStepSize(min, max, method = 'linear', divisions =
       100) {
2      if (min >= max) {
3          console.error('Invalid range: min must be less than max');
4          return null;
5      }
6      // get the range
7      const range = max - min;
8
9      if (method === 'logarithmic') {
10         const factor = Math.log10(range + 1); // Avoid log10(0)
11         return (Math.pow(10, factor / divisions) - 1); // Base step
               size for log scaling
12     } else if (method === 'linear') {
13         return range / divisions;
14     } else {
15         console.error('Invalid method: Choose "logarithmic" or "
               linear"');
16         return null;
```

```
17        }
18  }
```

Listing 4.2: Computes a parameter step size based on a given range and scaling method. Supports both linear and logarithmic step calculations, with optional control over the number of divisions. Returns null and logs an error if the range or method is invalid.

Patch Legibility

To enhance patch legibility and support visual navigation in dense modular graphs, Forking Paths includes dynamic cable highlighting and curvature adjustment features. When a player selects a module, the `highlightEdges()` function applies the `.connectedEdges` class to all cables connected to that module, making it easier to trace signal flow and identify structural relationships. Additionally, players can manually adjust the curvature of patch cables using the `updateCableControlPointDistances()` function, accessible through the synth app's settings menu. This allows for fine-tuning of bezier control point distances, helping to reduce clutter, prevent cable overlap, and improve the overall legibility of complex patches.

4.7 System Communication Overview

Forking Paths runs in multiple browser windows and keeps those windows in sync via three concurrent communication layers:

1. **Inter-window (postMessage).** The Synth View (`synthApp.js`) exchanges room state, patch-history commands, and peer-pointer data with the Patch-History Window (`patchHistory.js`) via the browser's `postMessage` API.

2. **Peer-to-peer (WebRTC).** Synth Views running on different computers form direct data-channel connections for low-latency exchange of Automerge updates and gesture data.

3. **Client–server (WebSocket).** Each window maintains a WebSocket connection to a Node.js back-end that handles room management, WebRTC signalling, patch-history DAG rendering, and database queries.

Figure 4.22 presents a high-level diagram of these layers, the components they connect, and the primary data that flows along each path. Subsequent sections unpack the implementation details of the DSP engine (see Section 4.8), server architecture (see Section 4.9), the multiplayer synchronization pipeline (see Section 4.10), and the Synth Designer (see Section 4.11).

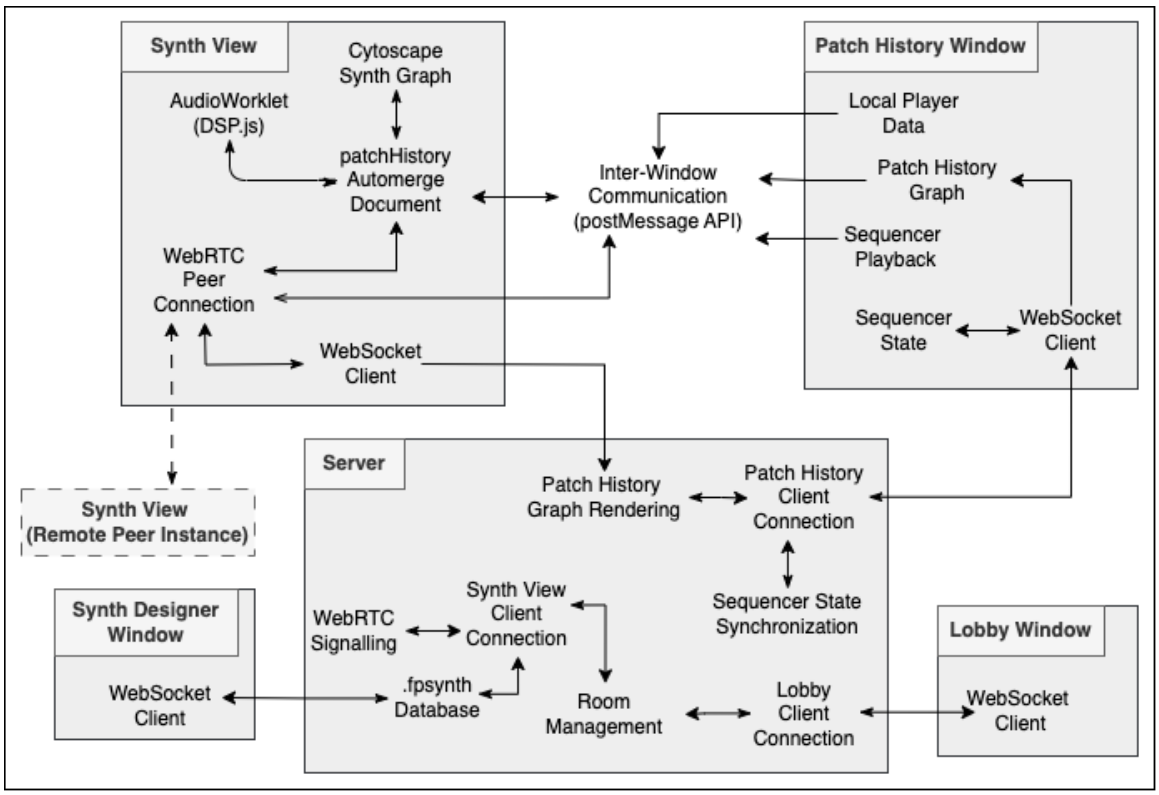


Figure 4.22: High-level communication architecture of Forking Paths from the perspective of a single peer. Solid arrows indicate local inter-window data flows via the postMessage API and client–server data flows over WebSockets. The single dashed arrow depicts the peer-to-peer WebRTC link to a remote peer's Synth View.

4.8 DSP Engine Implementation

The DSP engine in Forking Paths is implemented as a custom AudioWorkletProcessor named DSP.js (see Appendix F). This engine handles all real-time audio synthesis, modulation, and signal routing using a fully custom node graph architecture defined within the worklet. Nodes are instantiated via an internal method called `audioNodeBuilder()`, which builds one of several supported module types (e.g., Oscillator, VCA, Delay). The main app passes updates to the `AudioWorkletProcessor` through a function named `updateSynthWorklet` (see Listing B.6 in Appendix B).

4.8.1 Audio Graph Model

The synthesizer graph is structured as two separate but complete versions of the signal network: `currentState` and `nextState`. This dual-graph model allows seamless crossfading between patches when players load versions, whether through clicks on graph `changeNodes` or when using the Patch History Sequencer. Transitions between graphs begin when the main thread posts the audio graph from the `changeNodes` to the worklet, which rebuilds the `nextState` and starts a crossfade within the block-size boundary of 128 samples.

Each node in the DSP graph is represented as an object with several key fields: `baseParams`, `modulatedParams`, `output` (a typed audio buffer), and often module-specific fields such as phase or delayBuffer. Modulation offsets are applied per-sample using the helper function `getEffectiveParam()`, which sums a parameter's base value and modulation offset—see Listing 4.3. If the parameter has an attenuverter (e.g., "frequency cv +/-"), the modulation is scaled accordingly.

Modulation is handled per-sample using the `modulatedParams` field of each node. These values are added to the base parameter (scaled by an attenuverter if present) via `getEffectiveParam()` (see Listing 4.3). The result is used in real-time synthesis.

```
const getEffectiveParam = (node2, param, attenuverter) => {
      const base = node2.baseParams[param] || 0;
      let modulated = node2.modulatedParams[param] || 0;
      if (attenuverter) {
        modulated = modulated * attenuverter;
      }
      const result = base + modulated;
      if (typeof result !== "number" || isNaN(result)) {
        console.warn(`Invalid value for parameter ${param}:`, result
            );
        return 0;
      }
      return result;
};
```

Listing 4.3: Modulation offsets are applied per-sample using the helper function `getEffectiveParam()`, which adds a parameter's base value and modulation offset. If an attenuverter is present (e.g., "frequency cv +/-"), the modulation is scaled accordingly.

Signal routing is handled internally using arrays of connection descriptors: `signalConnections`, `cvConnections`, and `outputConnections`. These lists are populated when cables are added by the player. The signal graph is traversed recursively during audio processing, ensuring all upstream modulations are resolved before each node is evaluated.

Each processing frame, the `process()` method iterates over all nodes in `currentState`, evaluating them according to their type. Oscillators, filters, VCAs, and other modules are processed using per-sample logic. All modules are defined inline within `DSP.js`, not in separate classes, though helper methods such as `getEffectiveParam()` and clamping logic are used throughout.

When a patch version is loaded either by the player or the sequencer, `nextState` is populated and cross-faded in over the span of one block. This is achieved by fading between the output buffers of the corresponding nodes in `currentState` and `nextState`. No additional interpolation or smoothing is performed beyond this linear fade. Under the current implementation, `currentState` and `nextState` do not share memory buffers—each node's internal state, including delay lines, is deep-copied using cloneDeep from Lodash (*lodash* 2014), ensuring the two audio states remain fully isolated.

4.8.2 Feedback and Modulation

Feedback in modular synthesis is a powerful technique in hardware, but implementing it in software requires care to avoid zero-delay loops, which can cause infinite recursion. To address this, Forking Paths uses a cycle detection function, `isEdgeInCycle()` (see Listing B.11 in Appendix B), to determine whether a new connection would complete a feedback loop in the parent-node graph. This function is capable of detecting cycles involving multiple cables.

If a cycle is found, the system automatically inserts a `feedbackDelayNode` into the audio graph. This node introduces a delay of exactly one audio block (128 samples)—the minimum possible delay length in the Web Audio API—and is implemented using a circular buffer (`Float32Array(128)`). The node does not appear in the visual graph, and players are not notified directly. If the cycle is later broken (e.g., by removing a cable), the delay node is automatically removed.

Inside the DSP engine, feedback paths are separated from standard signal paths using `feedbackBuffers[id]` to avoid unsafe recursive dependencies.

4.8.3 Audio Context Controls

The main thread communicates with the worklet via the `updateSynthWorklet()` interface, which routes messages using a `handleMessage()` switch block. This is the sole entry point for operations, including:

- Adding or removing nodes or cables
- Modifying parameters
- Triggering version loads
- Enabling or disabling signal analysis
- Updating output volume

RMS monitoring is implemented directly within the worklet. When enabled, the system computes the RMS value of each block of the output buffer. To avoid performance issues, it throttles postMessage updates to once every 100 blocks. The result is sent back to the main thread and displayed in the `signalAnalysisDisplay` field. RMS values are linear (not dBFS) and not persisted between sessions due to CPU concerns.

Audio output can also be toggled on or off by suspending or resuming the Web Audio context. The interface updates its style dynamically: blinking red when suspended and reverting to gray when resumed.

4.8.4 Clamping and Parameter Validation

Most parameters are used directly without bounds-checking. However, specific parameters like gain or wet/dry ratio are clamped using helper functions to avoid instability or clipping. This is implemented within the DSP engine and applies selectively, not globally.

4.9 Server Architecture

To support real-time collaboration, dynamic session management, and scalable patch history rendering, Forking Paths includes a lightweight Node.js server (see Appendix D). The server handles four main tasks: rendering patch history graphs using Cytoscape.js in headless mode, facilitating WebRTC peer discovery, coordinating room-based sessions, and handling queries to the .fpsynth and patch history databases. This section focuses on the backend infrastructure, including signaling and server-side rendering. Real-time patch history synchronization, handled entirely peer-to-peer, is covered in Section 4.10.

4.9.1 WebSocket Implementation

The server is built from scratch using the ws WebSocket library (*ws* 2025). Hosted on Render.com, the server benefits from continuous deployment via GitHub integration and avoids cold starts or downtime.

4.9.2 Headless Graph Rendering

The patch history graph in Forking Paths can grow substantially during long or exploratory sessions, making client-side rendering computationally expensive. Initially, the system attempted to layout and render the entire DAG in-browser. However, as graphs became more complex, this caused lag across unrelated parts of the UI, including event handling and sequencer operation. After extensive optimization using Cytoscape.js's recommended perfor-

mance strategies[4], graph layout rendering was ultimately moved to a headless Cytoscape.js instance running on the server.

When the `patchHistory` document is updated, it is sent from the Patch History Window to the server via WebSocket (see Appendix D). Upon receiving the document, the server invokes the `buildHistoryGraph()` function (see Appendix E) which constructs the nodes and edges, computes their layout using Cytoscape.js in headless mode, and serializes the resulting graph structure as lightweight JSON. This structured response contains node positions and styling information and is then returned to the Patch History Window, where it is displayed in-browser. Offloading this layout computation enables the browser client to remain responsive and focus on player interaction and sequencing without incurring the overhead of computing large or complex DAGs.

4.9.3 Peer Signalling and Room Management

Collaboration in Forking Paths is organized into isolated sessions referred to as "rooms." These rooms are managed entirely in-memory using a lightweight custom system. When a client connects, it can request a specific room or be automatically assigned to the first available room with space. Room assignment follows a simple logic: if the requested room exists and has space, the player is added; if it is full, they must choose another; if it does not exist, it is created. If no room is specified, the server joins the client to the first room with availability or creates a new room with a default name. This system ensures ease of use while maintaining room-level isolation.

The Forking Paths Lobby—the landing page at https://forkingpaths.onrender.com—connects to the WebSocket server on load and requests the current list of active rooms and their participants. Joinable rooms are sorted to the top, and a new empty room is added if all existing rooms are full. This list is also updated using the `sendRooms()` function, which broadcasts room metadata whenever players join or leave a session, and avoiding constant polling while keeping session information accurate. Each room entry includes a "Join" button that opens the modular synth interface in `synthApp.html` in a new tab, preloaded with the selected room name.

[4]https://js.cytoscape.org/#performance/optimisations

Peer Initialization

A player's identity is determined by a name prompt upon the first time visiting the Forking Paths website, disambiguated with a seven-character hash to avoid collisions, and then stored in IndexedDB for subsequent page visits.

WebSocket Signaling Server

All peer discovery and connection setup occurs via a central signaling server using WebSocket. This server handles the exchange of SDP offers/answers and ICE candidates, assigns players to rooms, and relays information needed to initiate WebRTC connections. The WebSocket connection remains open throughout the session but is not used for fallback synchronization. All Automerge sync data is transmitted strictly peer-to-peer.

Peer disconnections and abrupt exits are handled gracefully by the server. When a player disconnects, they are removed from the internal session map, and if a room becomes empty, it is deleted. This approach keeps memory usage low and avoids orphaned sessions. The system currently includes no authentication or rate limiting, reflecting its status as a research tool intended for trusted, low-traffic usage.

4.9.4 Synth File Database

The server maintains a PostgreSQL database to store `.fpsynth` files created by players using the Synth Designer Window. Players can also query this database from within the Synth View to load their instruments or search by filename, tag, or author. All communication is handled over the existing WebSocket connections.

Handling Saves from the Synth Designer

When a player saves a synth, the Synth Designer serialises the instrument into a `.fpsynth` JSON object and emits a WebSocket message containing the file name, author name, tags, description, and the JSON payload. The server inserts this data into a table named `synth_templates`, using the following schema:

```
CREATE TABLE synth_templates (
  id            SERIAL PRIMARY KEY,
  name          TEXT   NOT NULL,
  author        TEXT,
  description   TEXT,
  tags          TEXT[],
  created_at    TIMESTAMPTZ DEFAULT now(),
  synth_json    JSONB  NOT NULL -- full .fpsynth payload
);
```

Handling File Loads from Synth View

When a player opens the Synth View, they may either load a `.fpsynth` file from their own filesystem, or search the Synth Database. In the latter case, the client sends a WebSocket message containing query terms (e.g., author, tag(s), filenames). The server responds with a list of matching synth templates, and upon selection, sends the full `synth_json` payload for instantiation as the first `changeNode` in a new patch history.

4.10 Multiplayer Synchronization

Forking Paths supports real-time collaboration through a lightweight peer-to-peer synchronization model built around WebRTC and Automerge. While a player's first introduction to Forking Paths is most likely going to be solo play, the system is designed with a synchronization layer that allows two players to share a patch history, the sequencer state, and observe each other's UI interactions in real time.

Peer Coordination

After initial signaling via the WebSocket server, Forking Paths establishes direct peer-to-peer communication using WebRTC data channels. These channels enable real-time synchronization of shared state and interface interactions between participants.

Upon joining a room, each client opens a peer connection from within the Synth View window. Two WebRTC data channels are then established:

- `syncChannel` — transmits Automerge binary synchronization messages and sequencer state updates.

- `peerPointerChannel` — broadcasts pointer positions for visual awareness, along with other UI interactions such as ghost cable creation, movement, and removal.

These channels are configured in reliable mode by default. Automerge's CRDT model guarantees that updates remain consistent regardless of message order, eliminating the need for custom queuing. Pointer and ghost cable messages are sent directly per event, with no throttling or debouncing.

4.10.1 Patch History Synchronization

Upon joining a room, the application follows a series of steps to determine which form of the patch history should be loaded or synchronized for the player—see Listing B.1 for the corresponding code.

When a player joins an empty room, the application creates an empty Automerge document. It then queries IndexedDB for a saved `patchHistory`; if one exists, it loads it, otherwise it starts a new history and loads the demo `.fpsynth` (see Section 4.3.2).

A subsequent player joining this same room also begins by initializating Automerge, but rather than reading a patch history from local storage it waits for an initial synchronization message from the remote peer to hydrate their history and patch state based on the shared patch history.

Immediately after hydration, each client extracts the `.fpsynth` synth file definition stored in the Automerge `patchHistory` document and instantiates the corresponding audio graph. This guarantees that everyone hears and sees an identical patch, independent of the order in which they joined.

No peer is granted elevated privileges; any participant may make a global change, such as making a patch or parameter change, recalling `changeNodes`, loading a different synth file from the Synth Browser, clearing the history, or creating a merge.

Real-time synchronization is achieved using Automerge's Sync Protocol, which transmits binary deltas over the WebRTC data channel. When a peer makes a change to the patch, the Automerge document `patchHistory` is updated, and a `sendSyncMessage()` function is called. This function uses `Automerge.generateSyncMessage()` to produce a binary Uint8Array, which is then sent over the `syncDataChannel`.

On the receiving side, `Automerge.receiveSyncMessage()` applies the incoming message to the local copy of `patchHistory`. If the message is valid and introduces new changes, the following actions are triggered:

- The patching system is updated by calling `updateSynthWorklet()` with the new audio graph structure.
- The Cytoscape.js visual patch is redrawn using `updateCytoscapeFromDocument()`.
- The Patch History Window is refreshed using `reDrawHistoryGraph()`.
- The local Automerge branch is cloned and replaced with the latest version.

This process does not use `Automerge.applyChanges()` directly. Instead, it reconstructs the current document state by calling `Automerge.view()` on the updated `patchHistory` document and then updates the application state based on that version. This allows Forking Paths to maintain temporal awareness of each change, ensuring both UI and DSP state are synchronized precisely to the most recent agreed-upon version.

Document synchronization is fully bidirectional. Once a peer receives and applies a change, it checks whether additional sync messages are required and responds accordingly. Each client therefore acts as both sender and receiver in the sync loop.

Although only a single shared `patchHistory` document is actively synced at runtime, this document internally contains multiple embedded Automerge documents representing different branches of the patch history. These branches are used to render the full DAG in the Patch History Window and are handled internally via Forking Paths' `patchHistory` document structure.

4.10.2 Multiplayer Synth View

Remote peer mouse pointers are rendered on the other peer's viewport as a small, coloured star with their username beside it. Pointer coordinates stream over a dedicated WebRTC data channel, so every remote action is mirrored locally in real time. When a peer begins routing a new cable, for example, a ghost cable instantly appears in both contexts, with its position continuously updated through the same channel until the connection is either completed or cancelled (see Figure 4.23 below, and Listing B.10 in Appendix B)

4.10.3 Multiplayer Patch History View

Like the Synth View, the patch history sequencer stays fully synchronized across all players in a multiplayer session: step changeNodes, step lengths, mode selections, tempo changes, transport start/stop commands, and clear-sequencer actions (see Figure 4.24).

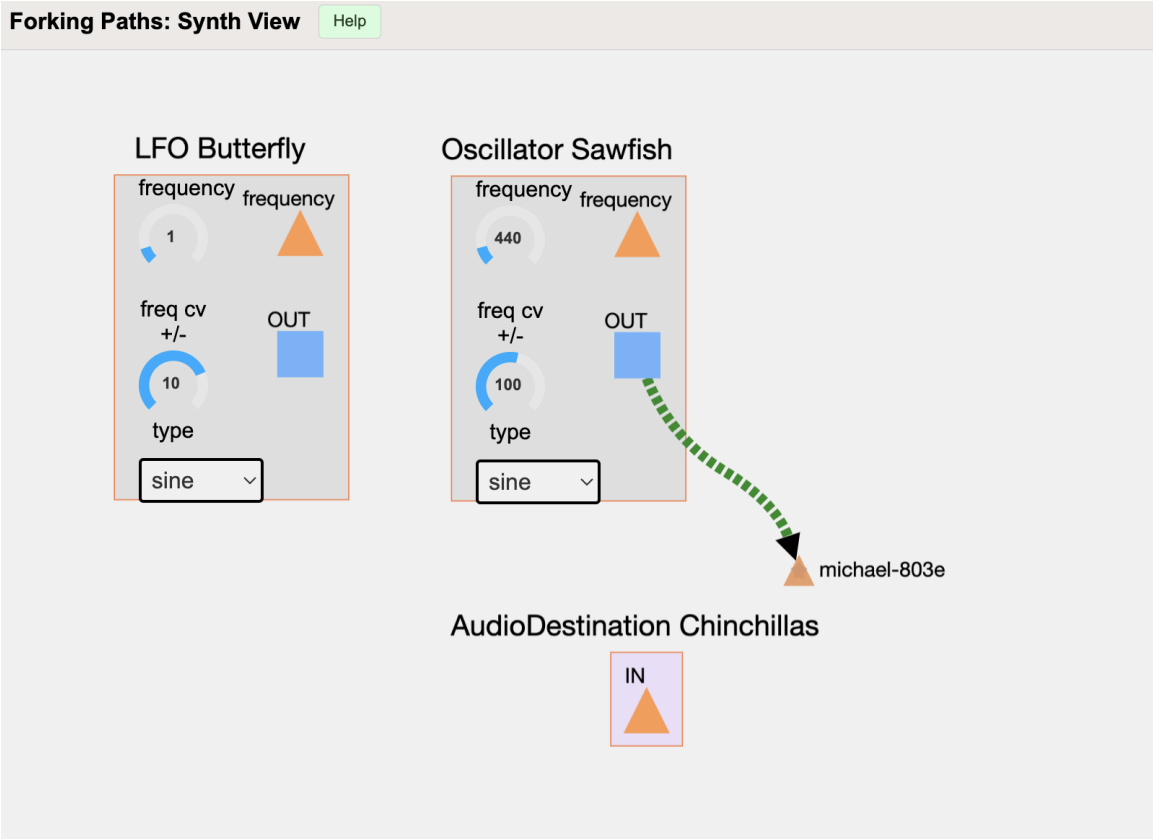


Figure 4.23: A remote peer's mouse pointer, and a cable that they are in the process of making, are rendered in the local player's Synth View.

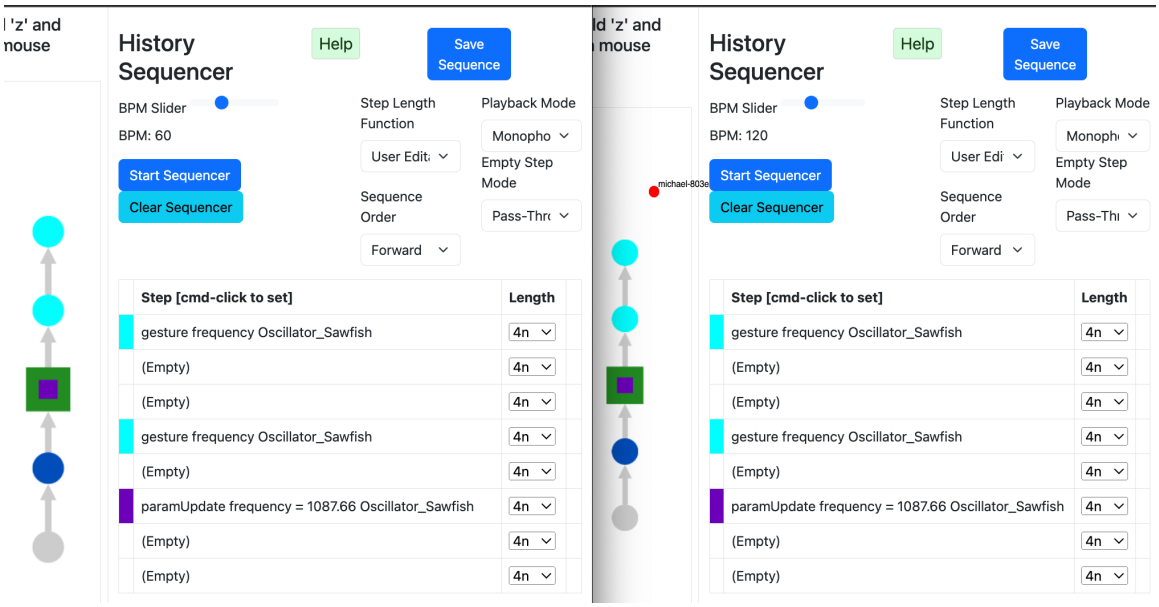


Figure 4.24: Real-time sequencer sync between two Forking Paths windows, with the left player's cursor shown as a red dot on the right.

The Patch History Window is not connected to WebRTC and does not receive sync messages directly. Instead, it listens for updates from the Synth View window, which are sent whenever the local `patchHistory` document changes—whether due to a player's own actions or in response to a sync message from a peer.

4.11 Synth Designer Window

The original vision for Forking Paths included the ability to add, remove, and reposition modules dynamically during a session, with each change captured in the version history. This would have allowed players to explore how the instrument design itself evolved over time, and to load varied changes into the Patch History Sequencer. An early implementation of this approach—complete with structural diffs and playback—exists on the archived branch titled `legacy/module-placement`[5].

Two key technical barriers made this model unworkable. First, Cytoscape.js, the graph library used for visualizing modular structures, struggled to accurately assign coordinates

[5]https://github.com/michaelpalumbo/forkingpaths/tree/legacy/module-placement

from loaded patch states during dynamic layout changes due to asynchronous rendering behavior and layout invalidation bugs.

Second, since Forking Paths relies on floating HTML elements, such as jQuery knobs and menus to represent parameters, these overlays were rendered separately from the Cytoscape.js canvas, requiring frequent updates to their position and scale. When switching between loaded states, these overlays either failed to detach cleanly, or persisted in memory, leading to rendering glitches, performance degradation, and memory leaks.

To maintain a stable and performant runtime environment, these features were removed from the main application and implemented within a separate design tool: the Synth Designer Window. While this separation limits real-time structural editing during performance, it improves reliability and reflects a pragmatic balance between flexibility and system coherence.

The Synth Designer Window includes a Cytoscape.js instance configured with the same layout preset as the Synth App so that saved synths can be loaded into the Synth App. A module browser panel dynamically loads available module types from a configuration file, as documented in Appendix G). Selecting a module from this list instantiates it at a default position on the canvas. Each module corresponds to both a visual representation in Cytoscape.js and a structure to be interpreted by the AudioWorklet.

For parameters, floating UI overlays are rendered using a function named `computePosition()` to ensure that knobs and menus are visually aligned with their associated nodes. When a player loads a synth file into the Synth View, the file instructs the app where to position these overlays.

Each synth is automatically assigned an AudioDestination module so that a valid audio output is always included.

4.11.1 File Format and Data Structures

Synths created in the designer are saved as `.fpsynth` files to the player's computer and also stored in the online synth database (see Appendix H). These are JSON-encoded definitions that include three top-level components:

- `visualGraph`: the Cytoscape.js graph including module layouts and parameter anchors, but no cables.
- `audioGraph`: a JSON object representing the DSP architecture, including module types and routing information.

- `paramUIOverlays`: a set of metadata entries defining the visual layout and configuration of knobs and menus.

When a `.fpsynth` file is loaded into the main Forking Paths application, it is parsed and used to initialize the DSP engine and generate parameter controls. Only `.fpsynth` files created within the Synth Designer can be loaded, eliminating the need for external validation or module presence checks. Although the format is technically extensible, creating new module types currently requires modifying both the DSP backend and the `.modules.json` schema. A user-friendly workflow for authoring new modules is not available.

4.11.2 Saving an Instrument

When a player clicks *Save* in the Synth Designer Window, the client performs two consecutive actions:

1. **Cloud save.** The `.fpsynth` JSON is sent to the Render-hosted Node.js server over the WebSocket connection. See Section 4.9.4 for how this is handled by the server.
2. **Local download.** After sending to the server, a dialog opens so the player can choose where to save the `.fpsynth` file on their own filesystem.

Players can search the Synth Database by author, filename, or tag without parsing the JSONB field.

4.11.3 Design Scope and Limitations

The Synth Designer Window intentionally omits several features present in the main application. It does not support:

- Cable patching or audio signal routing
- Real-time audio processing
- Version history or temporal navigation
- Undo/redo functionality
- Custom color schemes or interface theming

These omissions reflect the tool's minimal scope and its role as a setup utility rather than a creative performance environment. Parameter and jack elements are fixed in appearance and layout logic, optimized for clarity rather than customization.

4.12 Conclusion

In contrast to conventional modular synthesizer systems, where changes to a patch are ephemeral and difficult to reproduce, Forking Paths introduces a version-controlled framework in which every transformation of a patch—whether subtle parameter adjustments or large structural revisions—is recorded, recallable, and recombinable. This system uses VCS concepts as a performance tool, allowing musicians to explore the history of patch changes for further musical material.

Chapter 5

Evaluation and Discussion

Forking Paths was evaluated through a combination of public events and asynchronous online playtests. The public presentations ranged from general-audience salons to specialist meetups—which helped refine both its technical features and the metaphors used to communicate its VCS. These sessions provided valuable feedback on usability, creative potential, and the effectiveness of framing the software as a nonlinear alternative to traditional undo/redo workflows.

Seven participants took part in the online playtests at their own pace. While each session lasted at least one hour, the format was intentionally open-ended; several participants noted that they spread their engagement across multiple sessions. Feedback was collected via a Google Form[1] available throughout the testing period, as well as an optional submission to the issues threads of the Forking Paths GitHub repository[2]. In addition to reflections on their experience with Forking Paths, participants also provided information about their musical background, including the genres they work in, the hardware and software they use, and any other instruments they play. Participants in the Forking Paths evaluation were invited to complete a detailed online questionnaire designed to gather both experiential feedback and contextual information about their musical background. The form included a mix of demographic, technical, and reflective questions, grouped into several thematic areas:

- Participant information (name, pronouns, email, artist pseudonym)
- Musical background (genres, instruments, hardware/software setups)

[1] Playtesting feedback form is available is at https://forms.gle/ojyutCQW76bYrxwWA

[2] Forking Paths Github issues thread
https://github.com/michaelpalumbo/forkingpaths/issues

- Collaborative practices (ensemble roles, signal-sharing)
- System comparison (Forking Paths vs. hardware/software modular synths)
- User experience (playability, challenges, solo vs. multiplayer use)
- Version control and history sequencing (recall, branching, visual clarity)
- Suggestions and impressions (bugs, features, creative impact)

5.1 Public Demonstrations and Evolving Discourse

Over the course of development, *Forking Paths* was presented in a series of public demonstrations that helped shape both its technical design and the language used to describe it. These presentations served as opportunities to test features, gather informal feedback, and refine how version control concepts could be communicated to artist communities with varying levels of technical familiarity.

Presentation at Cool New Instruments Night 3

The first demonstration[3] took place on January 11, 2024, at *Cool New Instruments Night 3*, a recurring salon that I curate at The Tranzac in Toronto. The purpose of this event is to bring together instrument designers and artists who build tools for their own music-making, while also drawing in an audience with a wide range of familiarity with the topic. A significant portion of attendees are not musicians, nor are they necessarily versed in instrument design or audio technology. This diversity has made the event an important testing ground for how to communicate complex ideas—like VCS—to a general audience. At this early stage, Forking Paths included the foundational patching system, the history graph interface, and early prototypes of the history sequencer. I demonstrated basic patch creation, state recall via the version history graph, and the ability to sequence through past patches. The experience of presenting to a mixed-knowledge audience helped sharpen my awareness of the language and metaphors needed to make the system's core concepts more broadly accessible.

[3]CNIN3 version documented at https://github.com/michaelpalumbo/forkingpaths/tree/demos/CNIN3

Presentation at Cool New Instruments Night 4

By the time of Cool New Instruments Night 4 on April 12, 2024, both the software and my framing of it had evolved[4]. Based on conversations at the previous event, I began to articulate the project using more accessible metaphors—particularly the idea of Forking Paths as a nonlinear alternative to conventional undo/redo systems. I explained that most software treats undo/redo as a linear history, where returning to an earlier point and making changes erases subsequent edits. In contrast, Forking Paths preserves the full graph of player decisions, allowing alternative futures to coexist. This conceptual framing, supported by the visual directed acyclic graph (DAG) interface, helped situate the tool within familiar paradigms while foregrounding its experimental structure. During this session, I also demonstrated several new features: the sequencer's monophonic and polyphonic modes, the ability to set step length using Euclidean distance, the gesture editor, and the addition of quantized gesture playback within sequenced steps.

Presentation at Frequency Freaks April 2025 Workshop

A third presentation occurred at the Frequency Freaks meetup on April 19, 2024 at Arraymusic in Toronto—a gathering specifically oriented toward modular synthesizer enthusiasts[5] (see Figure 5.1). This session expanded upon the material from CNIN 4 but was tailored to an audience already deeply familiar with modular synthesis workflows. Although these demonstrations focused on solo use, discussion often turned to how Forking Paths could support networked improvisation. Audience members raised questions about multi-user patching and peer-to-peer sync, helping frame collaboration as a crucial dimension of the project. The shared technical vocabulary allowed for more focused discussion of how Forking Paths diverges from hardware and software patching conventions, particularly in its treatment of historical states, gesture layering, and sequenced transformations.

In addition to core features, the demonstration presented several newer tools and capabilities:

- A patch annotation system, allowing players to draw directly onto the patch

[4]CNIN4 version documented at https://github.com/michaelpalumbo/forkingpaths/tree/demos/CNIN4

[5]The version presented at Frequency Freaks is preserved at https://github.com/michaelpalumbo/forkingpaths/tree/demos/FFApril2025

- A synth browser for storing, navigating, and selecting instruments from a new cloud-based database
- Import/export functionality for patch history files
- Advanced sequencing options, including:
  - Custom sequence orders
  - Configurable empty step modes, allowing players to choose whether the sequencer should:
    * Pause with no action
    * Recall a silent default state
    * Skip the step entirely

    These empty step modes have since been implemented and are documented in Section 4.5.3

The feedback and curiosity from this group further affirmed the project's relevance within modular synthesis communities, with several attendees later participating in follow-up playtesting. Their insights helped highlight additional directions for improving usability and real-time control.

5.2 Feedback and Observations from Participating Musicians

Participants reported a wide range of musical backgrounds, spanning ambient, techno, drone, electronic, and experimental styles. Several identified with classical, jazz, and pop traditions, while others noted involvement in hip-hop, noise, free improvisation, and alternative RnB. Many responses reflected hybrid or cross-genre practices, suggesting that their creative work resists simple categorization. Responses here were especially important for contextualizing Forking Paths' multiplayer potential. Several participants already performed in ensembles that relied on shared signals (CV, clock, audio).

Of the ten participants in the study, five reported playing modular synthesizers alone, three indicated they perform both solo and with others, often alongside acoustic instruments or other electronic hardware, such as keyboards and other modular systems. Among those who collaborate with other electronic musicians, some reported sharing signals like control

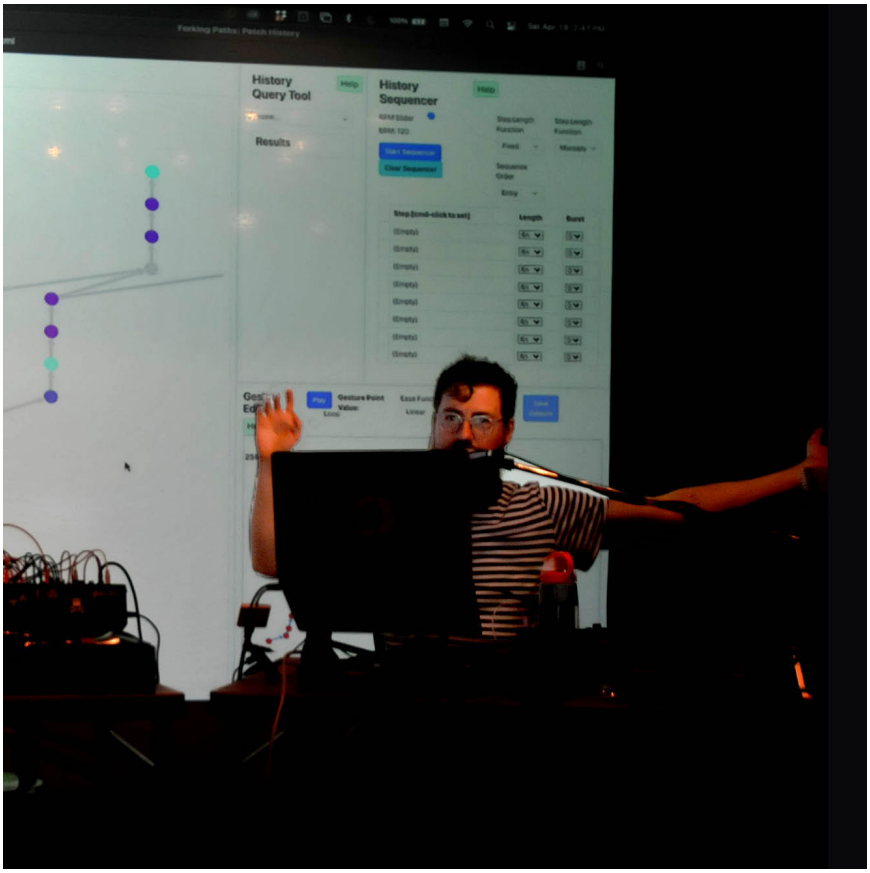


Figure 5.1: Presentation of Forking Paths at the Frequency Freaks monthly meetup on April 12, 2025 at Arraymusic in Toronto (Photo by Stephen Humphrey).

voltage (CV), clock, or audio for processing, while others noted no signal sharing or marked the question as not applicable. The remaining two participants, while experienced in electronic music, stated that Forking Paths was their first introduction to modular synthesis.

5.2.1 Comparisons to Hardware Modular Synthesis

Participant experiences with hardware modular synthesizers varied widely. Four respondents reported owning Eurorack systems, ranging from compact setups—such as a 104hp skiff — to larger configurations described simply as "a few rows." Four others indicated they did not own a hardware modular system, citing reasons such as cost or a preference for using digital audio workstations (DAWs). One respondent described their setup as a sequencer-based, portable, tactile DAW combining digital and analog elements, with influences from both East Coast and West Coast synthesis traditions.

Participants with prior experience using hardware modular synthesizers noted that Forking Paths evoked a strong sense of familiarity. Three participants highlighted the open-ended patching system—particularly the ability to freely route signals—as closely resembling the tactile and exploratory nature of Eurorack systems. Players described experiences of modulation, delay, and FM that yielded sonically familiar results, and some praised the interface's clear documentation, wishing similar clarity were available in analog systems. Others emphasized the intuitive layout of tools and modules, and the ease of connecting virtual components.

These participants identified several ways in which Forking Paths diverged from the experience of using a hardware modular synthesizer. The inclusion of features like patch history, state recall, and a sequencer capable of radically transforming the patch were noted as unique to the digital environment, offering forms of flexibility and memory that are not typically possible in hardware systems. One participant noted an unintended feature of parameter overlays was that the knobs always sit above the cables, meaning that they are always controllable (compared with hardware and software paradigm of cables being in front, having to comb through denser patches to get to the module interfaces). One participant mentioned that the multi-window interface made it harder to understand how different components related to each other, highlighting a contrast with the spatial immediacy of physical systems.

5.2.2 Comparisons to Software Modular Synthesizers

The group responded with a mix of novice and casual software-based modular synth usage, with only a minority indicating sustained engagement with modular synthesis in a digital format. Three participants had limited or outdated experience, mentioning only brief encounters with tools like VCV Rack or older soft synths. One participant noted spending just 15 minutes with VCV Rack several years ago, while others recalled using software like Arturia Modular V, Pure Data (including the Automatonism library), and VCV Rack with more intent. Four respondents stated they had little to no experience with software modular environments.

For the five participants with some experience using software modular synthesizers, Forking Paths evoked familiar interactions such as clicking and dragging patch cables or navigating a flexible, canvas-based interface. One respondent appreciated the drag-and-drop sandbox approach, contrasting it with more rigid or visually dense software environments. Two participants mentioned standard digital conveniences like file management and the ability to save and recall states—features commonly associated with software-based systems.

Participants highlighted several ways in which Forking Paths diverged from their expectations of a typical software modular synthesizer. A notable distinction was the ability to move "backwards" through patch states—an uncommon feature in most software environments, which typically focus on linear or manual state loading rather than versioned navigation. Others remarked on the simplicity and clarity of the interface, noting the absence of cluttered or overly stylized graphics that often characterize other soft synths. At the same time, one participant pointed out the challenge of not having all elements visible on a single screen, which differed from more consolidated software layouts. Others noted the limitation of being unable to dynamically add or remove modules once a patch was underway.

5.2.3 Reflections on Usability and Creative Potential

Playing Forking Paths solo was generally described as fun, exploratory, and immersive. Playtesters emphasized the usefulness of navigable patch histories, editable gestures, and shared control over musical time as meaningful and inspiring compositional tools. Participants described Forking Paths as a fun and creatively engaging tool, emphasizing its openness, visual clarity, and exploratory potential. Several highlighted the sequencer as a key source of enjoyment, citing both its functionality and the freedom it provided to structure evolving soundscapes. The ability to visualize parameters like tempo and frequency

made the interface approachable, while branching from different patch states or remapping gestures offered satisfying opportunities for experimentation. One participant appreciated the system's balance of creative possibility and built-in guidance, noting that the range of options never felt overwhelming. Another, approaching from a beginner's perspective, found the visual layout of cables and nodes especially enjoyable.

Participants saw the Forking Paths History Sequencer as a tool with significant creative potential, both in studio environments and on stage. Many highlighted its ability to save and recall patch states as a powerful feature that could encourage risk-taking and exploration without fear of losing progress. In performance contexts, some imagined using MIDI controllers to navigate the patch graph dynamically, envisioning a tactile, real-time interaction with past states as a compelling form of live control, though some noted that navigating the interface via a web browser might be less practical in those settings. In the studio, participants imagined using the sequencer to track and revisit sonic ideas over long sessions or even across months, creating a kind of evolving archive of their musical development. Others suggested it could streamline sound design workflows or facilitate returning to earlier patterns while crafting variations. One participant reflected on the tension between capturing fleeting modular states and maintaining their "living" quality, acknowledging the unique possibilities and trade-offs that such a system introduces. Overall, the History Sequencer was seen as a potentially transformative addition to creative practice, especially with sustained use.

Participants described a range of learning processes when engaging with the Forking Paths History Sequencer. Several found it intuitive or easy to pick up, especially when supported by clear documentation or mouse-hover instructions. Others required more time to understand its functionality, with some initially unsure how to program it or navigate across windows. The Help menu and query tool were repeatedly mentioned as useful entry points, helping participants grasp how to recall past states and integrate them into a sequence. One participant approached learning by interacting first with familiar modules—like the gesture system—before exploring the sequencer itself. Many responses emphasized a learning curve, particularly around understanding how to use the sequencer and achieve musical outcomes—but also expressed that once the system "clicked," it became highly engaging and even addictive. While most participants reported eventual success, at least one noted difficulty producing sound at all, limiting their ability to assess the potential of the sequencer. This player suggested adding a simplified "Hello World" demo synth with a demo patch history, to allow players to immediately verify sound production and observe the patch

history right away. Three participants recommended that an introductory demonstration video be included in the app to assist in rapidly explaining the functionality of the history window (private correspondence).

Multiplayer participants were drawn to Forking Paths precisely for its multiplayer possibilities and agreed that, once mastered, it could become a genuinely engaging performative instrument. At the same time, one tester admitted that a steeper learning curve still stands in the way of fully unlocking that potential, hinting at the need for clearer on-boarding resources or shared tutorials. Two playtesters emphasized that once they connected over a separate real-time voice chat, that made coordinating their actions smoother, turning the session into something that "felt more like a game." One participant who had previously tried Forking Paths solo noted that playing with a partner was far more enjoyable, whereas solo play had felt almost overly serious. These experiences suggest that Forking Paths' collaborative potential extends to the social experience of making music together, with participants consistently noting that shared play felt more engaging than solo use.

Overall, participants described Forking Paths as a playful, inventive, and at times challenging tool. The ability to edit gestures, switch between patch states, and revisit previous configurations was highlighted as a standout feature that set the system apart from traditional synthesizer workflows. Some likened the experience to using a "meta modular synthesizer," with access to multiple timelines of an instrument's state. For those familiar with similar research projects, Forking Paths evoked memories of experimental gesture-based systems, reinforcing its connection to broader creative and academic contexts. While some participants initially felt overwhelmed or uncertain—especially those without a modular background—most ultimately found the experience rewarding, with a few describing it as "super cool" or "really, really fun." One participant remarked that the system already "feels great as a musical instrument," emphasizing its capacity to generate "a lot of strange possibilities," which they associated with the most compelling qualities of musical instrument design. They appreciated the opportunity to interact dynamically with patch histories, using the interface as a compositional and improvisational resource. Several participants specifically highlighted the history sequencer and gesture version control features as particularly exciting, noting that they encouraged exploration of alternate musical possibilities without fear of irreversible mistakes. One participant remarked that seeing a timeplot of previous gestures helped recall unconscious creative choices, describing this reflective quality as "fascinating" and worthy of deeper exploration. Another suggested imagining the Patch

History Sequencer "like an old-school looper," where specific points in the history could function as distinct sections of a song.

Participants identified a range of enjoyable and frustrating aspects of their experience with Forking Paths. The sequencer emerged as a consistent source of enjoyment, especially its ability to radically alter patches from step to step, support subtle or drastic changes, and interact with looping gestures in musically complex ways. Several players appreciated the creative potential of combining sequenced and non-sequenced gestures, noting how this interaction led to rich, organic results. The inclusion of help tools was also noted positively, though some felt the guidance assumed a degree of prior familiarity with modular synthesis terminology. Frustrations were often tied to interface limitations or technical inconsistencies. Players expressed a desire for more direct interaction with the history graph, including clearer mechanisms for branching from specific nodes and filtering out minor parameter changes to reduce visual clutter. Others encountered sequencing glitches, such as states not recalling correctly when patch changes occurred mid-performance. Practical UI concerns also surfaced, including overlapping menus, mouse input issues, and challenges creating connections. Some participants requested expanded sequencing capabilities—like slower tempos or running concurrent parallel sequences—as well as organizational features for managing patch histories. While much of the evaluation centered on solo play, the multiplayer sessions underscored that Forking Paths' most distinctive contributions may emerge in collaborative contexts, where branching, forking, and history merging become shared resources for improvisation.

5.3 Future Directions

5.3.1 Forking Paths as a Standalone App

In Forking Paths, developing a custom software modular synthesizer was a pragmatic choice. It allowed full authorial control over the synth API and feature set, which greatly simplified the integration of versioning logic and the tracking of parameter changes. However, a long-term goal is to decouple the versioning system from a proprietary synth engine. Future iterations of Forking Paths aim to function as a device and suite of plugins that can integrate into a performer's existing ecosystem of hardware or software instruments. This could take the form of a MIDI device, a Max for Live plugin with scripting access in host environments such as Ableton Live, enabling musicians to incorporate versioning and patch histories into their creative workflow without abandoning their familiar tools.

5.3.2 Leveraging Other WebRTC Features

While the current peer-to-peer implementation of WebRTC (W3C WebRTC Working Group 2021) focuses on control and state synchronization, the WebRTC protocol also supports media streaming, making it feasible to transmit audio for voice chat or even networked instrumental performance. These features have not yet been implemented, but they represent a promising avenue for deepening the sense of co-presence and collaboration in distributed improvisational contexts.

5.3.3 Patch History Window Improvements

During playtesting, several participants proposed improvements that point toward valuable future directions for the development of Forking Paths. One common request was for greater control over sequencer behavior—particularly the ability to adjust the number of steps and run multiple sequencing processes in parallel. Others suggested consolidating the system's multi-window interface into a single-page experience, aiming to streamline navigation and improve usability during both live performance and studio use.

The history graph query tool will expand to include filtering features based on author and timestamps. Author-based filtering would allow players to isolate changes made by specific collaborators, while timestamp filtering would support selecting changes made within a particular time range.

Each room supports a maximum of two peers—a constraint that reflects the initial implementation but is flagged for future expansion.

Gesture editing in Forking Paths is limited to vertical adjustments, allowing players to modify parameter values but not their timing. Horizontal movement for time editing, as well as multi-point selection and editing, are not yet supported. Additionally, the system does not yet support multichannel or multi-parameter gestures—features that will be essential for capturing simultaneous knob movements across multiple parameters, particularly in collaborative settings. Implementing these capabilities will be critical for enabling more nuanced and expressive forms of real-time control, especially when multiple players are interacting with the system in parallel.

At the time of writing, it is possible to navigate the patch history graph using MIDI CC messages, where CC values (0–127) are mapped to the chronological order of the graph. For future development, this functionality could be expanded to involve external sequencers and hardware controllers, enabling more expressive, real-time control over patch

transitions. Such integration would allow performers to trigger version changes rhythmically or algorithmically, opening new creative possibilities for structured improvisation and live composition.

5.3.4 Introducing Lossy Versioning

Another future direction involves challenging the assumption that perfect recall is inherently beneficial to creativity. As one dissertation committee member observed during the review process, Forking Paths' ability to retain every change, branch, and abandoned idea introduces a kind of totalizing memory that may echo patterns of digital surveillance. As Ricoeur (2004) suggests, forgetting is a necessary counterpart to remembering, and is integral to how meaning is constructed over time and how experiences are prioritized. In contrast to the fluidity and partiality of human memory, Forking Paths offers an unblinking archive. This raises important questions about agency: who—or what—decides what is worth remembering, and what might be gained by deliberately allowing some creative actions to disappear?

In this light, it becomes worth exploring whether forgetting could be designed into the system as a feature. For example, a player could mark branches as "ephemeral", triggering automated decay over time unless they are actively revisited. Alternatively, players could select a "forgetting" mode in which older changes are gradually erased, compressed, or visually blurred. These gestures of loss could serve not only to reduce visual clutter, but also to acknowledge that creative work often involves moving on, letting go, or allowing unfinished gestures to recede into irretrievability.

This could also invite exploration of what happens when imperfection or decay is deliberately introduced into a VCS. For instance, what if a VCS were designed to accumulate "noise", or allow randomness to alter its record of changes? Randomized commits, degraded deltas, or decaying branches would disrupt the expectation of perfect recall. A decaying or generative version history might position a VCS within a system oriented to machine improvisation.

At a broader level, incorporating metaphors of forgetting or loss could have implications for community-based uses of Forking Paths. In collaborative improvisation, the ability to selectively forget may serve not only individual expression, but collective dynamics as well, preventing any one contributor's ideas from dominating simply by virtue of their permanence. This invites future research into socially-negotiated forms of memory and forgetting: how might a group collectively decide what to preserve, and what to relinquish?

5.3.5 Server Improvements

The server is single-threaded and handles graph rendering synchronously while processing one DAG layout at a time in the main Node.js thread. Future iterations may explore offloading this work to child processes or adopting multithreaded rendering for improved scalability.

Forking Paths currently operates on a trust-based model. Data channels are not encrypted or authenticated beyond the protections offered by WebRTC itself. No message signing, player authentication, or document validation is performed. To date, no desynchronization bugs have been observed, though future versions may implement mechanisms for detecting divergence or verifying state consistency.

5.4 Comparative Systems and Design Paradigms

A project with some conceptual overlap emerged during the development of Forking Paths: *SHARP* (Manesh, Bowman Jr, and Lee 2024), a VCS designed specifically for live coding with TidalCycles (McLean 2014). SHARP automatically generates version trees from code execution events, enabling performers to navigate past versions, branch new ideas from earlier states, and revisit prior structures during performance. Embedded directly into the coding interface, SHARP offers lightweight, pattern-level versioning that supports both improvisation and structured composition, making musical form more retrievable and manipulable in real time.

While SHARP and Forking Paths both treat version control as an expressive layer for temporal navigation, they are grounded in different paradigms. SHARP operates within a textual, code-driven environment, whereas Forking Paths emerges from a spatial, modular synthesis interface. Forking Paths diverges from SHARP's per-pattern linearity by enabling patch histories to be sequenced and replayed across multiple branches, allowing performers to sculpt time at the level of synthesis state rather than symbolic pattern. It supports the nesting of patch histories as compositional units, enabling histories to function as instruments in themselves.

Chapter 6

Conclusion

This dissertation set out to close a documented gap in multiplayer modular-synthesis software: the absence of tooling that both supports real-time, many-hands collaboration and treats incremental patch transformations as first-class, navigable data. Forking Paths demonstrates that integrating real-time version control techniques directly into the interface of a virtual modular synthesizer can satisfy these requirements. In doing so, the system prototypes what are described as a fourth-generation version control system: a social environment where atomic change logging, branching, and merging happen during performance rather than after the fact.

Chapter 2 laid the conceptual foundation for Forking Paths. It used theories of distributed creativity, software studies, and temporal authorship to argue that contemporary music-making needs version-aware, multiplayer instruments. By surveying the first three generations of version control systems, the chapter proposed the idea of a fourth generation to encapsulate social platforms like GitHub and real-time protocols such as Operational Transform (OT) (Sun and Ellis 1998) and Conflict Free Replicated Datatype (CRDT) (Kleppmann and Beresford 2017) that allow creators to co-edit shared material live.

Chapter 3 charted a research-creation path through a sequence of related projects. It began with exploratory studies that repurposed Git as a creative aid for scholarly writing and musical composition, demonstrating how conventional version control can foster reflective, process-oriented practice. Next, it introduced two projectional editors: one that leveraged Git worktrees to hot-load updates into a live simulation, and another—a real-time editor for abstract-syntax trees—built on a bespoke operational-transform system for graph structures. The chapter closed with two virtual-reality modular-synthesizer prototypes that used OT and CRDT frameworks, respectively, to support live multiplayer patching.

Chapter 4 documented the Forking Paths interface, design rationale, and full technical stack, illustrating that real-time version control can be embedded directly in a modular-synthesis environment. Forking Paths delivers three core technical contributions: first, it uses a CRDT via Automerge (Automerge 2017) to capture every patch change – cable connections and disconnections, discrete and gestural parametric changes, patch notes – allowing performers to scroll, branch, and replay sonic timelines with blocksize-accurate fidelity; second, it supports real-time, peer-to-peer collaboration over WebRTC that permits simultaneous edits to a shared state; and third, it introduces a History Sequencer and Gesture Editor that repurpose version data as compositional material, enabling musicians to create new musical structures from patch states from anywhere across the patch history.

Chapter 5 documented Forking Paths in action through three public-salon performances and a series of targeted play-tests; together, these events confirmed the system's creative potential, exposed its current limitations, and pointed toward the most promising directions for future development and research. Play-testing surfaced three recurring themes. First, participants found the system "familiar yet novel": while routing signals, FM, and delay felt immediately intuitive, the ability to rewind or branch a patch in performance was described as both transformative and addictive. Second, risk-free navigation of patch history changed creative behaviour, encouraging bolder improvisation and risk-taking without fear of losing track of ideas. Third, the retention of changes within a branching structure, and time-plots of gestures, offered reflective insight, helping musicians recall unconscious decisions and pointing toward new avenues for analysis and pedagogy.

Forking Paths still bears the practical limits of a tightly scoped proof-of-concept. Because the software is welded to its custom synth engine, it cannot yet slot into a performer's existing hardware or DAW workflow. Collaboration is capped at two players per room, the interface is split across several windows to simplify threading, and a single-threaded Node.js server renders history graphs synchronously—together these factors constrain scalability and usability. The next development cycle will relax those constraints. By separating the version control core from the in-house synth engine, Forking Paths can be reborn as a MIDI device, VST, or Max-for-Live plug-in that drops naturally into established musical instrument ecosystems. A consolidated, single-window interface built on worker threads will simplify navigation, while removing the two-peer limit and off-loading graph rendering to a multi-threaded server will enable ensemble-scale sessions in which larger groups improvise within one shared history.

Patch histories have long been a lossy residue of electronic music practice, beholden to fragile external workflows—scribbled notes, partial screenshots, saved files, half-remembered knob positions. Embedding a fourth generation VCS into Forking Paths turns that residue into an audible, navigable medium, and opens new terrain for distributed creativity and multiplayer improvisation, where every performance is both present sound and a branching archive of potential futures.

Bibliography

Aarseth, Espen (1995). *Cybertext: perspectives on ergodic literature*. University of Bergen.

Ableton (2024). *maxdiff: A tool for readable diffs of Max patches*. URL: https://github.com/Ableton/maxdevtools/tree/main/maxdiff.

Ahmed, Sara (2012). "On being included: Racism and diversity in institutional life". In: *On being included*. Duke University Press.

Allegretta, Chris and Benno Schulenberg (1999). *nano*. URL: https://www.nano-editor.org/ (visited on 06/18/2025).

Andersson, Nikolaj, Cumhur Erkut, and Stefania Serafin (Mar. 2019). "Immersive Audio Programming in a Virtual Reality Sandbox". In: *Audio Engineering Society Conference: 2019 AES International Conference on Immersive and Interactive Audio*. URL: http://www.aes.org/e-lib/browse.cfm?elib=20443.

Armitage, Joanne and Helen Thornham (Mar. 2021). "Don't Touch My MIDI Cables: Gender, Technology and Sound in Live Coding". en. In: *Feminist Review* 127.1, pp. 90–106. ISSN: 0141-7789, 1466-4380. DOI: 10.1177/0141778920973221. URL: https://journals.sagepub.com/doi/10.1177/0141778920973221 (visited on 06/04/2025).

Austin, Kevin (2016). "A Generalized Introduction to Modular Analogue Synthesis Concepts". In: *eContact!* 17.4. URL: https://econtact.ca/17_4/austin_synthesis.html (visited on 06/04/2025).

Automerge (2017). *Automerge*. URL: https://automerge.org.

Barbosa, lvaro (2003). "Displaced Soundscapes: A Survey of Network Systems for Music and Sonic Art Creation". In: *Leonardo Music Journal* 13. Publisher: The MIT Press, pp. 53–59. ISSN: 09611215, 15314812. URL: http://www.jstor.org.ezproxy.library.yorku.ca/stable/1513450 (visited on 08/17/2025).

Battersby, Christine (1989). *Gender and genius: Towards a feminist aesthetics*. Women's Press Limited.

Belt, Andrew (Sept. 2017). *VCV Rack*. URL: https://vcvrack.com.

Bird, Christian et al. (2008). "Latent Social Structure in Open Source Projects". In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '08/FSE-16. event-place: Atlanta, Georgia. New York, NY, USA: ACM, pp. 24–35. ISBN: 978-1-59593-995-1. DOI: 10.1145/1453101.1453107. URL: http://doi.acm.org.ezproxy.library.yorku.ca/10.1145/1453101.1453107.

BitMover Inc. (2000). *BitKeeper Distributed Version Control*.

Booth, Sean, Rob Brown, and Joe Muggs (June 2015). *Autechre: elseq et al*. URL: https://www.residentadvisor.net/features/2756 (visited on 04/01/2019).

Borges, Jorge Luis (1964). "The Garden Of Forking Paths". In: *Labyrinths*. New York: New Directions, p. 260.

Born, Georgina (Mar. 2005). "On Musical Mediation: Ontology, Technology and Creativity". en. In: *twentieth-century music* 2.01, pp. 7–36. ISSN: 1478-5722, 1478-5730. DOI: 10.1017/S147857220500023X. URL: http://www.journals.cambridge.org/abstract_S147857220500023X (visited on 02/13/2019).

Boutard, Guillaume and Catherine Guastavino (Dec. 2012). "Following Gesture Following: Grounding the Documentation of a Multi-Agent Music Creation Process". In: *Computer Music Journal* 36.4, pp. 59–80. ISSN: 0148-9267. DOI: 10.1162/COMJ_a_00147. URL: https://doi.org/10.1162/COMJ_a_00147 (visited on 10/25/2018).

Brown, Andrew R and Andrew C Sorensen (2007). "aa-cell in practice: An approach to musical live coding". In: *International computer music conference*, pp. 292–299.

Burlet, Gregory and Abram Hindle (2015). "An empirical study of end-user programmers in the computer music community". In: IEEE Press, pp. 292–302.

Burrows, Michael (1994). "A block-sorting lossless data compression algorithm". In: *SRS Research Report* 124.

Cabello, Ricardo (2010). *three.js*. URL: https://github.com/mrdoob/three.js (visited on 06/18/2025).

Cage, John (1951). *Imaginary Landscape No. 4*. For 12 radios and 24 performers.

Çakmak, Cem, Anil Çamci, and Angus Forbes (2016). "Networked virtual environments as collaborative music spaces". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 106–111.

Carroll, Lewis (1865). *Alice's Adventures in Wonderland*. Macmillan.

Cederqvist, Per, Roland Pesch, et al. (1992). *Version management with CVS*.

Chacon, Scott and Ben Straub (2014). *Pro Git*. en. Berkeley, CA: Apress. ISBN: 978-1-4842-0077-3 978-1-4842-0076-6. DOI: 10.1007/978-1-4842-0076-6. URL: http://link.springer.com/10.1007/978-1-4842-0076-6.

Chafe, Chris, Scott Wilson, and Daniel Walling (2002). "Physical model synthesis with application to Internet acoustics". In: *2002 IEEE international Conference on acoustics, speech, and signal processing*. Vol. 4. IEEE, pp. IV–4056.

Chen, Jason (June 2014). *Quill.js: Designing the Delta Format*. URL: https://quilljs.com/guides/designing-the-delta-format/ (visited on 04/01/2018).

Clark, A. and D. Chalmers (Jan. 1998). "The Extended Mind". en. In: *Analysis* 58.1, pp. 7–19. ISSN: 0003-2638, 1467-8284. DOI: 10.1093/analys/58.1.7. URL: https://academic.oup.com/analysis/article-lookup/doi/10.1093/analys/58.1.7.

Cowan, T.L. and Jasmine Rault (May 2018). "Onlining queer acts: Digital research ethics and caring for risky archives". en. In: *Women & Performance: a journal of feminist theory* 28.2, pp. 121–142. ISSN: 0740-770X, 1748-5819. DOI: 10.1080/0740770X.2018.1473985. URL: https://www.tandfonline.com/doi/full/10.1080/0740770X.2018.1473985 (visited on 06/04/2025).

Cycling '74 (2024). *Max 8*. URL: https://cycling74.com/products/max.

Dabbish, Laura et al. (Feb. 2012). "Social coding in GitHub: transparency and collaboration in an open software repository". en. In: *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. Seattle Washington USA: ACM, pp. 1277–1286. ISBN: 978-1-4503-1086-4. DOI: 10.1145/2145204.2145396. URL: https://dl.acm.org/doi/10.1145/2145204.2145396.

Davidson, Matthew (2013). *BEAP*. URL: https://www.ableton.com/en/blog/beap-powerful-modules-max-live/ (visited on 06/04/2025).

Dekker, Annet (2018). *Collecting and conserving net art: moving beyond conventional methods*. Routledge.

Deleuze, Gilles (1994). *Difference and repetition*. Columbia University Press.

Deleuze, Gilles and Félix Guattari (1987). *A Thousand Plateaus: Capitalism and Schizophrenia*. eng. Minneapolis: University of Minnesota Press. ISBN: 978-0-8166-1401-1 978-0-8166-1402-8.

DeVault, Drew (2018). *SourceHut*. URL: https://sourcehut.org.

Dourish, Paul (2004). *Where the action is: the foundations of embodied interaction*. eng. 1. MIT Press paperback ed. A Bradford book. Cambridge, Mass. London: MIT Press. ISBN: 978-0-262-04196-6 978-0-262-54178-7.

Duchamp, Marcel (1914). *Reseaux des stoppages*. French.

— (1923). *Le Grand Verre*. French.

Eghbal, Nadia (2016). *Roads and bridges: The unseen labor behind our digital infrastructure*. Ford Foundation.

Eriksson, Johan (2017). *Automatonism*. URL: https://www.automatonism.com/%20the-software/.

Estublier, Jacky (May 2000). "Software configuration management: a roadmap". en. In: *Proceedings of the Conference on The Future of Software Engineering*. Limerick Ireland: ACM, pp. 279–289. ISBN: 978-1-58113-253-3. DOI: 10.1145/336512.336576. URL: https://dl.acm.org/doi/10.1145/336512.336576.

Fasciani, Stefano and Habibur Rahman (2018). "Tangible virtual patch cords". In: *University of Wollongong*.

Franz, Max et al. (Jan. 2016). "Cytoscape.js: a graph theory library for visualisation and analysis". en. In: *Bioinformatics* 32.2, pp. 309–311. ISSN: 1367-4811, 1367-4803. DOI: 10.1093/bioinformatics/btv557. URL: https://academic.oup.com/bioinformatics/article/32/2/309/1744007.

Galfullin, B.N. and A.N. Novichkov (2000). "ClearCase-source code managing system". In: *2000 10th International Crimean Microwave Conference. "Microwave and Telecommunication Technology". Conference Proceedings (IEEE Cat. No.00EX415)*. Crimea, Ukraine: IEEE, pp. 90–93. ISBN: 978-966-572-048-5. DOI: 10.1109/CRMICO.2000.1255860. URL: http://ieeexplore.ieee.org/document/1255860/.

Gamma, Erich et al. (1993). "Design Patterns: Abstraction and Reuse of Object-Oriented Design". en. In: *ECOOP' 93 — Object-Oriented Programming*. Ed. by Gerhard Goos, Juris Hartmanis, and Oscar M. Nierstrasz. Vol. 707. Series Title: Lecture Notes in Com-

puter Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 406–431. ISBN: 978-3-540-57120-9 978-3-540-47910-9. DOI: 10.1007/3-540-47910-4_21. URL: http://link.springer.com/10.1007/3-540-47910-4_21 (visited on 06/19/2025).

Gaston, Ryan William (Feb. 2016a). *CEC — eContact! 17.4 — Plays Well With Others: Regarding modular synthesizer in collaborative performance practice by Ryan William Gaston*. URL: https://econtact.ca/17_4/gaston_playswell.html (visited on 02/13/2019).

— (2016b). "Well With Others: Regarding modular synthesizer in collaborative performance practice". In: *eContact!* 17.4. URL: https://econtact.ca/17_4/gaston_playswell.html (visited on 06/04/2025).

Gell, Alfred (1998). *Art and agency: an anthropological theory*. Oxford ; New York: Clarendon Press. ISBN: 978-0-19-828014-9 978-0-19-828013-2.

— (2015). "The Network of Standard Stoppages". English. In: *Distributed objects : meaning and mattering after Alfred Gell*. Ed. by Liana Chua and Mark Elliott, pp. 88–113. ISBN: 978-0-85745-743-1 0-85745-743-8.

GitHub Inc. (2008). *GitHub*. URL: https://github.com.

GitLab Inc. (2011). *GitLab*. URL: https://gitlab.com.

Glăveanu, Vlad Petre (2014). *Distributed Creativity: Thinking Outside the Box of the Creative Individual*. en. SpringerBriefs in Psychology. Cham: Springer International Publishing. ISBN: 978-3-319-05433-9 978-3-319-05434-6. DOI: 10.1007/978-3-319-05434-6. URL: https://link.springer.com/10.1007/978-3-319-05434-6.

Google (2024). *AudioWorklet*. URL: https://webaudio.github.io/web-audio-api/#AudioWorklet.

Goriunova, Olga (2014). *Fun and software: exploring pleasure, paradox and pain in computing*. Bloomsbury Publishing USA.

Gousios, Georgios, Martin Pinzger, and Arie Van Deursen (May 2014). "An exploratory study of the pull-based software development model". en. In: *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad India: ACM, pp. 345–355. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568260. URL: https://dl.acm.org/doi/10.1145/2568225.2568260 (visited on 06/02/2025).

Gresham-Lancaster, Scot (1998). "The aesthetics and history of the hub: The effects of changing technology on network computer music". In: *Leonardo Music Journal* 8.1. Publisher: The MIT Press, pp. 39–44.

Hallam, Elizabeth and Tim Ingold, eds. (Jan. 2007). *Creativity and Cultural Improvisation*. en. 1st ed. Routledge. ISBN: 978-1-00-313553-1. DOI: 10.4324/9781003135531. URL: https://www.taylorfrancis.com/books/9781000325454 (visited on 06/04/2025).

Harney, Stefano and Fred Moten (2013). "The undercommons: Fugitive planning and black study". In: Publisher: Minor Compositions.

Haverbeke, Marijn (2024a). *CodeMirror*. Version 6. URL: https://codemirror.net.

— (2024b). *CodeMirror MergeView Demo*. URL: https://codemirror.net/3/demo/merge.html.

Haworth, Christopher (2015). "Sound Synthesis Procedures as Texts: An Ontological Politcs in Electroacoustic and Computer Music". In: *Computer Music Journal* 39.1, pp. 41–58.

Hayles, N Katherine (2002). "Writing machines. Cambridge and London". In: *MIT Press. Retrieved June* 18, p. 2014.

Heidegger, Martin. and Andrew J. J. Mitchell (2012). *Bremen and Freiburg Lectures : Insight Into That Which Is and Basic Principles of Thinking*. English. URL: http://site.ebrary.com/id/10569640.

Holden, James (May 2015). *James Holden on using modular gear live and Max for Live patches*. URL: https://www.musicradar.com/news/tech/james-holden-on-using-modular-gear-live-and-max-for-live-patches-622023 (visited on 04/05/2019).

Hunt, James Wayne and M Douglas MacIlroy (1976). *An algorithm for differential file comparison*. Bell Laboratories Murray Hill.

Husserl, Edmund et al. (1964). *The phenomenology of internal time-consciousness*. English. The Hague: Nijhoff.

Hutchins, Charles Celeste (2015). "Live patch/live code". In: *Proceedings of the first international conference on live coding*, pp. 147–51.

Hutchins, Edwin. (2006). *Cognition in the Wild*. English. Cambridge, Mass. [u.a.]: MIT Press. ISBN: 0-262-58146-9 978-0-262-58146-2 0-262-08231-4 978-0-262-08231-0.

Ingold, Tim (Apr. 2013). *Making: Anthropology, Archaeology, Art and Architecture*. en. 1st ed. Routledge. ISBN: 978-0-203-55905-5. DOI: 10.4324/9780203559055. URL: https://www.taylorfrancis.com/books/9780203559055.

Jahns, Kevin (2017). *Yjs*. URL: https://yjs.dev.

Ji, Haru and Graham Wakefield (2018). *Artificial Nature: Insuperposition*. Exhibited at *Daejeon Biennale 2018 "Bio"*, Daejeon Museum of Art, Daejeon, Korea, July 16–October 24, 2018.

Jordà, Sergi (2002). "FMOL: Toward User-Friendly, Sophisticated New Musical Instruments". In: *Computer Music Journal* 26.3, pp. 23–39. ISSN: 01489267, 15315169. URL: http://www.jstor.org/stable/3681976 (visited on 08/17/2025).

Kalliamvakou, Eirini et al. (2014). "The code-centric collaboration perspective: Evidence from github". In: *The Code-Centric Collaboration Perspective: Evidence from Github, Technical Report DCS-352-IR, University of Victoria*.

Kasidiaris, Paris (2013). *xterm.js*. URL: https://github.com/xtermjs/xterm.js/ (visited on 06/18/2025).

Kelty, Christopher M. (2008). *Two Bits: The Cultural Significance of Free Software*. en. Duke University Press. ISBN: 978-0-8223-4242-7 978-0-8223-8900-2. DOI: 10.1215/9780822389002. URL: http://read.dukeupress.edu/books/book/1136/Two-BitsThe-Cultural-Significance-of-Free-Software (visited on 06/02/2025).

Kery, Mary Beth, Amber Horvath, and Brad Myers (2017). "Variolite: Supporting Exploratory Programming by Data Scientists". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA:

Association for Computing Machinery, pp. 1265–1276. ISBN: 9781450346559. DOI: 10.1145/3025453.3025626. URL: https://doi.org/10.1145/3025453.3025626.

Kleppmann, Martin and Alastair R. Beresford (Oct. 2017). "A Conflict-Free Replicated JSON Datatype". In: *IEEE Transactions on Parallel and Distributed Systems* 28.10, pp. 2733–2746. ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2697382. URL: http://ieeexplore.ieee.org/document/7909007/.

Kleppmann, Martin, Adam Wiggins, et al. (2019). "Local-first software: you own your data, in spite of the cloud". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 154–178.

Kodosky, Jeffrey (2020). "LabVIEW". In: *Proceedings of the ACM on Programming Languages* 4.HOPL. Publisher: ACM New York, NY, USA, pp. 1–54.

Lanier, Jaron (2003). "Why Gordian Software Has Convinced Me To Believe In The Reality Of Cats And Apples: A Talk With Jaron Lanier". In: *Interview with the Edge Foundation* 19, p. 2003.

Lave, Jean and Etienne Wenger (Sept. 1991). *Situated Learning: Legitimate Peripheral Participation*. 1st ed. Cambridge University Press. ISBN: 978-0-521-41308-4 978-0-521-42374-8 978-0-511-81535-5. DOI: 10.1017/CBO9780511815355. URL: https://www.cambridge.org/core/product/identifier/9780511815355/type/book.

Lee, Sang Won and Georg Essl (2014). "Models and opportunities for networked live coding". In: *Live Coding and Collaboration Symposium*. Vol. 1001, pp. 48109–2121.

Liang, Jenny T, Thomas Zimmermann, and Denae Ford (2022). "Understanding skills for OSS communities on GitHub". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 170–182.

Lindh, Martin (2018). "Beyond a Skeuomorphic Representation of Subtractive Synthesis." In: *IUI Workshops*.

lodash (2014). URL: https://github.com/lodash/lodash (visited on 06/19/2025).

Loeliger, Jon and Matthew McCullough (2012). *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.".

Magnusson, Thor (2014). "Herding cats: Observing live coding in the wild". In: *Computer Music Journal* 38.1. Publisher: MIT Press, pp. 8–16.

Manesh, Daniel, Douglas Bowman Jr, and Sang Won Lee (2024). "Sharp: Exploring version control systems in live coding music". In: *Proceedings of the 16th Conference on Creativity & Cognition*, pp. 426–437.

Mann, Yotam (2014). "Interactive Music with Tone.js". en. In: Paris, France. URL: https://wac.ircam.fr/pdf/wac15_submission_40.pdf.

Marcuse, Herbert (1964). *One-dimensional man : studies in the ideology of advanced industrial society*. English. Second edition. Section: xlix, 260 pages ; 21 cm. Boston: Beacon Press. ISBN: 0-8070-1417-6 978-0-8070-1417-2.

Marlow, Jennifer, Laura Dabbish, and Jim Herbsleb (Feb. 2013). "Impression formation in online peer production: activity traces and personal profiles in github". en. In: *Proceed-*

ings of the 2013 conference on Computer supported cooperative work. San Antonio Texas USA: ACM, pp. 117–128. ISBN: 978-1-4503-1331-5. DOI: 10.1145/2441776.2441792. URL: https://dl.acm.org/doi/10.1145/2441776.2441792.

Massachusetts Computer Associates, Inc. and Leslie Lamport (Oct. 2019). "Time, clocks, and the ordering of events in a distributed system". In: *Concurrency: the Works of Leslie Lamport*. Ed. by Dahlia Malkhi. Association for Computing Machinery. ISBN: 978-1-4503-7270-1. DOI: 10.1145/3335772.3335934. URL: https://dl.acm.org/citation.cfm?id=3335934 (visited on 06/02/2025).

Mayton, Brian D et al. (2012). "Patchwork: Multi-User Network Control of a Massive Modular Synthesizer." In: *NIME*.

Mazzola, Guerino et al. (2018). "Max®". In: *Basic Music Technology: An Introduction*. Publisher: Springer, pp. 145–156.

McHale, Brian (2012). *Constructing postmodernism*. Routledge.

McKiernan, Erin C et al. (July 2016). "How open science helps researchers succeed". en. In: *eLife* 5. ISSN: 2050-084X. DOI: 10.7554/eLife.16800. URL: https://elifesciences.org/articles/16800 (visited on 02/12/2019).

McKinney, Chad (2014). "Quick live coding collaboration in the web browser". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Citeseer, pp. 379–382.

— (2016). "Collaboration and embodiment in networked music interfaces for live performance". PhD thesis. University of Sussex.

McLean, Alex (2014). *TidalCycles: Live Coding Pattern Language*. URL: https://tidalcycles.org/.

McLean, Alex and Geraint Wiggins (2010). "Tidal–pattern language for the live coding of music". In: *Proceedings of the 7th sound and music computing conference*, pp. 331–334.

McPherson, Andrew and Youngmoo Kim (Dec. 2012). "The Problem of the Second Performer: Building a Community Around an Augmented Piano". In: *Computer Music Journal* 36.4, pp. 10–27. ISSN: 0148-9267. DOI: 10.1162/COMJ_a_00149. URL: https://doi.org/10.1162/COMJ_a_00149 (visited on 10/25/2018).

Mens, T. (May 2002). "A state-of-the-art survey on software merging". en. In: *IEEE Transactions on Software Engineering* 28.5, pp. 449–462. ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1000449. URL: http://ieeexplore.ieee.org/document/1000449/.

Miranda, Prof Eduardo R. and Hanns Holger Rutz (2010). "On the Traceability of the Compositional Process". en. In: URL: https://www.academia.edu/456488/On_the_Traceability_of_the_Compositional_Process (visited on 02/13/2019).

Möller, Christopher and Moritz Klack (2019). *React Flow*. URL: https://github.com/xyflow/xyflow (visited on 06/19/2025).

Nafus, Dawn (June 2012). "'Patches don't have gender': What is not open in open source software". en. In: *New Media & Society* 14.4, pp. 669–683. ISSN: 1461-4448, 1461-7315. DOI: 10.1177/1461444811422887. URL: https://journals.sagepub.com/doi/10.1177/1461444811422887 (visited on 06/02/2025).

Nilson, Click (2007). "Live coding practice". In: *Proceedings of the 7th international conference on New interfaces for musical expression*, pp. 112–117.

Nugroho, Yusuf Sulistyo, Hideaki Hata, and Kenichi Matsumoto (Jan. 2020). "How different are different diff algorithms in Git?: Use –histogram for code changes". en. In: *Empirical Software Engineering* 25.1, pp. 790–823. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-019-09772-z. URL: http://link.springer.com/10.1007/s10664-019-09772-z.

Oberlehner, Markus (Nov. 2020). *Partial Hydration Concepts: Lazy and Active*. en. URL: https://markus.oberlehner.net/blog/partial-hydration-concepts-lazy-and-active (visited on 06/19/2025).

Ogborn, David and Jamie Beverley (2017). "Estuary: Browser-based Collaborative Projectional Live Coding of Musical Patterns". en. In.

Oliveros, Pauline (2005). *Deep listening: A composer's sound practice*. IUniverse.

— (2009). "Networked Music: Low and High Tech". In: *Contemporary Music Review* 28.4-5. Publisher: Taylor & Francis, pp. 433–435.

Orlikowski, Wanda J. and Susan V. Scott (Jan. 2008). "10 Sociomateriality: Challenging the Separation of Technology, Work and Organization". en. In: *Academy of Management Annals* 2.1, pp. 433–474. ISSN: 1941-6520, 1941-6067. DOI: 10.5465/19416520802211644.

Palumbo, Michael (2015). *Art Video Games*. en-US. URL: http://www.palumbomichael.com/games (visited on 08/16/2025).

— (July 2017a). *A Technicity of Distributed Version Control Through Accelerationism*. URL: https://michaelpalumbo.github.io/spth6155/.

— (2017b). *Software Version Control Systems as Tools for Research-Creation in the Fine Arts: (or) Using git to Write a Paper About Using git*. URL: http://www.palumbomichael.com/self-referent/ (visited on 02/13/2019).

— (2019). *teaparty*. URL: https://github.com/worldmaking/teaparty.

Palumbo, Michael and Doug Van Nort (2020). "git show: Musical Creativity, Ideation, and GitHub". In: *Proceedings of the 26th International Symposium on Electronic Art*. Montreal, pp. 299–305. URL: https://www.isea-archives.org/docs/2020/isea2020_Proceedings.pdf.

Palumbo, Michael and Graham Wakefield (2018a). *Alicenode Editor*. URL: https://github.com/worldmaking/alicenode/tree/master/client.

— (2018b). *Cards*. GitHub Repository. URL: https://github.com/worldmaking/cards (visited on 06/18/2025).

— (2024). "From "What If?" To "What Diff?" and Back Again". In: *Modular Synthesis: Patching Machines and People*. Ed. by Ezra J Teboul, Einar Engström, and Andreas Kitzmann. Routledge, Taylor & Francis Group, pp. 449–455.

Palumbo, Michael, Graham Wakefield, and Alexander Zonta (2020). *Mischmasch*. original-date: 2019-03-06T17:22:20Z. URL: https://github.com/worldmaking/mischmasch (visited on 06/18/2025).

Palumbo, Michael, Graham Wakefield, and Alexander Zonta (Aug. 2022). *MSVR*. URL: https://github.com/worldmaking/mischmasch/tree/v/0.2.0.

Palumbo, Michael, Alexander Zonta, and Graham Wakefield (Jan. 2020). "Modular reality: Analogues of patching in immersive space". en. In: *Journal of New Music Research* 49.1, pp. 8–23. ISSN: 0929-8215, 1744-5027. DOI: 10.1080/09298215.2019.1706583. URL: https://www.tandfonline.com/doi/full/10.1080/09298215.2019.1706583 (visited on 06/24/2025).

Paris (2018). "Une approche du patching audio collaboratif". PhD Thesis. Université Paris 8 Vincennes-Saint-Denis.

Petre, Marian and Greg Wilson (2014). "Code review for and by scientists". In: *arXiv preprint arXiv:1407.5648*.

Pilato, C Michael, Ben Collins-Sussman, and Brian W Fitzpatrick (2008). *Version control with subversion: next generation open source version control*. " O'Reilly Media, Inc.".

Puckette, Miller et al. (1996). "Pure data: another integrated computer music environment". In: *Proceedings of the second intercollege computer music concerts*. Publisher: Tokyo, Japan, pp. 37–41.

Randell, Justin and Hillegonda C Rietveld (2024). "Eurorack to VCV Rack: Modular synthesis as compositional performance". In: *Modular Synthesis*. Focal Press, pp. 172–184.

Ricoeur, Paul (2004). *Memory, history, forgetting*. University of Chicago Press.

Roberts, Charles, Graham Wakefield, and Matthew Wright (2013). "The Web Browser As Synthesizer And Interface". In: DOI: 10.5281/ZENODO.1178648. (Visited on 06/04/2025).

Roberts, Charlie (2016). *genish.js*. URL: https://github.com/charlieroberts/genish.js.

Rochkind, Marc J. (1975). "The source code control system". In: *IEEE Transactions on Software Engineering* SE-1.4, pp. 364–370. DOI: 10.1109/TSE.1975.6312866.

Rodgers, Tara (2020). *Pink noises: Women on electronic music and sound*. Duke University Press.

Rohrhuber, Julian and Alberto De Campo (2009). "Improvising Formalisation: Conversational Programming and Live Coding". In: *New Computational Paradigms for Computer Music. Delatour France/Ircam-Centre Pompidou*.

Sawyer, Kieth and Stacy DeZutter (2009). "Distributed Creativity: How Collective Creations Emerge from Collaboration". In: *Psychology of Aesthetics, Creativity, and the Arts; American Psychological Association* 3.2, pp. 81–92.

Schwarz, Diemo (2025). *diff-for-max: Enhancing Version Control for Max Patches*. URL: https://github.com/diemoschwarz/diff-for-max.

Sèdes, Anne et al. (2017). "Teaching, investigating, creating: MUSICOLL". In: *Innovative Tools and Methods for Teaching Music and Signal Processing*.

Seiwald, Christopher (1995). *Perforce Version Control System*. Perforce Software Inc.

Serafin, Stefania et al. (2016). "Virtual reality musical instruments: State of the art, design principles, and future directions". In: *Computer Music Journal* 40.3, pp. 22–40.

Simondon, Gilbert, Ccile Malaspina, and John Rogove (2017). *On the mode of existence of technical objects*. Translated from the French. ISBN: 978-1-937561-03-1 1-937561-03-8.

Simonyi, Charles, Magnus Christerson, and Shane Clifford (2006). "Intentional software". In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. event-place: Portland, Oregon, USA. New York, NY, USA: Association for Computing Machinery, pp. 451–464. ISBN: 1-59593-348-4. DOI: 10.1145/1167473.1167511. URL: https://doi.org/10.1145/1167473.1167511.

Singer, Leif et al. (2013). "Mutual assessment in the social programmer ecosystem: An empirical investigation of developer profile aggregators". In: *Proceedings of the 2013 conference on Computer supported cooperative work*, pp. 103–116.

Spinellis, D. (Sept. 2005). "Version control systems". In: *IEEE Software* 22.5, pp. 108–109. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2005.140. URL: https://ieeexplore.ieee.org/document/1504674/.

Sterman, Sarah, Molly Jane Nicholas, and Eric Paulos (Nov. 2022). "Towards Creative Version Control". In: *Proc. ACM Hum.-Comput. Interact.* 6.CSCW2. DOI: 10.1145/3555756. URL: https://doi.org/10.1145/3555756.

Suchman, Lucy A. (2007). *Human-machine reconfigurations: Plans and situated actions, 2nd ed.* Human-machine reconfigurations: Plans and situated actions, 2nd ed. Pages: xii, 314. New York, NY, US: Cambridge University Press. ISBN: 0-521-67588-X (Paperback).

Sun, Chengzheng and David Chen (Mar. 2002). "Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems". In: *ACM Trans. Comput.-Hum. Interact.* 9.1, pp. 1–41. ISSN: 1073-0516. DOI: 10.1145/505151.505152. URL: https://doi.org/10.1145/505151.505152.

Sun, Chengzheng and Clarence Ellis (Nov. 1998). "Operational transformation in real-time group editors: issues, algorithms, and achievements". en. In: *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. Seattle Washington USA: ACM, pp. 59–68. ISBN: 978-1-58113-009-6. DOI: 10.1145/289444.289469. URL: https://dl.acm.org/doi/10.1145/289444.289469.

Tanaka, Atau (2000). "Musical performance practice on sensor-based instruments". In: *Trends in gestural control of music* 13.389-405. Publisher: Ircam Paris, p. 284.

— (2006). "Interaction, Experience and the Future of Music". en. In: *Consuming Music Together*. Ed. by Kenton O'Hara and Barry Brown. Vol. 35. Series Title: Computer Supported Cooperative Work. Berlin/Heidelberg: Springer-Verlag, pp. 267–288. DOI: 10.1007/1-4020-4097-0_13.

Tauberer, Joshua (2025). *JOT*. URL: https://github.com/JoshData/jot (visited on 06/18/2025).

Terrien, Anthony (2011). *jQuery-Knob*. URL: https://github.com/aterrien/jQuery-Knob (visited on 06/19/2025).

Tichy, Walter F. (July 1985). "Rcs — a system for version control". en. In: *Software: Practice and Experience* 15.7, pp. 637–654. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.

4380150703. URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.4380150703.

TOPLAP (2004). *TOPLAP Manifesto (Draft)*. URL: https://toplap.org/wiki/ManifestoDraft.

Torvalds, Linus, Junio Hamano, et al. (2005). "Git". In: *Software Freedom Conservancy*, p. 20.

Treude, Christoph and Margaret-Anne Storey (2010). "Work item tagging: Communicating concerns in collaborative software development". In: *IEEE Transactions on Software Engineering* 38.1. Publisher: IEEE, pp. 19–34.

Truax, Barry (Jan. 1996). "Soundscape, acoustic communication and environmental sound composition". In: *Contemporary Music Review* 15.1, pp. 49–65. ISSN: 0749-4467. DOI: 10.1080/07494469600640351. URL: http://www.tandfonline.com/doi/abs/10.1080/07494469600640351 (visited on 08/17/2025).

Tsay, Jason, Laura Dabbish, and James Herbsleb (May 2014). "Influence of social and technical factors for evaluating contribution in GitHub". en. In: *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad India: ACM, pp. 356–366. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568315. URL: https://dl.acm.org/doi/10.1145/2568225.2568315.

Van Nort, Doug (2023). "Distributed Networks of Listening and Sounding: 20 Years of Telematic Musicking". In: *Journal of Network Music and Arts* 5.1, p. 6.

Vasilescu, Bogdan, Alexander Serebrenik, and Vladimir Filkov (2015). "A Data Set for Social Diversity Studies of GitHub Teams". In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR '15. Piscataway, NJ, USA: IEEE Press, pp. 514–517. ISBN: 978-0-7695-5594-2. URL: http://dl.acm.org/citation.cfm?id=2820518.2820601.

W3C WebAssembly Working Group (2022). *WebAssembly Core Specification*. W3C Recommendation. Accessed May 24, 2025. URL: https://www.w3.org/TR/wasm-core-1/.

W3C WebRTC Working Group (2021). *Web Real-Time Communications (WebRTC)*. URL: https://www.w3.org/TR/webrtc/.

Wakefield, Graham (2019). *node-gles3*. URL: https://github.com/worldmaking/node-gles3 (visited on 06/18/2025).

Wakefield, Graham and Michael Palumbo (2019). *Personal communication*.

— (2020). *Graph Operational Transform*. URL: https://github.com/worldmaking/gotlib (visited on 06/18/2025).

Wakefield, Graham, Michael Palumbo, and Alexander Zonta (2020). "Affordances and Constraints of Modular Synthesis in Virtual Reality". In: Publisher: Zenodo. ISSN: 2220-4806. DOI: 10.5281/ZENODO.4813182.

Wakefield, Graham, Charlie Roberts, et al. (2014). "Collaborative live-coding with an immersive instrument". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 505–508.

Warth, Alessandro et al. (2011). "Worlds: Controlling the scope of side effects". In: *European Conference on Object-Oriented Programming*. Springer, pp. 179–203.

WebVR (2016). URL: https://github.com/immersive-web/webvr.

Weinberg, Gil (2005). "Interconnected Musical Networks: Toward a Theoretical Framework". In: *Computer Music Journal* 29.2, pp. 23–39. ISSN: 01489267, 15315169. URL: http://www.jstor.org/stable/3681711 (visited on 08/17/2025).

ws (May 2025). en. URL: https://www.npmjs.com/package/ws.

Yoon, YoungSeok and Brad A. Myers (2015). "Supporting Selective Undo in a Code Editor". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1, pp. 223–233. DOI: 10.1109/ICSE.2015.43.

Yoon, YoungSeok, Brad A. Myers, and Sebon Koo (2013). "Visualization of fine-grained code change history". In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pp. 119–126. DOI: 10.1109/VLHCC.2013.6645254.

Zagalsky, Alexey et al. (2015). "The Emergence of GitHub As a Collaborative Platform for Education". In: *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW '15. event-place: Vancouver, BC, Canada. New York, NY, USA: ACM, pp. 1906–1917. ISBN: 978-1-4503-2922-4. DOI: 10.1145/2675133.2675284. URL: http://doi.acm.org.ezproxy.library.yorku.ca/10.1145/2675133.2675284.

Zicarelli, David (Oct. 2018). *Article: Announcing Expo '74 | Cycling '74*. en. URL: https://cycling74.com/articles/announcing-expo-'74 (visited on 06/18/2025).

Appendix A

Forking Paths Playtesting Questionnaire

- Your name (first, last)
- What are your pronouns? (optional)
- Please list your artist pseudonym(s) (optional)
- What is your email address?
- What genre(s) of music do you play?
- Please describe your own hardware modular synth
- What led you to choose some (or all) of the modules you have in your hardware modular synth?
- When you play your modular synth, do you play it alone or with others?
- If you selected "with others" or "both", what instrument(s) do the others play?
- If you listed any electronic instruments, do you share any of the following signals with each other?
- In what ways did Forking Paths feel like playing a hardware modular synthesizer?
- In what ways did Forking Paths not feel like playing a hardware modular synthesizer?
- What (if any) software modular synths have you played before?
- In what ways did Forking Paths feel like playing a software modular synthesizer?
- In what ways did Forking Paths not feel like playing a software modular synthesizer?
- What was it like to play Forking Paths alone?

- If you played Forking Paths with another player, what was that like?
- Describe your process of learning to play with the Forking Paths "History Sequencer"
- In what ways was playing Forking Paths fun?
- If you had the Forking Paths "History Sequencer" in your studio or on-stage, in what ways could it contribute to your creativity or practice?
- Could you describe your overall impression of Forking Paths?
- Was anything particularly enjoyable or frustrating about using Forking Paths?
- Please tell us about anything that did not work the way you expected it to
- Anything else you'd like to share?
- What was it like to recall a previous patch state?
- Did the visual representation of different versions or branches effectively communicate the history and structure of your work?
- How did Forking Paths impact your ability to experiment freely or explore new ideas?
- What features or improvements would you most like to see added or adjusted in Forking Paths?
- Are there aspects of Forking Paths you feel need clarification or better documentation?

Appendix B

Code Excerpts from the Synth View

A selection of notable code listings from `synthApp.js`, the Script for the Synth View Window. Each excerpt is documented in this appendix, with its corresponding location in `synthApp.js` indicated in the Lines column below.

Table B.1: Code excerpts from the Synth View

Listing	Excerpt	Purpose	Lines
B.1	startAutomerge()	Initializes Automerge documents	691-820
B.2	createNewPatchHistory()	Clears patch history, DSP & Cytoscape	1097-1274
B.3	applyChange() examples	paramUpdate & gesture changeNodes	3536-3578
B.4	applyChange()	Manages document updates	834-961
B.5	loadVersion()	Handles changeNodes & branching	1505-1656
B.6	updateSynthWorklet()	Handles changes to DSP	4873-4936
B.7	updateCytoscapeFromDocument()	Handles changes to Cytoscape synth	1302-1405
B.8	createMerge()	Merges two changeNodes	1822–1901
B.9	startCable()	Creates a temporary cable	5213-5255
B.10	handleRemoteCables()	Renders peer temporary cables	5257–5314
B.11	isEdgeInCycle()	Graph cycle detection	5581–5703

B.1 Automerge Initialization

```
1 let automergeRunning = false
2 //* AUTOMERGE IMPLEMENTATION
3 async function startAutomerge () {
4     automergeRunning = true
5     // Load Automerge asynchronously and assign it to the global
        variable
```

```
6       Automerge = await import('@automerge/automerge');
7
8       // if the room contains 2 peers:
9       if(room && roomDetails.peer1 && roomDetails.peer2){
10          patchHistory = Automerge.init();
11          syncState = Automerge.initSyncState(); // this must exist
                here
12          console.log("Joining active room. Waiting for sync.");
13          holdSnackbar('Syncing with peers, please wait...')
14          // arm the reload function. if peer connection takes too
                long, it will reload the page
15          forceReload(true)
16          return
17      } else {
18          const saved = await loadDocument(patchHistoryKey);
19
20          if (saved) {
21              patchHistory = Automerge.load(saved);
22          } else {
23              patchHistory = Automerge.from({
24                  title: "Forking Paths Synth",
25                  forked_from_id: null, // used by the database to
                        either determine this as the root of a tree of
                        patch histories, or a fork from a stored history
26                  authors: [], // this will get added to as the doc is
                        forked from the database
27                  branches: {},
28                  branchOrder: [],
29                  docs: {},
30                  head: {
31                      hash: null,
32                      branch: null
33                  } ,
34                  userSettings: {
35                      focusNewBranch: true
36                  },
37                  sequencer: {
38                      bpm: 120,
39                      ms: 500,
40                      traversalMode: 'Sequential'
41                  },
42                  synth: {
43                      rnboDeviceCache: null,
44                  }
45              });
46              console.log("No saved patchHistory found. Starting fresh
                    :", patchHistoryKey);
47              await saveDocument(patchHistoryKey, Automerge.save(
                    patchHistory));
48          }
49      }
```

```
50
51      syncState = Automerge.initSyncState()
52
53      // * synth changes document
54      docID = 'forkingPathsDoc'; // Unique identifier for the document
55      // Load the document from patchHistory's store in IndexedDB or
            create a new one if it doesn't exist
56
57      // currentBranch = await loadDocument(docID);
58      // if patchHistory doesn't contain a document, create a new one
59      if (!patchHistory.docs[patchHistory.head.branch]) {
60
61          currentBranch = Automerge.init();
62
63          // load synthFile from indexedDB
64          // let synthFile = JSON.parse(localStorage.getItem('
                synthFile'))
65          // console.log('synthFile', synthFile)
66          if (patchHistory.synthFile) {
67
68              createNewPatchHistory(patchHistory.synthFile)
69              // const firstChangeLabel = synthFile.name
70              // ? `load_synth:${synthFile.name}`
71              // : 'load_synth:unnamed';
72
73              // currentBranch = Automerge.change(currentBranch,
                    firstChangeLabel, (currentBranch) => {
74              //     currentBranch.title = config.patchHistory.
                    firstBranchName;
75              //     currentBranch.changeNode = { msg:
                    firstChangeLabel };
76              //     currentBranch.elements = synthFile.visualGraph?.
                    elements?.nodes || [];
77              //     currentBranch.synth = {
78              //         graph: synthFile.audioGraph || {
79              //         modules: {},
80              //         connections: []
81              //         }
82              //     };
83              //     currentBranch.sequencer = { tableData: [] };
84              // });
85
86              // const hash = Automerge.getHeads(currentBranch)[0];
87              // previousHash = hash;
88
89              // patchHistory = Automerge.change(patchHistory, (
                    patchHistory) => {
90              //     patchHistory.branches[config.patchHistory.
                    firstBranchName] = {
91              //         head: hash,
92              //         root: null,
```

```
93              //          parent: null,
94              //          history: [{ hash: hash, parent: null, msg:
                    firstChangeLabel }]
95              //      };
96
97              //      patchHistory.docs[config.patchHistory.
                    firstBranchName] = Automerge.save(currentBranch);
98              //      patchHistory.head.branch = config.patchHistory.
                    firstBranchName;
99              //      patchHistory.head.hash = hash;
100             //      patchHistory.branchOrder.push(config.patchHistory
                    .firstBranchName);
101             // });
102
103             // // send doc to history app
104             // reDrawHistoryGraph()
105
106         } else {
107             console.log("No synth file found. currentBranch
                    initialized but not changed.");
108             previousHash = null;
109
110             try {
111                 // Fetch the Demo Synth
112                 const response = await fetch(`/assets/synths/${
                        import.meta.env.VITE_FIRST_SYNTH}.fpsynth`);
113
114                 if (!response.ok) {
115                     throw new Error(`HTTP error! status: ${response.
                            status}`);
116                 }
117
118                 // Parse the response as JSON
119                 const fileContent = await response.json();
120
121                 // Store the JSON string in localStorage if needed
122                 localStorage.setItem('synthFile', JSON.stringify(
                        fileContent));
123
124                 // Process the JSON content
125                 createNewPatchHistory(fileContent);
126
127                 // enable new history button now that a synth has
                        been loaded
128                 // UI.menus.file.newPatchHistory.disabled = false
129
130             } catch (error) {
131                 console.error("Error loading template file:", error)
                        ;
132             }
133
```

```
134
135
136             // patchHistory = Automerge.change(patchHistory, (
                    patchHistory) => {
137             //     // Only set up empty branch patchHistorydata
                    no doc yet
138             //     patchHistory.branches[config.patchHistory.
                    firstBranchName] = {
139             //         head: null,
140             //         root: null,
141             //         parent: null,
142             //         history: []
143             //     };
144             //     patchHistory.head.branch = config.patchHistory.
                    firstBranchName;
145             //     patchHistory.head.hash = null;
146             //     patchHistory.branchOrder.push(config.patchHistory
                    .firstBranchName);
147             // });
148         }
149
150     } else {
151
152         // patchHistory does contain at least one document, so grab
                whichever is the one that was last looked at
153         currentBranch = Automerge.load(patchHistory.docs[
                patchHistory.head.branch]);
154
155         // wait 1 second before loading content (give the audio
                worklet a moment to load)
156         setTimeout(()=>{
157             updateSynthWorklet('loadVersion', currentBranch.synth.
                    graph, null, currentBranch.type)
158
159             updateCytoscapeFromDocument(currentBranch, 'buildUI');
160
161             previousHash = patchHistory.head.hash
162
163             // send doc to history app
164             reDrawHistoryGraph()
165
166             // load the draw canvas
167             if(currentBranch.drawing){
168                 loadCanvasVersion(currentBranch.drawing)
169             }
170
171         }, 1000);
172     }
173 }
```

Listing B.1: Initial peer setup when joining a room

B.2 createNewPatchHistory()

```
function createNewPatchHistory(synthFile){
    // clear the pen tool interface
    resetDrawing()
    // delete the document in the indexedDB instance
    deleteDocument('patchHistory')
    // clear DSP
    updateSynthWorklet('clearGraph')
    // ensure floating UI container divs are removed
    clearparamContainerDivs()
    // clear the sequencer
    sendMsgToHistoryApp({
        appID: 'forkingPathsMain',
        cmd: 'newPatchHistory'

    })
    // tell server to erase the patchHistory & send a blank DAG to
       client(s)
    ws.send(JSON.stringify({
        cmd: 'clearHistoryGraph'
    }))

    // Clear existing elements from Cytoscape instance
    synthGraphCytoscape.elements().remove();

    // remove all dynamicly generated UI overlays (knobs, umenus,
       etc)
    removeUIOverlay('allNodes')
    // ensure their container divs are removed too
    clearparamContainerDivs()
    // init new patch history for Automerge
    let patchHistoryJSON = {
        title: "Forking Paths Patch History",
        forked_from_id: null, // used by the database to either
           determine this as the root of a tree of patch histories,
           or a fork from a stored history
        authors: [], // this will get added to as the doc is forked
           from the database
        branches: {},
        branchOrder: [],
        docs: {},
        head: {
            hash: null,
            branch: config.patchHistory.firstBranchName
        },

        userSettings: {
            focusNewBranch:false
        },
        sequencer: {
```

```
45              bpm: 120,
46              ms: 500,
47              traversalMode: 'Sequential'
48          },
49          synth: {
50              rnboDeviceCache: null,
51          },
52
53      }
54      if(synthFile){
55          patchHistoryJSON.synthFile = synthFile
56      } else if (patchHistory.synthFile){
57          // if a synth file had been previously loaded, load it again
58          patchHistoryJSON.synthFile = patchHistory.synthFile
59          synthFile = patchHistory.synthFile
60      }
61      // assign patch history to automerge
62      patchHistory = Automerge.from(patchHistoryJSON)
63      // clear the current automerge doc
64      currentBranch = Automerge.init();
65
66      if(synthFile || patchHistory.synthFile){
67
68          if(!patchHistory.synthFile) { synthFile = patchHistory.
                synthFile }
69
70          let amMsg = makeChangeMessage(config.patchHistory.
                firstBranchName, `loaded ${synthFile.filename}`)
71
72          // Apply initial changes to the new document
73          currentBranch = Automerge.change(currentBranch, amMsg, (
                currentBranch) => {
74              currentBranch.title = config.patchHistory.
                    firstBranchName;
75              currentBranch.elements = [ ]
76              patchHistory.synthFile.visualGraph.elements.nodes.
                    forEach((node)=>{
77                  currentBranch.elements.push(node)
78              })
79
80              currentBranch.synth = {
81                  graph: synthFile.audioGraph,
82                  connections: []
83              }
84
85              audioGraphDirty = true
86
87              currentBranch.drawing = []
88          }, onChange, `loaded ${synthFile.filename}`);
89
```

```
90          updateSynthWorklet('loadVersion', currentBranch.synth.graph,
                null, currentBranch.changeNode)
91
92          // load synth graph from file into cytoscape
93          synthGraphCytoscape.json(patchHistory.synthFile.visualGraph)
94
95          synthFile.visualGraph.elements.nodes.forEach((node, index)
                =>{
96              // set module grabbable to false -- prevents module
                    movements in main view
97              if(node.classes === ':parent'){
98                  // synthFile.visualGraph.elements.nodes[index].
                        grabbable = false
99                  // lock the module's position
100                 synthGraphCytoscape.getElementById(node.data.id).
                        lock();
101             }
102             // create overlays
103             if(node.classes === 'paramAnchorNode'){
104                 let value = patchHistory.synthFile.audioGraph.
                        modules[node.data.parent].params[node.data.label]
105                 createFloatingOverlay(node.data.parent, node, index,
                        value)
106
107                 // index++
108             }
109         })
110
111
112         setTimeout(() => {
113             updateKnobPositionAndScale('all');
114             // Make all nodes non-draggable
115
116         }, 10); // Wait for the current rendering cycle to complete
117     } else {
118         console.log('non synthFile')
119         console.warn('synthFile nor patchHistory.synthFile not found
                ')
120
121     }
122
123     let hash = Automerge.getHeads(currentBranch)[0]
124     previousHash = hash
125
126     let msg = 'blank_patch'
127     if (synthFile){
128         msg = `loaded ${synthFile.filename}`
129     }
130     patchHistory = Automerge.change(patchHistory, (patchHistory) =>
            {
```

```
131         patchHistory.branches[config.patchHistory.firstBranchName] =
                {
132             head: hash,
133             root: null,
134             parent: null,
135             // doc: currentBranch,
136             history: [ {hash: hash, parent: null, msg: msg} ]
137         }
138
139         // encode the doc as a binary object for efficiency
140         patchHistory.docs[config.patchHistory.firstBranchName] =
                Automerge.save(currentBranch)
141         patchHistory.head.branch = config.patchHistory.
                firstBranchName
142         patchHistory.head.hash = hash
143         patchHistory.branchOrder.push(patchHistory.head.branch)
144         patchHistory.synthFile = synthFile
145
146     });
147
148     docUpdated = true
149     previousHash = patchHistory.head.hash
150     // send doc to history app
151     reDrawHistoryGraph()
152
153     // store it in the database
154     const patch_binary = fromByteArray(Automerge.save(patchHistory))
155     ws.send(JSON.stringify({
156         cmd: 'newPatchHistory',
157         data: {
158             name: `${chance.animal()} ${uuidv7().split('-')[2]}`,
159             authors: [ thisPeerID ],
160             description: null,
161             modules: ['test'], // can be pulled from your patch
                    graph
162             synth_template: patchHistory.synthFile, // JSON object
163             patch_binary: patch_binary, // base64-encoded string
164             forked_from_id: patchHistory.forked_from_id, // or null
                    if this is a root version
165         }
166     }))
167
168     // get a binary from the new patchHistory
169     const fullBinary = Automerge.save(patchHistory);
170     // send it to any connected peer(s)
171     let message = {
172         cmd: 'replacePatchHistory',
173         data: fromByteArray(fullBinary)  // base64 encoded or send
                as Uint8Array directly if channel supports it
174     }
175     // sync with peer(s)
```

```
176     sendDataChannelMessage(message)
177 }
```

Listing B.2: Clears the patchHistory Automerge document, clears DSP and Cytoscape.js states, reloads the current .fpSynth file as the first change in the patchHistory and redraws UI elements

B.3 applyChange Examples

```
1  // Listen for mouse up event on the document
2  document.addEventListener('mouseup', function(event) {
3      hid.mouse.left = false
4
5      // if the player has been playing with a param knob, we need to
           store it in automerge
6      if(Object.keys(groupChange).length > 0){
7          // if we are storing a single param change, do a paramUpdate
8          if(groupChange.values.length === 1){
9              // Update in Automerge
10             currentBranch = applyChange(currentBranch, (
                   currentBranch) => {
11                 // update DSP
12                 currentBranch.synth.graph.modules[groupChange.
                       parentNode].params[groupChange.paramLabel] =
                       groupChange.values[0];
13                 audioGraphDirty = true;
14                 // set the changeNode
15                 currentBranch.changeType = {
16                     msg: 'paramUpdate',
17                     param: groupChange.paramLabel,
18                     parent: groupChange.parentNode,
19                     value: groupChange.values
20                 }
21             }, onChange, `paramUpdate ${groupChange.paramLabel} = ${
                   groupChange.values[0]}$PARENT ${groupChange.
                   parentNode}`);
22
23         } else if(groupChange.values.length > 1){
24             // are storing a gesture
25             // Update in Automerge
26             currentBranch = applyChange(currentBranch, (
                   currentBranch) => {
27                 currentBranch.synth.graph.modules[groupChange.
                       parentNode].params[groupChange.paramLabel] =
                       groupChange.values;
28                 audioGraphDirty = true;
29                 // set the change type
30                 currentBranch.changeType = {
```

```
31                      msg: 'gesture',
32                      param: groupChange.paramLabel,
33                      parent: groupChange.parentNode,
34                      values: groupChange.values,
35                      timestamps: groupChange.timestamps
36                  }
37              }, onChange, `gesture ${groupChange.paramLabel}$PARENT $
                    {groupChange.parentNode}`);
38
39          }
40
41          // clear the groupChange
42          groupChange = { }
43      }
44
45      // ... rest of code omitted for brevity
46  });
```

Listing B.3: Creates a new changeNode, updates the DSP graph, adds both to the patch history. There are many instances of this throughout the synthApp.js, and the method for creating paramUpdate or gesture changeNodes is what is documented here.

B.4 applyChange

```
1   // handle document changes and call a callback
2   function applyChange(doc, changeCallback, onChangeCallback,
        changeMessage) {
3       // in this condition, we are applying a change on the current
            branch
4       if(automergeDocuments.newClone === false ){
5           let amMsg = makeChangeMessage(patchHistory.head.branch,
                changeMessage)
6           // we are working from a head
7
8           // grab the current hash before making the new change:
9           previousHash = patchHistory.head.hash
10
11          // Apply the change using Automerge.change
12          currentBranch = Automerge.change(currentBranch, amMsg,
                changeCallback);
13
14
15          // If there was a change, call the onChangeCallback
16          if (currentBranch !== doc && typeof onChangeCallback === '
                function') {
17              let hash = Automerge.getHeads(currentBranch)[0]
18
```

```
19              patchHistory = Automerge.change(patchHistory, (
                    patchHistory) => {
20
21                  // if the current patchHistory was loaded from the
                        database, then we need to create a fork for this
                        new change
22                  if (patchHistory.hasBeenModified === false) {
23                      patchHistory.forked_from_id = patchHistory.
                            databaseID
24                      patchHistory.hasBeenModified = true
25                      forkHistoryInDatabase(patchHistory.databaseID)
26                  }
27                  // Initialize the branch patchHistorydata if it
                        doesn't already exist
28                  if (!patchHistory.branches[patchHistory.head.branch
                        ]) {
29                      patchHistory.branches[patchHistory.head.branch]
                            = { head: null, history: [] };
30                  }
31                  // Update the head property
32                  patchHistory.branches[patchHistory.head.branch].head
                        = hash;
33
34                  // Push the new history entry into the existing
                        array
35                  patchHistory.branches[patchHistory.head.branch].
                        history.push({
36                      hash: hash,
37                      parent: previousHash,
38                      msg: changeMessage,
39                      timeStamp: new Date().getTime()
40
41                  });
42
43                  // encode the doc as a binary object for efficiency
44                  patchHistory.docs[patchHistory.head.branch] =
                        Automerge.save(currentBranch)
45                  // store the HEAD info
46                  patchHistory.head.hash = hash
47                  patchHistory.timeStamp = new Date().getTime()
48                  //? patchHistory.head.branch = currentBranch.title
49
50              });
51
52
53
54              updatePatchHistoryDatabase()
55
56              onChangeCallback(currentBranch);
57          }
58          return currentBranch;
```

```
59      } else {
60          // player has made changes to an earlier version, so create
                a branch and set currentBranch to new clone
61
62          // store previous currentBranch in automergeDocuments, and
                its property is the hash of its head
63          automergeDocuments.otherDocs[patchHistory.head.branch] =
                currentBranch
64          // set currentBranch to current cloned doc
65          currentBranch = Automerge.clone(automergeDocuments.current.
                doc)
66
67          // create a new branch name
68          const newBranchName = uuidv7();
69          // use the new branch title
70          let amMsg = makeChangeMessage(patchHistory.head.branch,
                changeMessage)
71
72          // grab the current hash before making the new change:
73          previousHash = Automerge.getHeads(currentBranch)[0]
74
75          // Apply the change using Automerge.change
76          currentBranch = Automerge.change(currentBranch, amMsg,
                changeCallback);
77          let hash = Automerge.getHeads(currentBranch)[0]
78
79          // If there was a change, call the onChangeCallback
80          if (currentBranch !== doc && typeof onChangeCallback === '
                function') {
81              const timestamp = new Date().getTime()
82              patchHistory = Automerge.change(patchHistory, (
                    patchHistory) => {
83
84                  // create the branch
85                  patchHistory.branches[newBranchName] = {
86                      head: hash,
87                      parent: previousHash,
88                      history: [{
89                          hash: hash,
90                          msg: changeMessage,
91                          parent: previousHash,
92                          timeStamp: timestamp
93                      }]
94                  }
95
96                  // store current doc
97                  patchHistory.docs[newBranchName] = Automerge.save(
                        currentBranch)
98
99                  // store the HEAD info
100                 patchHistory.head.hash = hash
```

```
101                 patchHistory.head.branch = newBranchName
102
103                 patchHistory.timeStamp = timestamp
104
105                 // store the branch name so that we can ensure its
                        ordering later on
106                 patchHistory.branchOrder.push(newBranchName)
107
108                 // if the current patchHistory was loaded from the
                        database, then we need to create a fork for this
                        new change
109                 if (patchHistory.hasBeenModified === false) {
110                     patchHistory.forked_from_id = patchHistory.
                            databaseID
111                     patchHistory.hasBeenModified = true
112                     forkHistoryInDatabase(patchHistory.databaseID)
113                 }
114             });
115
116             // makeBranch(changeMessage, Automerge.getHeads(newDoc)
                    [0])
117             onChangeCallback(currentBranch);
118
119             updatePatchHistoryDatabase()
120             automergeDocuments.newClone = false
121
122         }
123         return currentBranch;
124
125     }
126
127
128 }
129
130     // define the onChange Callback
131     onChange = () => {
132         // send to peer(s)
133         sendSyncMessage()
134         // set docUpdated so that indexedDB will save it
135         docUpdated = true
136         // update the historyGraph in the server
137         reDrawHistoryGraph()
138         // set audioDirty flag
139         if(audioGraphDirty){
140             audioGraphDirty = false
141     }
```

```
};
```

Listing B.4: applyChange() manages Automerge document updates within Forking Paths. If the user is working on the current branch, the function applies changes directly and appends a new commit to that branch's history. If the user is editing a past state (i.e., newClone is true), the function forks a new branch, applies the change, and updates the patch history metadata accordingly. In both cases, the current Automerge document is saved, a new hash is generated, and the onChangeCallback is triggered to sync peers, update the UI, and redraw the version history graph.

B.5 loadVersion()

```
// Load a version from the DAG
async function loadVersion(targetHash, branch, fromPeer,
    fromPeerSequencer) {
    // get the head from this branch
    let head = patchHistory.branches[branch].head
    // get the automerge doc associated with the requested hash
    let requestedDoc = loadAutomergeDoc(branch)
    // Use `Automerge.view()` to view the state at this specific
        point in history
    const historicalView = Automerge.view(requestedDoc, [targetHash
        ]);
    // load the pen tool drawing
    if(historicalView.drawing){
        loadCanvasVersion(historicalView.drawing)
    }

    // Check if we're on the head; reset clone if true (so we don't
        trigger opening a new branch with changes made to head)
    // compare the point in history we want (targetHash) against the
        head of its associated branch (head)
    if (head === targetHash){

        // no need to create a new branch if the user makes changes
            after this operation
        automergeDocuments.newClone = false
        // send the synth graph from this point in the history to
            the DSP worklet first
        updateSynthWorklet('loadVersion', historicalView.synth.graph
            )
        // send the visual graph from this point in the history to
            the synth cytoscape
        updateCytoscapeFromDocument(historicalView);
        // update patchHistory to set the current head and change
            hash
        patchHistory = Automerge.change(patchHistory, (patchHistory)
            => {
```

```
26              // store the HEAD info (the most recent HEAD and branch
                    that were viewed or operated on)
27              patchHistory.head.hash = targetHash
28              patchHistory.head.branch = branch
29          });
30          // set global var for easy checking
31          automergeDocuments.current = {
32              doc: requestedDoc
33          }
34      }
35      // this is necessary for loading a hash on another branch that
            ISN'T the head
36      else if (branch != patchHistory.head.branch) {
37
38          // if we are dealing with a blank patch, then clear the
                audio graph
39          if(!historicalView.synth){
40              updateSynthWorklet('clearGraph')
41          } else {
42              // send the synth graph from this point in the history
                    to the DSP worklet first
43              updateSynthWorklet('loadVersion', historicalView.synth.
                    graph, null, historicalView.changeType)
44          }
45
46          // send the visual graph from this point in the history to
                the synth cytoscape
47          updateCytoscapeFromDocument(historicalView);
48          // set global var for easy checking
49          automergeDocuments.current = {
50              doc: requestedDoc
51          }
52          // update patchHistory to set the current head and change
                hash
53          patchHistory = Automerge.change(patchHistory, (patchHistory)
                => {
54              // store the HEAD info (the most recent HEAD and branch
                    that were viewed or operated on)
55              patchHistory.head.hash = targetHash
56              patchHistory.head.branch = branch
57          });
58          // set newClone to true
59          automergeDocuments.newClone = true
60      }
61      // the selected hash belongs to the current branch
62      else {
63          // send the synth graph from this point in the history to
                the DSP worklet first
64          updateSynthWorklet('loadVersion', historicalView.synth.graph
                , null, historicalView.changeType)
```

```
65          // send the visual graph from this point in the history to
                the synth cytoscape
66          updateCytoscapeFromDocument(historicalView);
67          // create a clone of the branch in case the player begins
                making changes
68          let clonedDoc = Automerge.clone(historicalView)
69          // store it
70          automergeDocuments.current = {
71              doc: clonedDoc
72          }
73          // set newClone to true
74          automergeDocuments.newClone = true
75
76          // update patchHistory to set the current head and change
                hash
77          patchHistory = Automerge.change(patchHistory, (patchHistory)
                => {
78              // store the HEAD info (the most recent HEAD and branch
                    that were viewed or operated on)
79              patchHistory.head.hash = targetHash
80              patchHistory.head.branch = branch
81          });
82      }
83
84      // Optional sync/permission handling AFTER local load
85      const recallMode = getVersionRecallMode();
86      // ensure that loadVersion calls from the peer don't make past
            this point, becuase otherwise they'd send it back and forth
            forever
87      if (recallMode === 'openLoadVersion' && !fromPeer && !
            fromPeerSequencer) {
88          console.log('openVersionRecall')
89          openVersionRecall(targetHash, branch);
90      }
91
92      if (recallMode === 'requestOpenLoadVersion'  && !fromPeer) {
93          // requestVersionRecallWithPermission(currentBranch,
                Automerge.getHeads(currentBranch)[0], patchHistory.head.
                branch);
94          console.warn('not set up yet')
95      }
96
97  }
```

Listing B.5: The loadVersion() function loads a specific changeNode from the patch history when a player selects it in the DAG, restoring the associated synth configuration and patch history sequencer state using the provided branch name and hash.

B.6 updateSynthWorklet()

```
function updateSynthWorklet(cmd, data, structure, changeNode){

    switch (cmd) {
        case 'setOutputVolume':
            synthWorklet.port.postMessage({
                cmd: 'setOutputVolume',
                data: data
            });
        break
        case 'clearGraph':
            synthWorklet.port.postMessage({
                cmd: 'clearGraph'
            });
        break
        case 'loadVersion':
            synthWorklet.port.postMessage({
                cmd: 'loadVersion',
                data: data,
            });
        break

        case 'setSignalAnalysis':
            synthWorklet.port.postMessage({
                cmd: 'setSignalAnalysis',
                data: data
            });
        break

        case 'addNode':
            synthWorklet.port.postMessage({
                cmd: 'addNode',
                data: data,
                structure: structure
            });
        break

        case 'removeNode':
            synthWorklet.port.postMessage({
                cmd: 'removeNode',
                data: data
            });
        break

        case 'addCable':
            synthWorklet.port.postMessage({
                cmd: 'addCable',
                data: data
            });

```

```
50          break
51
52          case 'removeCable':
53              synthWorklet.port.postMessage({
54                  cmd: 'removeCable',
55                  data: data
56              });
57          break
58
59          case 'paramChange':
60
61              synthWorklet.port.postMessage({ cmd: 'paramChange', data
                    : data });
62          break
63      }
64
65  }
```

Listing B.6: Sends commands from the UI to the synthWorklet audio engine via the Web Audio API's message port. Supports graph updates (e.g., adding/removing nodes or cables), parameter changes, and signal analysis configuration. Commands are structured and dispatched using a switch statement.

B.7 updateCytoscapeFromDocument()

```
1  // Function to update Cytoscape with the state from the loaded
       changeNode doc (referred to here as 'forkedDoc')
2  function updateCytoscapeFromDocument(forkedDoc, cmd,
       lastGestureValue) {
3      App.synth.visual.modules = forkedDoc.synth.graph.modules
4      let elements = forkedDoc.elements
5      peers.remote = {}
6      // only rebuild the UI if needed
7      if(cmd === 'buildUI'){
8          parentNodePositions = []; // Array to store positions of all
               parent nodes
9
10         // Step 1: Extract all parent nodes from the given document
11         const parentNodes = forkedDoc.elements.filter(el => el.
               classes === ':parent'); // Adjust based on your schema
12         parentNodes.forEach(parentNode => {
13             if (parentNode.position) {
14                 parentNodePositions.push({
15                     id: parentNode.data.id,
16                     position: parentNode.position
17                 });
18             }
19         })
```

```
20
21          // Clear existing elements from Cytoscape instance
22          synthGraphCytoscape.elements().remove();
23
24          // remove all dynamicly generated UI overlays (knobs, umenus
                , etc)
25          removeUIOverlay('allNodes')
26
27          // ensure their container divs are removed too
28          clearparamContainerDivs()
29
30          // I do this from the synthFile because the parentNodes'
                dimensions respond to their childs' positioning
31          synthGraphCytoscape.json(patchHistory.synthFile)
32          // synthGraphCytoscape.nodes(':parent').forEach(n => console
                .log(n.id())); // lock all parent nodes so they can't be
                dragged
33          // Sync the positions in `elements`
34          const syncedElements = syncPositions(forkedDoc);
35          // add all cables back in with a check to make sure we don't
                render edges to empty parent nodes
36          for (let i = 0; i < syncedElements.length; i++) {
37              const el = syncedElements[i];
38              if (el.type === 'edge') {
39                  const sourceExists = synthGraphCytoscape.
                        getElementById(el.data.source).length > 0;
40                  const targetExists = synthGraphCytoscape.
                        getElementById(el.data.target).length > 0;
41
42                  if (!sourceExists || !targetExists) {
43                      console.warn(`Skipping edge: ${el.data.id} due
                            to missing source or target`);
44                      continue; // Skip this iteration and move to the
                            next element
45                  }
46              }
47
48              synthGraphCytoscape.add(el);
49          }
50
51          let index = 0
52          elements.forEach((node)=>{
53              // set module grabbable to false -- prevents module
                    movements in main view
54              if(node.classes === ':parent'){
55                  // lock the module's position
56                  synthGraphCytoscape.getElementById(node.data.id).
                        lock();
57              }
58              if(node.classes === 'paramAnchorNode'){
```

```
59                  let value = App.synth.visual.modules[node.data.
                        parent].params[node.data.label]
60                  createFloatingOverlay(node.data.parent, node, index,
                        value)
61                  index++
62              }
63          })
64          // Initial position and scale update. delay it to wait for
                cytoscape rendering to complete.
65          setTimeout(() => {
66              updateKnobPositionAndScale('all');
67          }, 10); // Wait for the current rendering cycle to complete
68
69          // After loading your synth and creating overlays:
70          cacheVisibleParamControls(App.synth.visual.modules);
71
72      } else if (cmd === 'buildFromSyncMessage'){
73          // Sync the positions in `elements`
74          const syncedElements = syncPositions(forkedDoc);
75          // clear
76          synthGraphCytoscape.elements().remove();
77
78          // 3. Add new elements to Cytoscape
79          synthGraphCytoscape.add(syncedElements)
80
81          // check to see if none of the overlays were made. this is
                the case if peer has a blank document and is syncing to
                another peer's doc
82          const overlaysExist = document.querySelector('[id^="
                paramControl_parent:"]') !== null;
83
84          if (!overlaysExist) {
85              updateCytoscapeFromDocument(forkedDoc, 'buildUI');
86              return; // skip the rest, since buildUI handles
                    everything
87          }
88
89          refreshParamControls(App.synth.visual.modules,
                lastGestureValue);
90      }
91
92
93      else {
94          // Sync the positions in `elements`
95          const syncedElements = syncPositions(forkedDoc);
96          // clear
97          synthGraphCytoscape.elements().remove();
98
99          // 3. Add new elements to Cytoscape
100         synthGraphCytoscape.add(syncedElements)
101
```

```
102             refreshParamControls(App.synth.visual.modules);
103         }
104     }
```

Listing B.7: The updateCytoscapeFromDocument() function rebuilds the visual patch graph in Cytoscape.js based on a loaded changeNode document. It supports three modes: buildUI is triggered after loading a .fpsynth synthesizer file and fully reconstructs the interface, including parameter overlays. The default mode responds to changes or version loads initiated by the local player, while buildFromSyncMessage applies state changes received from a remote peer. Since rebuilding UI overlays involves additional computation, the latter two modes skip this step unless overlays are missing and must be regenerated.

B.8 createMerge()

```
1  // merge 2 versions & create a new node in the history graph
2  function createMerge(nodes){
3      let doc1 = nodes[0]
4      let doc2 = nodes[1]
5
6      // load historical views of both docs
7      let requestedDoc1 = loadAutomergeDoc(doc1.branch)
8      let requestedDoc2 = loadAutomergeDoc(doc2.branch)
9      // merge them
10     let mergedDoc = Automerge.merge(requestedDoc1, requestedDoc2)
11     let hashes = [ doc1.id, doc2.id ]
12
13     // create empty change to 'flatten' the merged Doc
14     currentBranch = Automerge.emptyChange(mergedDoc);
15
16     let hash = Automerge.getHeads(currentBranch)[0]
17     // make new branch name
18     const newBranchName = uuidv7()
19     // update the patchHistory in Automerge
20     patchHistory = Automerge.change(patchHistory, { message: `merge
           parents: ${doc1.id} ${doc2.id} `}, (patchHistory) => {
21         // Initialize the branch patchHistorydata if it doesn't
               already exist
22         if (!patchHistory.branches[newBranchName]) {
23             patchHistory.branches[newBranchName] = {
24                 head: null,
25                 parent: [ doc1.id, doc2.id ],
26                 history: []
27             };
28         }
29         // Update the head property
30         patchHistory.branches[newBranchName].head = hash;
31         // Push the new history entry into the existing array
```

```
32          patchHistory.branches[newBranchName].history.push({
33              hash: hash,
34              msg: 'merge',
35              parent: hashes,
36              nodes: [doc1, doc2]
37          });
38          // store current doc
39          patchHistory.docs[newBranchName] = Automerge.save(
                currentBranch)
40          // store the HEAD info
41          patchHistory.head.hash = hash
42          patchHistory.head.branch = newBranchName
43          // store the branch name so that we can ensure its ordering
                later on
44          patchHistory.branchOrder.push(newBranchName)
45      });
46      // set docUpdated so that indexedDB will save it
47      docUpdated = true
48      // update the audio graph
49      updateSynthWorklet('loadVersion', currentBranch.synth.graph)
50      // update the synth graph (visual)
51      updateCytoscapeFromDocument(currentBranch, 'buildUI');
52      // update the historyGraph
53      reDrawHistoryGraph()
54  }
```

Listing B.8: The createMerge function takes two changeNodes selected by the player and merges them. A new branch is created to store the result of the merge. The unified patch state is then loaded into the main application and rendered in the visual and audio synth graphs.

B.9 startCable()

```
1  function startCable(source, position){
2      temporaryCables.local.source = source;
3      const mousePos = position;
4
5      // Get the ghostCable property from the temporaryCables.local.
           ghostNode, default to 'ellipse' if undefined
6      const ghostShape =  temporaryCables.local.source.data('
           ghostCableShape') || 'ellipse';
7      const ghostColour =  temporaryCables.local.source.data('
           ghostCableColour') || '#5C9AE3';
8
9      // Create a ghost node at the mouse position to act as the
           moving endpoint
10     temporaryCables.local.ghostNode = synthGraphCytoscape.add({
11         group: 'nodes',
```

```
        data: { id: 'localGhostNode' },
        position: mousePos,
        grabbable: false, // Ghost node shouldn't be draggable
        classes: 'ghostNode'
    });

    // Apply the ghostShape to the ghostNode using a direct style override
    temporaryCables.local.ghostNode.style({
        'shape': ghostShape,
        'background-color': ghostColour
    });
    // Create a temporary edge from ghostNodes.local.source to temporaryCables.local.ghostNode
    temporaryCables.local.tempEdge = synthGraphCytoscape.add({
        group: 'edges',
        data: { id: 'localTempEdge', source: temporaryCables.local.source.id(), target: 'localGhostNode' },
        classes: 'tempEdge'
    });

    // Set target endpoint to mouse position initially
    temporaryCables.local.tempEdge.style({ 'line-color': '#FFA500', 'line-style': 'dashed',  'source-arrow-shape': 'none'  }); // Set temporary edge color
}
```

Listing B.9: Function startCable() creates a temporary visual connection between a source node and the mouse position. A ghost node and dashed cable are rendered to indicate an in-progress patching action.

B.10 handleRemoteCables()

```
function handleRemoteCables(cmd,  peerID, sourceID, position){
    switch(cmd){
        case 'startRemoteGhostCable':
            let ghostId = `ghostNode-${peerID}`
            let tempEdgeID = `tempEdge-${peerID}`

            temporaryCables.peers[peerID] = {
                source: synthGraphCytoscape.getElementById(sourceID),
                // Create a ghost node at the mouse position to act as the moving endpoint
                ghostNode: synthGraphCytoscape.add({
                    group: 'nodes',
                    data: { id: ghostId },
                    position: position,
```

```
14                      grabbable: false, // Ghost node shouldn't be
                            draggable
15                      classes: 'ghostNode'
16                  }),
17                  // Create a temporary edge from temporaryCables.
                        local.source to ghostNode
18                  tempEdge: synthGraphCytoscape.add({
19                      group: 'edges',
20                      data: { id: tempEdgeID, source: sourceID, target
                            : ghostId },
21                      classes: 'tempEdge'
22                  }),
23                  targetNode: null
24              }
25
26              // set ghostNode Style according to opposite of source
                    Node
27              const ghostShape = temporaryCables.peers[peerID].source.
                    data('ghostCableShape') || 'ellipse';
28              const ghostColour = temporaryCables.peers[peerID].source
                    .data('ghostCableColour') || '#5C9AE3';
29              temporaryCables.peers[peerID].ghostNode.style({
30                  'shape': ghostShape,
31                  'background-color': ghostColour
32              });
33
34              temporaryCables.peers[peerID].tempEdge.style({ 'line-
                    color': '#228B22', 'line-style': 'dashed', 'source-
                    arrow-shape': 'none' }); // Set peer temporary edge
                    color
35          break;
36
37          case 'updateRemoteGhostCable':
38              // update the tmporary cable's position
39              temporaryCables.peers[peerID].ghostNode.position(
                    position)
40          break
41
42          case 'finishRemoteGhostCable':
43              // remove the tempEdge
44              synthGraphCytoscape.remove(temporaryCables.peers[peerID
                    ].tempEdge)
45              // remove remote ghostnode
46              synthGraphCytoscape.remove(temporaryCables.peers[peerID
                    ].ghostNode);
47              synthGraphCytoscape.nodes().removeClass('highlighted');
48
49              // remove remote peer's temporary cable from object
50              delete temporaryCables.peers[peerID]
51
52          break;
```

```
53      }
54  }
```

Listing B.10: Handles the rendering of temporary cables created by a remote peer, updating their position in real time. These updates are transmitted over a WebRTC data channel and are intentionally kept outside the Automerge sync protocol to avoid polluting the version history.

B.11 isEdgeInCycle()

```
1   function isEdgeInCycle(edgeToCheck) {
2       const parentEdges = new Map();
3       const selfLoopEdges = []; // Store self-loop edges for same
            parent connections
4
5       // Step 1: Build a directed graph of parent nodes with child
            information
6       synthGraphCytoscape.edges().forEach((edge) => {
7           const sourceParent = edge.data().source.split('.')[0]; //
                Parent of the source node
8           const targetParent = edge.data().target.split('.')[0]; //
                Parent of the target node
9           const sourceChild = edge.data().source.split('.')[1]; //
                Source child
10          const targetChild = edge.data().target.split('.')[1]; //
                Target child
11
12          if (targetParent.includes('AudioDestination')) {
13              return; // Ignore connections to the audio destination
14          }
15
16          if (sourceParent === targetParent) {
17              // Detect self-loop: connections within the same parent
                    node
18              selfLoopEdges.push({
19                  sourceParent,
20                  targetParent,
21                  sourceChild,
22                  targetChild,
23              });
24          } else {
25              // Add normal inter-parent edges
26              if (!parentEdges.has(sourceParent)) {
27                  parentEdges.set(sourceParent, []);
28              }
29              parentEdges.get(sourceParent).push({
30                  targetParent,
31                  sourceChild,
```

```
32                  targetChild,
33              });
34          }
35      });
36
37      // Step 2: Detect self-loop cycles
38      const selfLoopMatch = selfLoopEdges.some(
39          (edge) =>
40              edge.sourceParent === edgeToCheck.data().source.split('.
                    ')[0] &&
41              edge.targetParent === edgeToCheck.data().target.split('.
                    ')[0] &&
42              edge.sourceChild === edgeToCheck.data().source.split('.'
                    )[1] &&
43              edge.targetChild === edgeToCheck.data().target.split('.'
                    )[1]
44      );
45
46      if (selfLoopMatch) {
47          // console.log(
48          //     `Edge ${edgeToCheck.id()} forms a self-loop cycle
                    within ${edgeToCheck.data().source.split('.')[0]}`
49          // );
50          return true;
51      }
52
53      // Step 3: Detect inter-parent cycles using Depth-First Search (
            DFS)
54      const visited = new Set();
55      const recStack = new Set();
56      let edgeInCycle = false;
57
58      function dfs(node, path = [], edgePath = []) {
59          if (!visited.has(node)) {
60              visited.add(node); // Mark the node as visited
61              recStack.add(node); // Add the node to the recursion
                    stack
62
63              const neighbors = parentEdges.get(node) || [];
64              for (const neighbor of neighbors) {
65                  const currentEdge = {
66                      sourceParent: node,
67                      targetParent: neighbor.targetParent,
68                      sourceChild: neighbor.sourceChild,
69                      targetChild: neighbor.targetChild,
70                  };
71
72                  if (!visited.has(neighbor.targetParent)) {
73                      if (
74                          dfs(
75                              neighbor.targetParent,
```

```
76                              [...path, node],
77                              [...edgePath, currentEdge]
78                          )
79                      ) {
80                          return true; // Cycle detected
81                      }
82                  } else if (recStack.has(neighbor.targetParent)) {
83                      // Cycle detected: back edge
84                      const cycleStartIndex = path.indexOf(neighbor.
                            targetParent);
85                      const cycleEdges = [
86                          ...edgePath.slice(cycleStartIndex),
87                          currentEdge,
88                      ]; // Edges in the cycle
89
90                      // Check if the edgeToCheck belongs to this
                            cycle
91                      if (
92                          cycleEdges.some(
93                              (edge) =>
94                                  edge.sourceParent ===
95                                      edgeToCheck.data().source.split(
                                            '.')[0] &&
96                                  edge.targetParent ===
97                                      edgeToCheck.data().target.split(
                                            '.')[0] &&
98                                  edge.sourceChild ===
99                                      edgeToCheck.data().source.split(
                                            '.')[1] &&
100                                 edge.targetChild ===
101                                     edgeToCheck.data().target.split(
                                            '.')[1]
102                         )
103                     ) {
104                         edgeInCycle = true;
105                         return true;
106                     }
107                 }
108             }
109         }
110
111         recStack.delete(node);
112         return false;
113     }
114
115     for (const node of parentEdges.keys()) {
116         if (!visited.has(node)) {
117             dfs(node);
118             if (edgeInCycle) break; // Exit early if the edge is
                    found in a cycle
119         }
```

```
120     }
121
122     return edgeInCycle; // Return true if the edge belongs to any
            cycle
123 }
```

Listing B.11: Determines whether a given cable (edge) introduces a cycle in the patch graph rendered by Cytoscape. The function checks for both self-loop cycles and inter-node cycles using depth-first search (DFS). If a cycle is detected, it inserts a feedbackDelayNode with a fixed delay of 128 samples into the audio graph to allow feedback patching.

Complete File — The full, commented source is archived at https://github.com/michaelpalumbo/forkingpaths/blob/dissertationJune/src/scripts/synthApp.js

Appendix C

Code Excerpts from the Patch History Window

A selection of notable code listings from `patchHistory.js`, the script for the Patch History Window. Each excerpt is documented in this appendix, with its corresponding location in `patchHistory.js` indicated in the Lines column below.

Table C.1: Code excerpts from the Patch History Window

Listing	Excerpt	Purpose	Lines
C.1	playGesture()	Gesture playback from changeNode	2370-2459
C.2	playGestureFromSequencerStep()	Gesture playback from sequencer	2022-2067
C.3	MIDI CC Input	Map MIDI CC to graph traversal	283-315
C.4	updateStepRow()	Assign a changeNode to a step	583-660
C.5	Monophonic Playback	Steps recalled one at a time	1697-1782
C.6	Polyphonic Playback	Each step becomes its own loop	1787-1848
C.7	calculateCentrality()	Map centrality onto step length	1960 - 1999
C.8	calculateEuclideanDistance()	Map distance onto step length	1915-1959

C.1 playGesture()

```
1  function playGesture(mode) {
2      gestureCy.nodes().removeClass('highlighted')
3      // 'repeat' is passed by the function call when looping is on,
          so we don't want to have to get the same data again if the
          loop is on
4      if(mode != 'repeat'){
5          // reset the gesture scheduler
```

```
6           gestureData.scheduler = [ ]
7       }
8
9       // create the scheduler
10      gestureData.nodes.forEach((node) => {
11          const delay = node.data.timestamp - gestureData.startTime;
                // Calculate delay from the start
12
13          // Use setTimeout to schedule the callback
14          const timeoutID = setTimeout(() => {
15
16              // highlight the node
17              let hNode = gestureCy.getElementById(node.data.id);
18              hNode.addClass('highlighted');
19
20              if(gestureData.assign.param === 'default'){
21
22                  let data = {
23                      parent: node.data.parents,
24                      param: node.data.param,
25                      value: node.data.value
26                  }
27                  sendToMainApp({
28                      cmd: 'playGesture',
29                      data: data,
30                      kind: 'n/a'
31                  })
32
33
34              } else {
35                  // process it using the gesturedata assign range
                        data for scaling
36
37                  // convert the value from the source value's min and
                         max to gestureData.assign.range
38                  // first get the min and max of the source value
39                  // synthParamRanges
40                  let value = node.data.value
41
42                  let storedParam = patchHistory.synthFile.audioGraph.
                        modules[node.data.parents].moduleSpec.parameters[
                        node.data.param]
43                  let targetParam = gestureData.assign
44
45                  sendToMainApp({
46                      cmd: 'playGesture',
47                      data: convertParams(storedParam, targetParam,
                            value),
48                      kind: targetParam.kind
49
50                  })
```

```
51
52              }
53
54              if(UI.gestureEditor.loop.checked && gestureData.length
                    === delay){
55                  playGesture('repeat')
56                  // setTimeout(() => {
57                  //     playGesture('repeat')
58                  // }, 250);
59              }
60          }, delay);
61
62
63
64          gestureData.scheduler.push(timeoutID)
65      });
66
67      // get the end of the gesture
68      const maxDelay = Math.max(...gestureData.nodes.map(node => node.
            data.timestamp - gestureData.startTime));
69
70      const finalTimeoutID = setTimeout(() => {
71          gestureCy.elements().removeClass('highlighted');
72
73          // if looping is off, set the stop button back to 'play'
74          if (!UI.gestureEditor.loop.checked) {
75              UI.gestureEditor.play.textContent = 'Play'
76          }
77      }, maxDelay + 1); // +1 to ensure it's last
78
79      gestureData.scheduler.push(finalTimeoutID);
80
81
82
83  }
```

Listing C.1: Plays a gesture when the player either selects its associated changeNode, or from within the Gesture Editor

C.2 playGestureFromSequencerStep()

```
1  // playback a stored gesture from a sequencer step
2  function playGestureFromSequencerStep(gesture, stepLength){
3
4      let quantizedGesture = quantizeGesture(gesture, stepLength)
5
6      // create the scheduler
7      quantizedGesture.forEach((node) => {
```

```
8           const delay = node.t * 1000; // (convert to milliseconds)
9
10          // Use setTimeout to schedule the callback
11          const timeoutID = setTimeout(() => {
12              if(gesture.assign.param === 'default'){
13
14                  let data = {
15                      parent: node.parent,
16                      param: node.param,
17                      value: node.value
18                  }
19
20                  sendToMainApp({
21                      cmd: 'playGesture',
22                      data: data,
23                      kind: 'n/a'
24                  })
25
26
27              } else {
28
29                  let value = node.value
30
31                  let storedParam = patchHistory.synthFile.audioGraph.
                        modules[node.parent].moduleSpec.parameters[node.
                        param]
32                  let targetParam = gesture.assign
33
34                  sendToMainApp({
35                      cmd: 'playGesture',
36                      data: convertParams(storedParam, targetParam,
                            value),
37                      kind: targetParam.kind
38
39                  })
40
41              }
42
43          }, delay);
44
45          gesture.scheduler.push(timeoutID)
46      });
47  }
48
49  function quantizeGesture(gesture, stepLength) {
50      const duration = gesture.endTime - gesture.startTime;
51      const scale = stepLength / duration;
52
53      // Map each point's timestamp to the new interval [0, stepLength
            ]
54      return gesture.gesturePoints.map(point => ({
```

```
55          ...point,
56          t: (point.timestamp - gesture.startTime) * scale
57      }));
58  }
```

Listing C.2: This function plays a gesture when triggered from a sequencer step. It uses the quantizeGesture function to ensure that it plays back entirely within the current step length.

C.3 MIDI CC Input

```
1  // Listen to control change (knobs/faders usually send these)
2  midiInput.addListener("controlchange", (e) => {
3      console.log(`Control Change on CC\#${e.controller.number}: ${e.
           value}`);
4      // cycle through graph
5      if (e.controller.number === 8) {
6
7          if(!midiValues.controllers[e.controller.number]){
8              midiValues.controllers[e.controller.number] = {value:
                   null}
9          }
10
11         const scaled = scaleMidiValue(e.rawValue, 0, 127, 0,
               historyGraphNodesArray.length - 1);
12         if(midiValues.controllers[e.controller.number].value !=
               scaled){
13
14             let n = historyGraphNodesArray[scaled].data
15             loadVersion(n.id, n.branch)
16
17             let historyNode = historyDAG_cy.getElementById(n.id)
18             highlightNode(historyNode)
19
20             midiValues.controllers[e.controller.number].value =
                   scaled
21         }
22     }
23   });
24 })
25 .catch((err) => console.error("WebMidi could not be enabled:", err))
       ;
26
27 function scaleMidiValue(input, inMin, inMax, outMin, outMax) {
28     return Math.round(((input - inMin) / (inMax - inMin)) * (outMax
           - outMin) + outMin);
```

```
}
```

Listing C.3: Code for handling MIDI CC input to navigate the patch history graph. CC values from 0 to 127 are mapped onto a depth-first traversal of available changeNodes, allowing external MIDI controllers to trigger version changes during performance.

C.4 Set Sequencer Step

```
function updateStepRow(index, nodeData, gestureData = null,
    stepLength, fromRemote = false) {
     if(!fromRemote){

          sendToMainApp({
              cmd: 'syncPeerSequencer',
              action: 'updateStepRow',
              payload: {
                  index: index,
                  nodeData: nodeData,
                  gestureData: gestureData,
                  stepLength, stepLength
              }
          })
     }


     const row = document.querySelectorAll("#dynamicTableBody2 tr")[
         index];
     if (!row) return;

     const changeCell = row.cells[0];
     const stepLabelCell = row.cells[1];
     const stepLengthSelect = row.cells[2].querySelector("select");

     // Set background color based on change node label
     const labelKey = nodeData.label.split(" ")[0];
     changeCell.style.backgroundColor = docHistoryGraphStyling.
         nodeColours[labelKey] || "#888";

     // Set label text and dataset info
     const label = nodeData.label;
     stepLabelCell.textContent = label;

     row.dataset.id = nodeData.id;
     row.dataset.label = label;
     row.dataset.branch = nodeData.branch;

     // Handle gesture metadata
     if (label.startsWith("gesture")) {
```

```
38          row.dataset.gesture = true;
39          row.dataset.gestureData = JSON.stringify(gestureData);
40
41          // hydrate the sequencerData
42          sequencerData.gestures[index] = gestureData
43      }
44
45      if (nodeData.gestureDataPoint) {
46          const abrv = `${nodeData.parents.split('_')[0]}_${nodeData.
                parents.split('_')[1]}`;
47          const gestureLabel = `gesturePoint: ${abrv}:${nodeData.param
                }:${nodeData.value}`;
48          stepLabelCell.textContent = gestureLabel;
49
50          row.dataset.label = gestureLabel;
51          row.dataset.isGestureDataPoint = true;
52          row.dataset.gestureDataPointValue = nodeData.value;
53          row.dataset.param = nodeData.param;
54          row.dataset.parent = nodeData.parents;
55          row.dataset.id = nodeData.historyID; // override with
                gesture node ID
56      }
57
58      // handle sequence change nodes (where we set a previous
            sequence into the sequencer step)
59      if(nodeData.sequencerTable){
60          row.dataset.sequencerTable = JSON.stringify(nodeData.
                sequencerTable)
61      }
62
63      if(stepLength){
64          stepLengthSelect.value = stepLength
65      }
66
67      setGestureSaveButtonState(false)
68
69      if(sequencerData.settings.modes.stepLengthFunction === '
            euclideanDistance'){
70          calculateEuclideanDistances()
71          // that function calls saveSequencerTable() afterwards
                already
72      } else {
73          saveSequencerTable(); // Save the new state immediately
74      }
75
76  }
```

Listing C.4: The updateStepRow() function updates a single row in the Patch History Sequencer, setting metadata, color, and label information based on the selected changeNode.

C.5 Patch History Sequencer: Monophonic Playback

```
1  // Set the initial BPM
2  transport.bpm.value = 120;
3
4  // MONOPHONIC SEQUENCER (Lock Step Operation through steps 1-8
       linearly)
5  let currentStepIndex = 0; // Tracks the current step in the table
6
7  const loop = new Tone.Loop(function(time){
8      // Set current step's duration immediately
9      const stepLength2 = storedSequencerTable[currentStepIndex].
           stepLength;
10
11     const stepDuration = Tone.Time(stepLength2).toSeconds();
12
13     let stepsToAdvance = 1 // this could be used to randomize steps
           too. (i.e. if random is selected, randomize this value at
           each loop)
14
15     // Get the current step
16     const currentStep = storedSequencerTable[currentStepIndex];
17
18     let nextIndex = (currentStepIndex + stepsToAdvance) %
           storedSequencerTable.length;
19     let nextStep = storedSequencerTable[nextIndex];
20
21
22     // Count how many future steps should be skipped
23     while (sequencerData.settings.modes.emptyStep === 'skip' &&
           nextStep && nextStep.status === 'Inactive') {
24         stepsToAdvance++;
25         nextIndex = (currentStepIndex + stepsToAdvance) %
               storedSequencerTable.length;
26         nextStep = storedSequencerTable[nextIndex];
27     }
28
29     // Highlight the current step in the table
30     const tableRows = document.querySelectorAll("#dynamicTableBody2
           tr");
31     tableRows.forEach((row) => row.classList.remove("table-active"))
           ;
32     const targetRow = tableRows[currentStepIndex];
33
34     if (targetRow) targetRow.classList.add("table-active");
35
36     // if step is active, send request to load the version
37     if (currentStep.status == "Active"){
38
39         // first check if we're loading a gesture point (a single
               knob position within a gesture)
```

```
40          if(targetRow.dataset.isGestureDataPoint){
41              let dataPoint = {
42                  parent: targetRow.dataset.parent,
43                  param: targetRow.dataset.param,
44                  value: targetRow.dataset.gestureDataPointValue
45              }
46
47              // it's a special form of loadVersion, where we want to
                    load the version, but ensure that the associated
                    gesture point value is loaded
48              loadVersionWithGestureDataPoint(currentStep.node.id,
                    currentStep.node.branch, dataPoint)
49
50          }
51          else if (targetRow.dataset.sequencerTable) {
52              const embeddedSeq = JSON.parse(targetRow.dataset.
                    sequencerTable);
53              const totalSubsteps = embeddedSeq.length;
54              const outerStepDuration = stepDuration; // duration of
                    current step
55              const subStepDuration = outerStepDuration /
                    totalSubsteps;
56              const embeddedEvents = embeddedSeq.map((row, i) => {
57                  return [i * subStepDuration, () => {
58                    if (row.status === "Active") {
59                      if (row.isGestureDataPoint) {
60                        const dataPoint = {
61                          parent: row.parent,
62                          param: row.param,
63                          value: row.value
64                        };
65                        loadVersionWithGestureDataPoint(row.node.id,
                              row.node.branch, dataPoint, true);
66                      } else {
67                        loadVersion(row.node.id, row.node.branch, null
                              , true);
68
69                        if (row.stepChange?.startsWith("gesture") &&
                              row.gestureData) {
70
71                          playGestureFromSequencerStep(sequencerData.
                                gestures[currentStepIndex], `${
                                subStepDuration}s`);
72                        }
73                      }
74                    }
75                  }];
76              });
77
78              const embeddedPart = new Tone.Part((t, eventCallback) =>
                    {
```

```
79
80                  eventCallback(t);
81              }, embeddedEvents);
82
83              embeddedPart.start(time); // starts at the same moment
                    the outer step begins
84
85              // Optional cleanup
86              transport.scheduleOnce(() => {
87                  embeddedPart.dispose(); // or .stop() if you want to
                        reuse
88              }, time + outerStepDuration);
89
90          }
91
92          else {
93              // load the version
94              loadVersion(currentStep.node.id, currentStep.node.branch
                    , null, true)
95
96              if(targetRow.dataset.gesture){
                    playGestureFromSequencerStep(sequencerData.gestures[
                    currentStepIndex], stepDuration)
97              }
98          }
99
100         let historyNode = historyDAG_cy.getElementById(currentStep.
                node.id)
101         highlightNode(historyNode)
102     } else {
103         switch(sequencerData.settings.modes.emptyStep){
104             case 'passThrough':
105                 // do nothing, let previous step's value continue
106             break
107
108             case 'blank':
109                 let blankPatch = firstNode.data()
110                 highlightNode(firstNode)
111                 loadVersion(blankPatch.id, blankPatch.branch)
112             break
113         }
114     }
115     // randomize the next step
116     if(sequencerData.settings.modes.order === 'random'){
117         // if skip mode is activated for empty steps
118         if(sequencerData.settings.modes.emptyStep === 'skip'){
119             // special case. only skip to a step that is active
120             // get all indices of active steps
121             const activeStepIndices = storedSequencerTable
122                 .map((step, index) => step.status === 'Active' ?
                        index : null)
```

```
                .filter(index => index !== null);
            // select a random step only from active steps
            currentStepIndex = activeStepIndices[Math.floor(Math.
                random() * activeStepIndices.length)];
        } else {
            // randomize the the next step from the entire table
            currentStepIndex = Math.floor(Math.random() *
                storedSequencerTable.length);
        }
    } else {
        // normal mode, advance or skip to next step
        currentStepIndex = (currentStepIndex + stepsToAdvance) %
            storedSequencerTable.length;
    }
    // update the next step's length
    loop.interval = stepLength2;
}, sequencerData.settings.stepLength)
```

Listing C.5: The patch history sequencer is based on a Tone.js Loop that iterates through the storedSequencerTable to load historical patch states. Step transitions are determined by user-defined modes (e.g., skip, blank, random), and gesture playback is synchronized with each step's duration.

C.6 Patch History Sequencer: Polyphonic Playback

```
function startPolyphonicSequencer() {
    stopPolyphonicSequencer(); // clear any previous ones

    const tableRows = document.querySelectorAll("#dynamicTableBody2
        tr");

    tableRows.forEach((row, index) => {
        const step = storedSequencerTable[index];
        const loop = new Tone.Loop((time) => {
            if (step.status === "Active") {
            const targetRow = tableRows[index];
            targetRow.classList.add("table-active");
            setTimeout(() => targetRow.classList.remove("table-
                active"), 100);

            if (targetRow.dataset.isGestureDataPoint) {
                const dataPoint = {
                parent: targetRow.dataset.parent,
                param: targetRow.dataset.param,
                value: targetRow.dataset.gestureDataPointValue
                };
                loadVersionWithGestureDataPoint(step.node.id, step.
                    node.branch, dataPoint);
```

```
21              } else {
22                  loadVersion(step.node.id, step.node.branch);
23                  if (targetRow.dataset.gesture) {
24                  playGestureFromSequencerStep(sequencerData.gestures[
                        index], step.stepLength);
25                  }
26              }
27              }
28          }, step.stepLength);
29
30          loop.start(0);
31          polyphonicLoops.push(loop);
32      });
33  }
34
35  function stopPolyphonicSequencer() {
36      polyphonicLoops.forEach((loop) => loop.dispose());
37      polyphonicLoops = [];
38      loop.stop(); // also stop mono loop, just in case
39  }
40
41  function setStepLengthFunction(func){
42      sequencerData.settings.modes.stepLengthFunction = func
43          // Perform actions based on the selected value
44      if (func === "setAllTo4n") {
45          document.querySelectorAll(".step-length").forEach((select, i
                ) => {
46              select.value = '4n'
47
48
49          })
50          setSequencerSaveButtonState(false)
51          saveSequencerTable();
52      } else if (func === "userEditable") {
53          // Add logic for user-editable step length
54      } else if (func === "closenessCentrality") {
55          calculateDistancesFromTableRows()
56      }
57      else if (func === "euclideanDistance") {
58          calculateEuclideanDistances()
59      }
60  }
```

Listing C.6: The polyphonic sequencer triggers each active step in parallel using independent Tone.Loop instances. Each step can recall a patch version allowing overlapping playback.

C.7 Map Centrality onto Step Lengths

```
function calculateCentrality() {
    if(!storedSequencerTable){
        return
    }

    const rows = UI.sequencer.table.body.querySelectorAll("tr");

    for (let i = 0; i < storedSequencerTable.length - 1; i++) {
        const currentNodeID = storedSequencerTable[i].node.id;
        // Get the ID of the next node (circular for the last row)
        const nextNodeID = i < storedSequencerTable.length - 1
        ? storedSequencerTable[i + 1].node.id // Next row for all except last
        : storedSequencerTable[0].node.id;    // First row for the last row


        // Compute the shortest path distance using Dijkstra
        const dijkstra = historyDAG_cy.elements().dijkstra({
            root: historyDAG_cy.$(`#${currentNodeID}`), // Current node
            directed: true // Set to false if the graph is undirected
        });

        const distance = dijkstra.distanceTo(historyDAG_cy.$(`#${nextNodeID}`));

        // Update the 2nd column (Step Length) of the current row
        const stepLengthCell = rows[i].children[1]; // 2nd cell of the current row
        stepLengthCell.textContent = isFinite(distance) ? distance.toFixed(2) : "No Path";

        if(stepLengthCell.textContent === 'No Path'){
            // set the active step setting to 'skip'
            rows[i].children[2].textContent = 'Inactive'
        }else {
            rows[i].children[2].textContent = 'Active'
        }

    }

        // Handle the last row's Step Length column (no "next" node)
        // rows[storedSequencerTable.length - 1].children[1].textContent = "N/A";
    saveSequencerTable()
```

```
}
```

Listing C.7: The calculateEuclideanDistances() function computes the spatial distance between consecutive active steps in the patch history graph and maps those distances to musical note lengths. This allows the step durations in the sequencer to reflect the visual proximity of nodes in the DAG layout, offering a spatially-informed temporal structure. The function includes basic checks for data availability and prompts the user when Euclidean mode requires at least two consecutive active steps.

C.8 Map Euclidean Distance onto Step Lengths

```
function calculateEuclideanDistances(){
    if(!storedSequencerTable){
        return
    }

    if(!hasActiveNeighbor(storedSequencerTable)){
        showSnackbar('Tip: Euclidean mode requires at least 2
            consecutive active steps.')
    }
    const maxDistance = calculateMaxEuclideanDistance()

    const rows = UI.sequencer.table.body.querySelectorAll("tr");

    for (let i = 0; i < storedSequencerTable.length - 1; i++) {
        if(!storedSequencerTable[i].node || !storedSequencerTable[i
            + 1].node){
            continue
        }
        const currentNodeID = storedSequencerTable[i].node.id;

        // Get the ID of the next node (circular for the last row)
        const nextNodeID = i < storedSequencerTable.length - 1
        ? storedSequencerTable[i + 1].node.id // Next row for all
            except last
        : storedSequencerTable[0].node.id;    // get value of first
            row for the last row's length

        // compute the euclidean distance between 2 nodes
        const currentPosition = historyDAG_cy.$(`#${currentNodeID}`)
            .position();
        const nextPosition = historyDAG_cy.$(`#${nextNodeID}`).
            position();

        let distance = Math.sqrt(
            Math.pow(currentPosition.x - nextPosition.x, 2) +
            Math.pow(currentPosition.y - nextPosition.y, 2)
        );
```

```
32
33
34
35          // Update the 2nd column (Step Length) of the current row
36          const stepLengthCell = rows[i].cells[2].querySelector("
                select"); // stepLength selectmenu
37          // Map a distance value to a corresponding musical note
                length in Tone.js based on a defined range.
38          stepLengthCell.value = mapDistanceToNoteLength(distance.
                toFixed(2), maxDistance)
39          storedSequencerTable[i].stepLength = stepLengthCell.
                textContent
40
41      }
42      saveSequencerTable()
43
44  }
```

Listing C.8: The calculateEuclideanDistances() function computes the spatial distance between consecutive active steps in the patch history graph and maps those distances to musical note lengths. This allows the step durations in the sequencer to reflect the visual proximity of nodes in the DAG layout, offering a spatially-informed temporal structure. The function includes basic checks for data availability and prompts the user when Euclidean mode requires at least two consecutive active steps.

Complete File — The full, commented source is archived at https://github.com/michaelpalumbo/forkingpaths/blob/archive/dissertationJune/src/scripts/patchHistory.js

Appendix D

Server.js

```
1
2
3  import cytoscape from 'cytoscape';
4  import bodyParser from 'body-parser';
5  import dagre from 'cytoscape-dagre';
6  import buildHistoryGraph from './buildHistoryGraph.js';
7
8  import express from 'express';
9  import { createServer} from 'http';
10
11 import { WebSocketServer } from 'ws';
12 import dotenv from 'dotenv';
13 dotenv.config();
14 import { Pool } from 'pg';
15
16
17 // const pool = new Pool({
18 //     connectionString: 'postgresql://localhost:5432/forkingpaths',
19 //     ssl: process.env.NODE_ENV === 'production' ? {
      rejectUnauthorized: false } : false,
20 // });
21
22
23 const pool = new Pool({
24   connectionString:
25     process.env.NODE_ENV === 'production'
26       ? process.env.DATABASE_URL
27       : 'postgresql://localhost:5432/forkingpaths',
28   ssl:
29     process.env.NODE_ENV === 'production'
30       ? { rejectUnauthorized: false }
31       : false,
32 });
33
```

```
34
35 pool.query('SELECT current_database(), current_user, current_schema
       ()')
36   .then(res => console.log('DB Connection:', res.rows[0]))
37   .catch(err => console.error('DB info error:', err));
38
39
40 import patchHistoryRouter from './patchHistoryStorage.js';
41 import synthRouter from './synthFiles.js';
42
43 let peers = {
44
45 }
46
47 let sequencerStates = {}
48
49 let patchHistory;
50 let existingHistoryNodeIDs = new Set()
51
52 let graphStyle = 'MANUAL_DAG'
53 let graphLayouts = {
54     // https://github.com/dagrejs/dagre/wiki#configuring-the-layout
55     DAG: {
56         name: 'dagre',
57         rankDir: 'BT', // Set the graph direction to top-to-bottom
58         nodeSep: 300, // Optional: adjust node separation
59         edgeSep: 100, // Optional: adjust edge separation
60         rankSep: 50, // Optional: adjust rank separation for
               vertical spacing,
61         fit: false,
62         padding: 30
63     },
64     // wrote this specifically to fix the ordering of branches being
           changed on the fly in the dagre package version (above)
65     MANUAL_DAG: {
66         name: 'preset',
67         fit: false,
68         padding: 30,
69         animate: false
70     },
71     breadthfirst: {
72         name: 'breadthfirst',
73
74         fit: false, // whether to fit the viewport to the graph
75         directed: true, // whether the tree is directed downwards (
               or edges can point in any direction if false)
76         padding: 30, // padding on fit
77         circle: false, // put depths in concentric circles if true,
               put depths top down if false
78         grid: false, // whether to create an even grid into which
               the DAG is placed (circle:false only)
```

```
79          spacingFactor: 1.75, // positive spacing factor, larger =>
               more space between nodes (N.B. n/a if causes overlap)
80          boundingBox: undefined, // constrain layout bounds; { x1, y1
               , x2, y2 } or { x1, y1, w, h }
81          avoidOverlap: true, // prevents node overlap, may overflow
               boundingBox if not enough space
82          nodeDimensionsIncludeLabels: false, // Excludes the label
               when calculating node bounding boxes for the layout
               algorithm
83          roots: undefined, // the roots of the trees
84          depthSort: undefined, // a sorting function to order nodes
               at equal depth. e.g. function(a, b){ return a.data('
               weight') - b.data('weight') }
85          animate: false, // whether to transition the node positions
86          animationDuration: 500, // duration of animation in ms if
               enabled
87          animationEasing: undefined, // easing of animation if
               enabled,
88          animateFilter: function ( node, i ){ return true; }, // a
               function that determines whether the node should be
               animated.  All nodes animated by default on animate
               enabled.  Non-animated nodes are positioned immediately
               when the layout starts
89          ready: undefined, // callback on layoutready
90          stop: undefined, // callback on layoutstop
91          transform: function (node, position ){ return position; } //
                transform a given node position. Useful for changing
               flow direction in discrete layouts
92
93      }
94
95
96  }
97
98
99
100
101 // Create Cytoscape instance
102 cytoscape.use( dagre );
103 const historyDAG_cy = cytoscape({
104     headless: true, // Enable headless mode for server-side
           rendering
105
106     // container: document.getElementById('docHistory-cy'),
107     elements: [],
108 //    zoom: parseFloat(localStorage.getItem('docHistoryCy_Zoom')) ||
       1.,
109     // viewport: {
110     //     zoom: parseFloat(localStorage.getItem('docHistoryCy_Zoom
           ')) || 1.
111     // },
```

```
112     boxSelectionEnabled: true,
113     selectionType: "additive",
114     zoomingEnabled: false,
115 
116     layout: graphLayouts[graphStyle],
117     style: [
118         {
119             selector: 'node',
120             style: {
121                 'background-color': 'data(color)', // based on edit
                        type
122                 'label': 'data(label)', // Use the custom label
                        attribute
123                 'width': 30,
124                 'height': 30,
125                 'color': '#000',            // Label text color
126                 'text-valign': 'center',    // Vertically center the
                        label
127                 'text-halign': 'right',      // Horizontally align
                        label to the left of the node
128                 'text-margin-x': 15, //
129                 'text-wrap': 'wrap',
130                 'text-max-width': 120
131                 // 'text-margin-y': 15, // move the label down a
                        little to make space for branch edges
132                 // 'shape': 'data(shape)' // set this for
                        accessibility (colour blindness)
133             }
134 
135         },
136         {
137             selector: 'edge',
138             style: {
139                 'width': 6,
140                 'line-color': '#ccc',
141                 'target-arrow-shape': 'triangle',
142                 'target-arrow-color': '#ccc',
143                 'target-arrow-width': 20, // Size of the target
                        endpoint shape
144                 'curve-style': 'bezier' // Use a Bezier curve to
                        help arrows render more clearly
145 
146             }
147         },
148         {
149             selector: 'node.highlighted',
150             style: {
151                 'border-color': '#228B22', // Highlight color
152                 'border-width': 15,
153                 'shape': 'rectangle'
154             }
```

```
155         },
156         {
157             selector: '.sequencerSelectionBox',
158             style: {
159                 'border-color': 'blue', // Highlight color
160                 'border-width': 4,
161                 'shape': 'rectangle',
162                 'background-color': 'white',
163                 "background-opacity": 0,
164                 "width": 50,
165                 "height": 'data(height)',
166                 "label": '',
167                 "z-index": -1
168 
169             }
170         },
171         {
172             selector: '.sequencerSelectionBox-handle',
173             style: {
174                 // 'border-color': 'blue', // Highlight color
175                 'border-width': 0,
176                 'shape': 'ellipse',
177                 'background-color': 'blue',
178                 // "background-opacity": 0,
179                 "width": '10',
180                 "height": '10',
181                 "label": '',
182                 "z-index": 10
183 
184             }
185         },
186         {
187             selector: '.sequencerNode',
188             style: {
189                 'border-color': '#000000',  // Change to your
                        desired color
190                 'border-width': '8px',
191                 'border-style': 'solid'
192 
193 
194             }
195         },
196         {
197             selector: '.sequencerEdge',
198             style: {
199                 // 'border-color': 'blue', // Highlight color
200                 'line-color': 'blue',
201                 "width": '10',
202                 'target-arrow-color': 'blue'
203 
204 
```

```
205             }
206         },
207     ]
208 });
209
210 // A simple in-memory store for rooms
211 const rooms = {};
212
213 // Helper function to assign a client to a room
214 function assignRoom(ws, desiredRoom) {
215     // If a specific room is provided
216     if (desiredRoom) {
217       // If the desired room already exists
218       if (rooms[desiredRoom]) {
219         // If room is not full, assign the client there.
220         if (rooms[desiredRoom].length < 2) {
221           rooms[desiredRoom].push(ws);
222           return desiredRoom;
223         } else {
224           console.log(`Desired room ${desiredRoom} is full. Falling
                back to default assignment.`);
225         }
226       } else {
227         // Create the room if it doesn't exist.
228         rooms[desiredRoom] = [ws];
229         return desiredRoom;
230       }
231     }
232
233     // Fallback: Loop over existing rooms and join one that has less
          than 2 clients.
234     for (const room in rooms) {
235       if (rooms[room].length < 2) {
236         rooms[room].push(ws);
237         return room;
238       }
239     }
240
241     // If no room is available, create a new one with a default
          naming scheme.
242     const newRoom = `room${Object.keys(rooms).length + 1}`;
243     rooms[newRoom] = [ws];
244     return newRoom;
245   }
246
247 const PORT = process.env.PORT || 3001;
248
249 // Create an Express app (Only for handling basic HTTP requests)
250 const app = express();
251
252 app.use(express.json({ limit: '10mb' })); // for parsing JSON bodies
```

```
253
254 //
255 // app.use('/api/patchHistories', patchHistoryRouter);
256 // app.use('/api/synthFiles', synthRouter);
257
258
259 // Serve static frontend files from Vite's `dist` folder
260 app.use(express.static('dist'));
261
262 // Create an HTTP server and attach WebSocket
263 const server = createServer(app, (req, res)=>{
264     res.writeHead(200);
265     res.end('WebRTC signaling server is running\n');
266 });
267 // Create a WebSocket server that only upgrades `/ws` requests
268 const wss = new WebSocketServer({ noServer: true });
269
270 server.on('upgrade', (request, socket, head) => {
271     console.log('WebSocket upgrade request received');
272     wss.handleUpgrade(request, socket, head, (ws) => {
273       wss.emit('connection', ws, request);
274     });
275   });
276
277
278 let numClients = 0
279 // Handle client connections
280 wss.on('connection', (ws, req) => {
281     numClients++
282
283     const clientIp = req.socket.remoteAddress;
284     console.log(`New connection from ${clientIp}`);
285     console.log(`Number of clients: ${numClients}`)
286
287     // send all synth templates to client:
288     getSynthTemplates(ws);
289
290     // send all patch histories tags
291     getPatchHistories(ws)
292
293     // Handle messages received from clients
294     ws.on('message', (message) => {
295
296         let msg = JSON.parse(message)
297
298         switch(msg.cmd){
299             case 'newPatchHistory':
300
301                 (async () => {
302                     try {
303                         const {
```

```
304                             name,
305                             authors,
306                             description,
307                             modules,
308                             synth_template,
309                             patch_binary,       // should be sent as
                                    base64 from the client
310                             forked_from_id,     // may be null if
                                this is the root
311
312                         } = msg.data;
313
314                         // Decode base64 patch binary from client
315                         const binaryBuffer = Buffer.from(
                               patch_binary, 'base64');
316
317                         const result = await pool.query(
318                             `INSERT INTO patch_histories
319                             (name, authors, description, modules,
                                   synth_template, patch_binary,
                                   forked_from_id, created_at)
320                             VALUES ($1, $2, $3, $4, $5, $6, $7, now
                                   ())
321                             RETURNING id, forked_from_id, tags,
                                   description, name`,
322                             [name, authors, description, modules,
                                   synth_template, binaryBuffer,
                                   forked_from_id]
323                         );
324
325
326                         let message = JSON.stringify({
327                             cmd: 'newPatchHistoryDatabaseID',
328                             success: true,
329                             patchHistoryId: result.rows[0].id,
330                             name: result.rows[0].name,
331                             forked_from_id: result.rows[0].
                                   forked_from_id,
332                             tags: result.rows[0].tags,
333                             description: result.rows[0].description,
334                         })
335
336
337
338
339                         await getUpdatedHistories(ws)
340
341                         ws.send(message);
342
343                     } catch (err) {
```

```
344                         console.error('DB error (savePatchHistory):'
                                , err);
345                         ws.send(JSON.stringify({
346                             cmd: 'savePatchHistoryResponse',
347                             success: false,
348                             error: err.message
349                         }));
350                     }
351                 })();
352 
353             break;
354 
355             case 'newFork':
356 
357               (async () => {
358                 try {
359                     const result = await pool.query(
360                     `INSERT INTO patch_histories (
361                         name,
362                         authors,
363                         description,
364                         tags,
365                         modules,
366                         synth_template,
367                         patch_binary,
368                         forked_from_id,
369                         created_at
370                     )
371                     SELECT
372                         name,
373                         authors,
374                         description,
375                         tags,
376                         modules,
377                         synth_template,
378                         patch_binary,
379                         id,              -- original id becomes
                            forked_from_id
380                         now()
381                     FROM patch_histories
382                     WHERE id = $1
383                     RETURNING id;`,
384                     [msg.data.forked_from_id]
385                     );
386 
387                     // Send back the new fork's id so the client can
                            load or track it
388                     ws.send(JSON.stringify({
389                         cmd: 'newFork',
390                         success: true,
391                         newId: result.rows[0].id
```

```
392                     }));
393
394                     // getUpdatedHistories(ws)
395
396                 } catch (err) {
397                     console.error('DB error (newFork):', err);
398                     ws.send(JSON.stringify({
399                     cmd: 'newFork',
400                     success: false,
401                     error: err.message
402                     }));
403                 }
404                 })();
405
406
407             break;
408
409
410             case 'getPatchHistory':
411
412                 (async () => {
413                     try {
414                         const result = await pool.query(
415                             `SELECT * FROM patch_histories WHERE id
                                = $1`,
416                             [msg.id]
417                         );
418
419                         if (result.rows.length === 0) {
420                             ws.send(JSON.stringify({
421                                 cmd: 'patchHistoryNotFound',
422                                 id: msg.id
423                             }));
424                         } else {
425                             ws.send(JSON.stringify({
426                                 cmd: 'retrievedPatchHistory',
427                                 data: result.rows[0]
428                             }));
429                         }
430                     } catch (err) {
431                         console.error('DB error (getPatchHistoryById
                            ):', err);
432                         ws.send(JSON.stringify({
433                             cmd: 'patchHistoryLoaded',
434                             error: err.message
435                         }));
436                     }
437                 })();
438             break
439
440
```

```
441             case 'saveSynth':
442
443                 (async () => {
444                     try {
445
446                         const { name, author, description, tags,
                                synth_json } = msg.data;
447
448                         const result = await pool.query(
449                             `INSERT INTO synth_templates
450                             (name, author, description, tags,
                                    created_at, synth_json)
451                             VALUES ($1, $2, $3, $4, now(), $5)
452                             RETURNING id`,
453                             [name, author, description, tags,
                                synth_json]
454                         );
455
456                         ws.send(JSON.stringify({
457                             cmd: 'saveSynthTemplateResponse',
458                             success: true,
459                             synthFileId: result.rows[0].id
460                         }));
461
462                         wss.clients.forEach((client) => {
463                             getSynthTemplates(client);
464                         });
465                         } catch (err) {
466                         console.error('DB error:', err);
467                         ws.send(JSON.stringify({
468                             cmd: 'saveSynthTemplateResponse',
469                             success: false,
470                             error: err.message
471                         }));
472                     }
473                 })(); // <-- IIFE to allow await
474
475             break
476
477             case 'updatePatchHistoryEntry':
478
479
480                 (async () => {
481                     try {
482                     const { id, patch_binary, authors,
                            forked_from_id } = msg.data;
483
484                     // Decode base64 patch binary from client
485                     const binaryBuffer = Buffer.from(patch_binary, '
                            base64');
486
```

```
                    const result = await pool.query(
                        `UPDATE patch_histories
                        SET patch_binary = $1,
                            authors = $2,
                            forked_from_id = $3
                        WHERE id = $4`,
                        [binaryBuffer, authors, forked_from_id, id]
                    );

                    getUpdatedHistories(ws)

                    } catch (err) {
                    console.error('DB error (updatePatcHistory):',
                        err);
                    ws.send(JSON.stringify({
                        cmd: 'updatePatchHistoryResponse',
                        success: false,
                        error: err.message
                    }));
                    }
                })();
            break

            case 'updatePatchHistoryMetadata':


                (async () => {

                    try {
                    const { id, name, description, tags } = msg.data
                        ;

                    const cleanTags = Array.isArray(tags)
                    ? tags.map(t => t.trim()).filter(t => t.length >
                        0)
                    : [];

                    const result = await pool.query(
                        `UPDATE patch_histories
                        SET name = $1,
                            description = $2,
                            tags = $3
                        WHERE id = $4`,
                        [name, description, cleanTags, id]
                    );

                    getUpdatedHistories(ws)

                    } catch (err) {
                    console.error('    DB error (updatePatchMetadata
                        ):', err);
```

```
534                     ws.send(JSON.stringify({
535                         cmd: 'updatePatchHistoryResponse',
536                         success: false,
537                         error: err.message
538                     }));
539                     }
540                 })();
541             break
542 
543             case 'getSynthFile':
544                 (async () => {
545                     try {
546                         const result = await pool.query(
547                             `SELECT * FROM synth_templates WHERE id
                                = $1`,
548                             [msg.id]
549                         );
550 
551                         if (result.rows.length === 0) {
552                             ws.send(JSON.stringify({
553                             cmd: 'synthTemplateNotFound',
554                             id: msg.id
555                             }));
556                         } else {
557                             ws.send(JSON.stringify({
558                                 cmd: 'retrievedSynthFile',
559                                 data: result
560                             }));
561                         }
562                     } catch (err) {
563                         console.error('DB error (
                            getSynthTemplateById):', err);
564                         ws.send(JSON.stringify({
565                             cmd: 'synthTemplateLoaded',
566                             error: err.message
567                         }));
568                     }
569                 })();
570             break
571 
572             case 'getSynthTemplates':
573                 if(msg.filter){
574                     getSynthTemplates(ws, msg.filter, msg.query);
575                 } else {
576                     getSynthTemplates(ws);
577                 }
578 
579 
580             break
581 
582             case 'getPatchHistories':
```

```
583                 if(msg.filter){
584                     getPatchHistories(ws, msg.filter, msg.query);
585                 } else {
586                     getPatchHistories(ws);
587                 }
588 
589 
590             break
591 
592 
593 
594             case 'updateGraph':
595                 patchHistory = msg.patchHistory
596                 updateHistoryGraph(ws, patchHistory, msg.
                        docHistoryGraphStyling)
597             break
598 
599 
600             case 'clearHistoryGraph':
601                 historyDAG_cy.elements().remove();
602                 if(existingHistoryNodeIDs){
603                     existingHistoryNodeIDs.clear()
604                 }
605                 historyDAG_cy.layout(graphLayouts[graphStyle]).run()
606             break
607 
608             // case 'eraseRoomPatchHistory':
609             //     // erase the patch history for all peers in this
                    room
610             //     historyDAG_cy.elements().remove();
611             //     if(existingHistoryNodeIDs){
612             //         existingHistoryNodeIDs.clear()
613             //     }
614             //     historyDAG_cy.layout(graphLayouts[graphStyle]).
                    run()
615             // break
616 
617             case 'collapseNodes':
618 
619             break
620 
621             case 'expandNodes':
622 
623             break
624             case 'joinRoom':
625                 // Assign the connecting client to a room
626                 ws.room = assignRoom(ws, msg.room);
627 
628                 ws.peerID = msg.peerID
629                 console.log(`New client assigned to ${ws.room}`);
630                 // console.log('number of peers in room',  )
```

```
631                 // update all lobby pages
632                 wss.clients.forEach((client) => {
633                     if (client !== ws && client.lobby === true) {
634                         sendRooms(client)
635                     }
636                 });
637
638                 if(sequencerStates[ws.room]){
639                     console.log('sequencer state exists for this
                        room:', sequencerStates[ws.room])
640                 } else {
641                     sequencerStates[ws.room] = {}
642                     console.log('sequencerStates', sequencerStates)
643                 }
644                 // console.log('ws.room', ws.room, 'room info',
                    rooms[ws.room])
645
646             break
647             case 'newPeer':
648                 // Convert the incoming message to a string if
                        i t s  a Buffer.
649                 const payload = Buffer.isBuffer(message) ? message.
                    toString() : message;
650                 // peers[msg.peerID] = {}
651
652
653                 // Relay the message to the other client in the same
                        room (if exists)
654                 // Use ws.room (assigned in 'joinRoom') to determine
                        the correct room.
655                 const clientRoom = ws.room;
656                 if (clientRoom && rooms[clientRoom]) {
657                 rooms[clientRoom].forEach(client => {
658                     if (client !== ws && client.readyState === ws.
                        OPEN) {
659                     client.send(JSON.stringify({
660                         cmd: 'newPeer',
661                         msg: payload
662                     }), { binary: false });
663                     }
664                 });
665                 } else {
666                 console.error('Client not assigned to any room');
667                 }
668
669
670
671             break
672
673             case 'getRooms':
674                 ws.lobby = true
```

```
675                 // Build an array of active rooms.
676                 sendRooms(ws)
677             break;
678
679             case 'sequencerStateUpdate':
680                 console.log('update:', msg)
681                 // console.log('room', ws.room, 'ws.room', ws.room)
682                 sequencerStates[msg.room] = msg.state
683
684                 console.log('sequencerStates', sequencerStates)
685
686             break
687
688             case 'getSequencerState':
689                 ws.send(JSON.stringify({
690                     cmd: 'sequencerState',
691                     state: sequencerStates[msg.room]
692                 }))
693                 // console.log('ws.room', ws.room, 'room info',
                        rooms[ws.room])
694             break
695
696
697             default: console.log('no switch case exists for msg:',
                    message)
698         }
699     });
700
701     // Handle client disconnection
702     ws.on('close', () => {
703         console.log('Client disconnected');
704         numClients--
705         console.log('number of clients:', numClients)
706         if (ws.room && rooms[ws.room]) {
707             // Remove the client from the room
708             rooms[ws.room] = rooms[ws.room].filter(client => client
                    !== ws);
709             // Clean up the room if empty
710             if (rooms[ws.room].length === 0) {
711                 console.log('should be removing seq state now')
712                 // first clear the sequencer state
713                 delete sequencerStates[ws.room]
714                 // then remote the room
715                 delete rooms[ws.room];
716             }
717             // update all lobby clients
718             wss.clients.forEach((client) => {
719                 if (client.lobby === true) {
720                     sendRooms(client)
721                 }
722             });
```

```
723         }
724
725     });
726
727     // Handle errors
728     ws.on('error', (error) => {
729         console.error('WebSocket error:', error);
730     });
731 });
732
733 // Start the server
734 server.listen(PORT, () => {
735     console.log(`WebSocket server running on ws://localhost:${PORT
        }`);
736 });
737
738 function sendRooms(ws){
739     const activeRooms = [];
740     for (const roomName in rooms) {
741         // Only include rooms with at least one peer.
742         if (rooms[roomName].length > 0) {
743         const roomInfo = {
744             room: roomName,
745             peer1: rooms[roomName][0].peerID || null,
746             peer2: rooms[roomName].length > 1 ? rooms[roomName][1].
                peerID || null : null,
747             sequencerState: ws.sequencerState || 'empty'
748         };
749         activeRooms.push(roomInfo);
750         }
751     }
752     // Send the room info back to the client that requested it.
753     ws.send(JSON.stringify({
754         cmd: 'roomsInfo',
755         rooms: activeRooms
756     }));
757 }
758
759 function updateHistoryGraph(ws, patchHistory, docHistoryGraphStyling
    ){
760
761     if (!existingHistoryNodeIDs || existingHistoryNodeIDs.size ===
        0){
762         existingHistoryNodeIDs = new Set(historyDAG_cy.nodes().map(
            node => node.id()));
763     }
764
765     if(!patchHistory || !patchHistory.branchOrder) return
766     const { nodes, edges, historyNodes } = buildHistoryGraph(
767         patchHistory,
768         existingHistoryNodeIDs,
```

```
769             docHistoryGraphStyling
770         );
771         // dumb hack for weird bug where the parent prop in each node
                was coming out undefined despite existing in the return
                statement of buildHistoryGraph
772         const stringed = JSON.parse(JSON.stringify(nodes, null, 2))
773         // Run the layout and get the rendered graph
774         // historyDAG_cy.layout(layout).run();
775         try {
776             if (nodes.length > 0) {
777                 historyDAG_cy.add(stringed);
778             }
779             if (edges.length > 0) {
780                 historyDAG_cy.add(edges); //        this is where it was
                        crashing
781             }
782         } catch (err) {
783             console.error('Failed to update Cytoscape graph:', err.
                    message);
784             console.error('       Possibly due to missing source or
                    target node.');
785             console.error('       Reason:', err.stack);
786             console.error('       Edges:', JSON.stringify(edges, null,
                    2));
787             // send message to client to force a new patch history
788             setTimeout(() => {
789                 ws.send(JSON.stringify({
790                     cmd: "forceNewPatchHistoryDueToError",
791                     message: 'Server failed to create graph; forcing a
                            new patch history now...'
792                 }))
793             }, 1000);
794
795
796             return; // prevent graph layout from running
797         }
798         existingHistoryNodeIDs = historyNodes
799
800         historyDAG_cy.layout(graphLayouts[graphStyle]).run();
801
802         // Send the graph JSON back to the client
803         const graphJSON = historyDAG_cy.json();
804
805         ws.send(JSON.stringify({
806             cmd: "historyGraphRenderUpdate",
807             data: graphJSON
808         }))
809     }
810
811     async function getSynthTemplates(ws, filter, query) {
812         if(filter){
```

```
813         switch (filter){
814             case 'tags':
815                 const tag = query;
816                 const result = await pool.query(
817                     `SELECT id, name, author, description, tags FROM
                            synth_templates WHERE $1 = ANY(tags) ORDER
                           BY created_at DESC`,
818                     [tag]
819                 );
820 
821                 // Filter tags to include only the selected one
822                 const filteredRows = result.rows.map(entry => ({
823                     ...entry,
824                     tags: entry.tags.filter(t => t === tag)
825                 }));
826 
827                 ws.send(JSON.stringify({ cmd: 'synthTemplatesList',
                        data: filteredRows }));
828             break
829 
830             case 'authors':
831                 const author = query;
832                 try {
833                     const result = await pool.query(
834                     `SELECT id, name, author, description, tags
835                     FROM synth_templates
836                     WHERE author = $1
837                     ORDER BY created_at DESC`,
838                     [author]
839                     );
840 
841                     ws.send(JSON.stringify({
842                         cmd: 'synthTemplatesList',
843                         data: result.rows
844                     }));
845                 } catch (err) {
846                     console.error('DB error (filterByAuthor):', err)
                            ;
847                     ws.send(JSON.stringify({
848                         cmd: 'synthTemplatesList',
849                         data: [],
850                         error: err.message
851                     }));
852                 }
853             break
854         }
855     } else {
856         try {
857             const result = await pool.query(
858                 `SELECT id, name, author, description, tags FROM
                        synth_templates ORDER BY created_at DESC`
```

```
            );

            ws.send(JSON.stringify({
                cmd: 'synthTemplatesList',
                data: result.rows
            }));
        } catch (err) {
            console.error('DB error (getSynthTemplates):', err);
            ws.send(JSON.stringify({
                cmd: 'synthTemplatesList',
                data: [],
                error: err.message
            }));
        }
    }

}


async function getPatchHistories(ws, filter, query) {

    try {
        let result;

        switch (filter) {
            case 'authors':
                result = await pool.query(
                    `SELECT * FROM patch_histories WHERE authors @>
                       ARRAY[$1] ORDER BY created_at DESC`,
                    [query[0]]
                );
                break;

            case 'modules':
                result = await pool.query(
                    `SELECT * FROM patch_histories WHERE $1 = ANY(
                       modules) ORDER BY created_at DESC`,
                    [query]
                );
                break;

            case 'forksOf':
                result = await pool.query(
                    `SELECT * FROM patch_histories WHERE
                       forked_from_id = $1 ORDER BY created_at ASC`,
                    [query]
                );
                break;

            case 'id':
                result = await pool.query(
```

```
                    `SELECT * FROM patch_histories WHERE id = $1`,
                    [query]
                );
                break;

            default:
                result = await pool.query(
                    `SELECT * FROM patch_histories ORDER BY
                       created_at DESC LIMIT 50`
                );
        }

        ws.send(JSON.stringify({
            cmd: 'patchHistoriesList',
            data: result.rows
        }));

    } catch (err) {
        console.error('DB error (getPatchHistories):', err);
        ws.send(JSON.stringify({
            cmd: 'patchHistoriesList',
            data: [],
            error: err.message
        }));
    }
}

async function getUpdatedHistories(ws){

    try {
        let updatedList = await pool.query(
            `SELECT
                id,
                name,
                authors,
                description,
                tags,
                modules,
                forked_from_id,
                created_at
                FROM patch_histories
                ORDER BY created_at DESC
                LIMIT 50;
`
        );
        wss.clients.forEach((client) => {
            client.send(JSON.stringify({
                cmd: 'patchHistoriesList',
                data: updatedList.rows
            }));
        });
```

```
957
958       } catch (err) {
959           console.error('DB error (getUpdatedHistoryList):', err);
960       }
961
962   }
```

Listing D.1: The server app code

Appendix E

buildHistoryGraph.js

```
1
2  /**
3   * Generates nodes and edges for a history graph.
4   */
5  import { fileURLToPath } from 'url';
6  import { dirname, join } from 'path';
7
8  const __filename = fileURLToPath(import.meta.url);
9  const __dirname = dirname(__filename);
10
11 const historyGraphYIncrement = 75
12
13 function buildHistoryGraph(patchHistory, existingHistoryNodeIDs,
       docHistoryGraphStyling) {
14
15     const outputPath = join(__dirname, 'patchHistory.json');
16     // writeFileSync(outputPath, JSON.stringify(patchHistory, null,
           2));
17     const nodes = [];
18     const edges = [];
19
20     const nodeIdToYPos = new Map();
21     const branchRootY = new Map();
22
23     // Pass 1: calculate branch root Y positions
24     patchHistory.branchOrder.forEach((branchName, branchIndex) => {
25     const branch = patchHistory.branches[branchName];
26     const firstItem = branch.history[0];
27     let y;
28
29     if (firstItem && firstItem.parent) {
30         const parentId = Array.isArray(firstItem.parent) ? firstItem
               .parent[0] : firstItem.parent;
31         const parentY = nodeIdToYPos.get(parentId);
```

```
32          y = parentY !== undefined ? parentY : 0;
33      } else {
34          y = 0; // root branch
35      }
36
37          branchRootY.set(branchName, y);
38      });
39
40      const plannedYPositions = new Map();
41
42      patchHistory.branchOrder.forEach(branchName => {
43          const branch = patchHistory.branches[branchName];
44          const firstItem = branch.history[0];
45
46          let rootY = 0;
47
48          if (firstItem && firstItem.parent) {
49              const parentId = Array.isArray(firstItem.parent)
50              ? firstItem.parent[0]
51              : firstItem.parent;
52
53              const parentY = plannedYPositions.get(parentId);
54              rootY = parentY !== undefined ? parentY : 0;
55          }
56
57          branch.history.forEach((item, i) => {
58                  const y = rootY - (i + 1) * historyGraphYIncrement;
59                  plannedYPositions.set(item.hash, y);
60              });
61      });
62
63
64
65
66
67      // Accessing branches in order, create nodes and edges for each
            branch
68      patchHistory.branchOrder.forEach((branchName, branchIndex) => {
69          const branch = patchHistory.branches[branchName];
70          const rootY = branchRootY.get(branchName);
71          // Iterate over each history item in the branch
72          branch.history.forEach((item, nodeIndex) => {
73              const nodeId = item.hash;
74              console.log(item.peer)
75              if (existingHistoryNodeIDs.has(nodeId)) return;
76
77              const yPos = plannedYPositions.get(item.hash);
78
79              nodeIdToYPos.set(nodeId, yPos);
80
81              let label;
```

```
82              let parent = []
83              let sequencerTable
84              let mergeData
85              // we now store the parent module data in the change
                   message, so extract that so it doesn't appear as the
                   label, and place it in the 'parent' prop
86              // check if its a $PARENT or $PARENTS condition
87              if(item.msg.includes('$PARENT ')){
88
89                  parent = item.msg.split('$PARENT ')[1]
90                  label = `${item.msg.split('$PARENT ')[0]} ${parent.
                       split('_')[0]}_${parent.split('_')[1]}`
91              } else if (item.msg.includes("$PARENTS ")){
92                  label = item.msg.split('$PARENTS ')[0]
93                  parent = item.msg.split('$PARENTS ')[1]
94              } else if (item.msg.includes('.fpsynth')){
95                  label = `loaded ${item.msg}`
96                  // parent = []
97              } else if(item.msg.includes('sequence')){
98                  label = item.msg.split('tableData:')[0]
99                  sequencerTable = JSON.parse(item.msg.split('
                       tableData:')[1])
100             }
101             else if(item.msg.includes('draw')){
102                 label = item.msg
103                 // sequencerTable = JSON.parse(item.msg.split('
                       tableData:')[1])
104             }
105             else if(item.msg.includes('loaded')){
106                 label = item.msg
107                 // sequencerTable = JSON.parse(item.msg.split('
                       tableData:')[1])
108             }
109             else if(item.msg.includes('merge')){
110                 label = item.msg
111                 mergeData = {
112                     parents: item.parent,
113                     nodes: item.nodes
114                 }
115                 // sequencerTable = JSON.parse(item.msg.split('
                       tableData:')[1])
116             }
117
118             const newNode = {
119                 group: "nodes",
120                 data: {
121                     id: nodeId,
122                     label: label,
123                     color: docHistoryGraphStyling.nodeColours[item.
                           msg.split(" ")[0]] || "#ccc",
124                     branch: branchName,
```

```
125                     parents: parent || null,
126                     sequencerTable: sequencerTable || null,
127                     timeStamp: item.timeStamp,
128                     mergeData: mergeData || null,
129                     peer: item.peer || null
130
131                 },
132                 // Add a manual position!
133                 position: {
134                     x: branchIndex * 110, // horizontal slot per
                            branch
135                     y: yPos   // stack nodes top to bottom
136                 }
137
138
139             }
140
141
142             // Add node to the history graph
143             nodes.push(newNode);
144
145             // Add the newly added node's ID to the set to track it
146             existingHistoryNodeIDs.add(nodeId);
147
148         });
149     });
150
151     // now that all nodes are made, create their edges
152
153     patchHistory.branchOrder.forEach((branchName) => {
154         const branch = patchHistory.branches[branchName];
155
156         // Iterate over each history item in the branch
157         branch.history.forEach((item) => {
158             const nodeId = item.hash;
159
160             // ! this is where we'll also figure out how to deal
                    with 2 parents in the case of a merge!
161             // If the history item has a parent, add an edge to
                    connect the parent
162             if (item.parent) {
163                 // first check if item.parent is an array. if it is,
                        then this node is a merge between 2 parents
164                 if(Array.isArray(item.parent)){
165                     item.parent.forEach((parent, index)=>{
166                         edges.push({
167                             group: "edges",
168                             data: {
169                                 id: `${item.parent[index]}_to_${
                                        nodeId}`,
170                                 source: item.parent[index],
```

```
                                    target: nodeId,
                                },
                            });
                        })
                    } else {
                                    // //Make sure the parent node also
                                       exists before adding the edge
                    // if (existingHistoryNodeIDs.has(item.parent)) {
                        edges.push({
                            group: "edges",
                            data: {
                                id: `${item.parent}_to_${nodeId}`,
                                source: item.parent,
                                target: nodeId,
                            },
                        });
                    }
                }

            });


    });

    return { nodes, edges, existingHistoryNodeIDs };
}

export default buildHistoryGraph;
```

Listing E.1: Returns a Cytoscape-compatible `nodes, edges` object that visualizes an Automerge patch-history DAG.

Appendix F

Custom DSP

```
var DSP = class extends AudioWorkletProcessor {
  constructor() {
        super();
        this.currentState = { nodes: {}, signalConnections: [],
           outputConnections: [], cvConnections: [] };
        this.nextState = { nodes: {}, signalConnections: [],
           outputConnections: [], cvConnections: [] };
        this.signalBuffers = {};
        this.signalBuffersCurrent = {};
        this.signalBuffersNext = {};
        this.feedbackBuffers = {};
        this.crossfadeInProgress = false;
        this.crossfadeProgress = 0;
        this.crossfadeStep = 1 / (sampleRate / 128 * 0.1);
        this.outputVolume = 0.5;
        this.port.onmessage = (event) => this.handleMessage(event.
           data);
        this.analyze = false;
        this.audioStatus = 'suspended';
    }
    getSampleRate() {
    return sampleRate;
    }
    // receive module from main app, walk it, build it here
    audioNodeBuilder(type, moduleName, params, loadState) {
        switch (type) {
          case "VCA":
            let vca = {
              node: "VCA",
              structure: "webAudioNode",
              baseParams: {
                gain: parseFloat(1),
                "gain cv +/-": parseFloat(params["gain cv +/-"])
              },
```

```
            modulatedParams: {
              // offsets for modulation
              gain: 0
            },
            output: new Float32Array(128),
            modulationTarget: null,
            // Target node or parameter for modulation
            startTime: null,
            // Optional: Scheduled start time
            stopTime: null
            // Optional: Scheduled stop time
          };
          if (loadState) {
            this.nextState.nodes[moduleName] = vca;
          } else {
            this.currentState.nodes[moduleName] = vca;
          }
          break;
        case "HighPassFilter":
          let hpf = {
            node: "HighPassFilter",
            structure: "webAudioNode",
            baseParams: {
              freq: parseFloat(params.freq),
              Q: parseFloat(params.Q),
              "freq cv +/-": parseFloat(params["freq cv +/-"]),
              "Q cv +/-": parseFloat(params["Q cv +/-"])
            },
            modulatedParams: {
              // offsets for modulation
              freq: 0,
              Q: 0
            },
            output: new Float32Array(128),
            modulationTarget: null,
            // Target node or parameter for modulation
            startTime: null,
            // Optional: Scheduled start time
            stopTime: null
            // Optional: Scheduled stop time
          };
          if (loadState) {
            this.nextState.nodes[moduleName] = hpf;
          } else {
            this.currentState.nodes[moduleName] = hpf;
          }
          break;
        case "LFO":
        case "SLOWFO":
        case "Oscillator":
          let osc = {
```

```
            node: "Oscillator",
            structure: "webAudioNode",
            baseParams: {
              frequency: parseFloat(params.frequency),
              gain: parseFloat(1),
              "freq cv +/-": parseFloat(params["freq cv +/-"]),
              type: params.type
            },
            modulatedParams: {
              // offsets for modulation
              frequency: 0,
              gain: 0
            },
            output: new Float32Array(128),
            phase: 0,
            customWaveform: null,
            type: params.type,
            modulationTarget: null,
            // Target node or parameter for modulation
            startTime: null,
            // Optional: Scheduled start time
            stopTime: null
            // Optional: Scheduled stop time
          };
          if (loadState) {
            this.nextState.nodes[moduleName] = osc;
          } else {
            this.currentState.nodes[moduleName] = osc;
          }
          break;
        case "Gain":
        case "ModGain":
          let gain = {
            node: "Gain",
            structure: "webAudioNode",
            baseParams: {
              gain: parseFloat(params.gain) || 1
            },
            modulatedParams: {
              gain: 0
              // Offset for modulation
            },
            output: new Float32Array(128)
          };
          if (loadState) {
            this.nextState.nodes[moduleName] = gain;
          } else {
            this.currentState.nodes[moduleName] = gain;
          }
          break;
        case "Mixer":
```

```
134             console.warn("Mixer DSP code is not yet working. the
                    inputs are not being processed correctly (returning
                    undefined)");
135             let mixer = {
136               node: "Mixer",
137               structure: "webAudioNode",
138               baseParams: {
139                 gainA: parseFloat(params.gainA) || 1,
140                 gainB: parseFloat(params.gainB) || 1
141               },
142               modulatedParams: {},
143               output: new Float32Array(128)
144             };
145             if (loadState) {
146               this.nextState.nodes[moduleName] = mixer;
147             } else {
148               this.currentState.nodes[moduleName] = mixer;
149             }
150             break;
151           case "Delay":
152           case "Flutter":
153             let delay = {
154               node: "Delay",
155               structure: "webAudioNode",
156               baseParams: {
157                 delayTime: parseFloat(params.delayTime) || 500,
158                 "time cv +/-": parseFloat(params["time cv +/-"]) ||
                        100,
159                 feedback: parseFloat(params.feedback) || 0.5,
160                 "feedback cv +/-": parseFloat(params["feedback cv
                        +/-"]) || 0.3,
161                 dryWet: parseFloat(params.dryWet) || 0.4,
162                 "dryWet cv +/-": parseFloat(params["dryWet cv +/-"])
                         || 0
163               },
164               connections: {
165                 feedback: null
166                 // Will store a Web Audio GainNode for feedback
167               },
168               modulatedParams: {
169                 delayTime: 0,
170                 // Offset for modulation
171                 feedback: 0,
172                 dryWet: 0
173               },
174               output: new Float32Array(128)
175             };
176             if (loadState) {
177               this.nextState.nodes[moduleName] = delay;
178             } else {
179               this.currentState.nodes[moduleName] = delay;
```

```
180             }
181             break;
182           case "feedbackDelayNode":
183             let feedbackDelayNode = {
184               node: "feedbackDelayNode",
185               structure: "webAudioNode",
186               output: new Float32Array(128)
187               // Fixed output buffer for block size
188             };
189             if (loadState) {
190               this.nextState.nodes[moduleName] = feedbackDelayNode;
191             } else {
192               this.currentState.nodes[moduleName] =
                      feedbackDelayNode;
193             }
194             break;
195           case "Pulses":
196             let pulses = {
197               node: "Pulses",
198               structure: "webAudioNode",
199               baseParams: {
200                 stepCount: parseFloat(params.stepCount) || 8,
201                 tempo: parseFloat(params.tempo) || 120,
202                 "tempo cv +/-": parseFloat(params["tempo cv +/-"])
                      || 10,
203                 pulseWidth: parseFloat(params.gateLength) || 0.5
204               },
205               modulatedParams: {
206                 tempo: 0,
207                 pulseWidth: 0
208               },
209               output: new Float32Array(128),
210               // Single output buffer
211               stepIndex: 0,
212               clockPhase: 0
213             };
214             if (loadState) {
215               this.nextState.nodes[moduleName] = pulses;
216             } else {
217               this.currentState.nodes[moduleName] = pulses;
218             }
219             break;
220           case "Euclid":
221             let euclideanPulses = {
222               node: "Euclid",
223               structure: "webAudioNode",
224               baseParams: {
225                 numSteps: parseInt(params.numSteps) || 8,
226                 activeSteps: parseInt(params.activeSteps) || 8,
227                 tempo: parseFloat(params.tempo) || 120,
```

```
228                 "tempo cv +/-": parseFloat(params["tempo cv +/-"])
                        || 10,
229                 ratchet: parseInt(params.ratchet) || 0,
230                 "ratchet cv +/-": parseFloat(params["ratchet cv +/-"
                        ]) || 10
231               },
232               modulatedParams: {
233                 tempo: 0,
234                 ratchet: 0
235               },
236               output: new Float32Array(128),
237               // Single output buffer
238               stepIndex: 0,
239               clockPhase: 0
240             };
241             if (loadState) {
242               this.nextState.nodes[moduleName] = euclideanPulses;
243             } else {
244               this.currentState.nodes[moduleName] = euclideanPulses;
245             }
246             break;
247           default:
248         }
249       }
250     // receive messages from main thread
251       handleMessage(msg) {
252         switch (msg.cmd) {
253           case "setOutputVolume":
254             this.outputVolume = msg.data;
255             break;
256           case "clearGraph":
257             this.currentState.nodes = {};
258             this.currentState.signalConnections = [];
259             this.currentState.outputConnections = [];
260             this.currentState.cvConnections = [];
261             break;
262           case "loadVersion":
263             const synthGraph = msg.data;
264             if (this.crossfadeInProgress) return;
265             this.nextState = {
266               nodes: {},
267               signalConnections: [],
268               outputConnections: [],
269               cvConnections: []
270             };
271             Object.keys(synthGraph.modules).forEach((moduleID) => {
272               const module = synthGraph.modules[moduleID];
273               let moduleParams = null;
274               if (module.params) {
275                 moduleParams = module.params;
276               }
```

```
277                 if (moduleID.startsWith("feedbackDelayNode")) {
278                   this.audioNodeBuilder("feedbackDelayNode", moduleID,
                          null, "loadstate");
279                 } else if (module.structure === "webAudioNodes") {
280                   this.audioNodeBuilder(module.type, moduleID, module.
                          params, "loadstate");
281                 } else if (module.structure === "rnboDevices") {
282                   this.port.postMessage({ cmd: "fetchRNBOsrc", data:
                          module });
283                 } else {
284                   console.warn("node module structure defined for
                          module ", moduleID);
285                 }
286               });
287               synthGraph.connections.forEach((cable) => {
288                 if (cable.target.includes("AudioDestination")) {
289                   this.nextState.outputConnections.push(cable.source);
290                 } else if (cable.target.split(".")[1].startsWith("IN")
                        ) {
291                   this.nextState.signalConnections.push(cable);
292                 } else {
293                   cable.param = cable.target.split(".")[1];
294                   this.nextState.cvConnections.push(cable);
295                 }
296               });
297               this.crossfadeInProgress = true;
298               this.crossfadeProgress = 0;
299               break;
300             case "setSignalAnalysis":
301               this.analyze = msg.data;
302               break;
303             case "addNode":
304               if (msg.structure === "webAudioNodes") {
305                 this.audioNodeBuilder(msg.data.moduleName);
306               } else if (msg.structure === "feedbackDelayNode") {
307                 this.audioNodeBuilder("feedbackDelayNode", msg.data);
308               }
309               break;
310             case "removeNode":
311               delete this.currentState.nodes[msg.data];
312               this.currentState.signalConnections = this.currentState.
                      signalConnections.filter(
313                 (item) => item.source.split(".")[0] !== msg.data &&
                        item.target.split(".")[0] !== msg.data
314               );
315               this.currentState.outputConnections = this.currentState.
                      outputConnections.filter(
316                 (item) => item !== msg.data
317               );
318               this.currentState.cvConnections = this.currentState.
                      cvConnections.filter(
```

```
319               (item) => item.source.split(".")[0] !== msg.data &&
                      item.target.split(".")[0] !== msg.data
320             );
321             break;
322           case "addCable":
323             if (msg.data.target.includes("AudioDestination")) {
324               this.currentState.outputConnections.push(msg.data.
                      source);
325             } else if (msg.data.target.split(".")[1].startsWith("IN"
                    )) {
326               this.currentState.signalConnections.push(msg.data);
327             } else {
328               msg.data.param = msg.data.target.split(".")[1];
329               this.currentState.cvConnections.push(msg.data);
330             }
331             break;
332           case "connectToOutput":
333             if (this.currentState.nodes[msg.data.split(".")[0]] && !
                    this.currentState.outputConnections.includes(msg.data
                    )) {
334             }
335             break;
336           case "connectCV":
337             this.currentState.cvConnections.push({
338               source: msg.data.source,
339               target: msg.data.target,
340               param: msg.data.param
341               // The parameter to modulate
342             });
343             break;
344           case "removeCable":
345             if (msg.data.target.includes("AudioDestination")) {
346               const index = this.currentState.outputConnections.
                      findIndex(
347                 (item) => item === msg.data.source
348                 // item.source === msg.data.source &&
349                 // item.target === msg.data.target
350               );
351               if (index !== -1) {
352                 this.currentState.outputConnections.splice(index, 1)
                        ;
353               }
354             } else if (msg.data.target.split(".")[1] === "IN") {
355               const index = this.currentState.signalConnections.
                      findIndex(
356                 (item) => (
357                   // item === msg.data.source
358                   item.source === msg.data.source && item.target ===
                          msg.data.target
359                 )
360               );
```

```
361               if (index !== -1) {
362                 this.currentState.signalConnections.splice(index, 1)
                      ;
363               }
364             } else {
365               const index = this.currentState.cvConnections.
                     findIndex(
366               (item) => (
367                 // item === msg.data.source
368                 item.source === msg.data.source && item.target ===
                        msg.data.target
369               )
370             );
371             if (index !== -1) {
372               let thisCvConnection = this.currentState.
                     cvConnections[index];
373               this.currentState.nodes[thisCvConnection.target.
                     split(".")[0]].modulatedParams[thisCvConnection.
                     target.split(".")[1]] = 0;
374               this.currentState.cvConnections.splice(index, 1);
375             }
376           }
377           break;
378         case "paramChange":
379           const targetNode = this.currentState.nodes[msg.data.
                  parent];
380           if (targetNode && targetNode.baseParams[msg.data.param]
                  !== void 0) {
381             const newValue = parseFloat(msg.data.value);
382             if (!isNaN(newValue)) {
383               targetNode.baseParams[msg.data.param] = newValue;
384             } else {
385               targetNode.baseParams[msg.data.param] = msg.data.
                     value;
386             }
387           } else {
388             if(this.audioStatus === 'running'){
389               // only warn about unknown params if the DSP is
                     actually running (otherwise we were getting this
                     warning when DSP was off, which makes sense)
390               console.warn(`Parameter ${msg.data.param} not found
                     for node ${msg.data.parent}`);
391
392             }
393           }
394           break;
395
396           case 'audioStatus':
397             this.audioStatus = msg.state
398           break
399         default:
```

```
400             console.log(`no switch case exists for ${msg.cmd}`);
401         }
402     }
403     clamp(value, min, max) {
404         return Math.min(Math.max(value, min), max);
405     }
406     // this runs every block (128 samples)
407     process(inputs, outputs) {
408         const output = outputs[0][0];
409         output.fill(0);
410         // handle crossfading from making a structural change or
                loading a version
411         if (this.crossfadeInProgress) {
412           this.crossfadeProgress += this.crossfadeStep;
413           if (this.crossfadeProgress >= 1) {
414             this.crossfadeProgress = 1;
415             this.crossfadeInProgress = false;
416             this.currentState = (0, import_cloneDeep.default)(this.
                    nextState);
417             this.nextState = {
418               nodes: {},
419               signalConnections: [],
420               outputConnections: [],
421               cvConnections: []
422             };
423           }
424         }
425         this.signalBuffersCurrent = this.signalBuffersCurrent || {};
426         this.signalBuffersNext = this.signalBuffersNext || {};
427         for (const id in this.currentState.nodes) {
428           if (!this.signalBuffersCurrent[id]) {
429             this.signalBuffersCurrent[id] = new Float32Array(128);
430           }
431         }
432         for (const id in this.nextState.nodes) {
433           if (!this.signalBuffersNext[id]) {
434             this.signalBuffersNext[id] = new Float32Array(128);
435           }
436         }
437         const processGraph = (state, signalBuffers, feedbackBuffers)
                => {
438           const visited = /* @__PURE__ */ new Set();
439           const processNode = (id) => {
440             if (visited.has(id)) return;
441             visited.add(id);
442             const node = state.nodes[id];
443             if (!node) return;
444             if (!signalBuffers[id]) signalBuffers[id] = new
                    Float32Array(128);
445             const inputBuffer = new Float32Array(128);
```

```
446             state.signalConnections.filter((conn) => conn.target.
                   includes(id)).forEach((conn) => {
447               const sourceId = conn.source.split(".")[0];
448               const sourceOutput = conn.source.split(".")[1];
449               processNode(sourceId);
450               if (state.nodes[sourceId] && state.nodes[sourceId].
                     output && typeof state.nodes[sourceId].output === "
                     object") {
451                 const sourceBuffer2 = state.nodes[sourceId].output[
                       sourceOutput] || new Float32Array(128);
452                 for (let i = 0; i < 128; i++) {
453                   inputBuffer[i] += sourceBuffer2[i];
454                 }
455               } else {
456                 const sourceBuffer2 = signalBuffers[sourceId] || new
                       Float32Array(128);
457                 for (let i = 0; i < 128; i++) {
458                   inputBuffer[i] += sourceBuffer2[i];
459                 }
460               }
461               const sourceBuffer = signalBuffers[sourceId] || new
                     Float32Array(128);
462               for (let i = 0; i < 128; i++) inputBuffer[i] +=
                     sourceBuffer[i];
463               if (conn.target.startsWith("feedbackDelayNode")) {
464                 const feedbackNodeId = conn.target.split(".")[0];
465                 if (!feedbackBuffers[feedbackNodeId]) {
466                   feedbackBuffers[feedbackNodeId] = new Float32Array
                         (128);
467                 }
468                 for (let i = 0; i < 128; i++) {
469                   feedbackBuffers[feedbackNodeId][i] += sourceBuffer
                         [i];
470                 }
471               }
472             });
473             state.cvConnections.filter((conn) => conn.target.split("
                   .")[0] === id).forEach((conn) => {
474               const modSourceId = conn.source.split(".")[0];
475               if (!visited.has(modSourceId)) {
476                 processNode(modSourceId);
477               }
478               const modBuffer = signalBuffers[modSourceId] || new
                     Float32Array(128);
479               const param = conn.param;
480               if (!node.modulatedParams.hasOwnProperty(param)) {
481                 console.warn(`Parameter ${param} not found for node
                       ${id}`);
482                 return;
483               }
484               let modValue = 0;
```

```
485                 for (let i = 0; i < 128; i++) {
486                   modValue += modBuffer[i];
487                 }
488                 modValue /= 128;
489                 node.modulatedParams[param] = modValue;
490               });
491               // process CV modulation inputs
492               const getEffectiveParam = (node2, param, attenuverter)
                     => {
493                 const base = node2.baseParams[param] || 0;
494                 let modulated = node2.modulatedParams[param] || 0;
495                 if (attenuverter) {
496                   modulated = modulated * attenuverter;
497                 }
498                 const result = base + modulated;
499                 if (typeof result !== "number" || isNaN(result)) {
500                   console.warn(`Invalid value for parameter ${param
                        }:`, result);
501                   return 0;
502                 }
503                 return result;
504               };
505               // handle modules!
506               if (node.node === "Oscillator") {
507                 const effectiveFrequency = getEffectiveParam(node, "
                      frequency", node.baseParams["freq cv +/-"]);
508                 const effectiveGain = getEffectiveParam(node, "gain");
509                 for (let i = 0; i < signalBuffers[id].length; i++) {
510                   node.phase += effectiveFrequency / sampleRate;
511                   if (node.phase >= 1) node.phase -= 1;
512                   switch (node.baseParams["type"]) {
513                     case "sine":
514                       signalBuffers[id][i] = Math.sin(2 * Math.PI *
                            node.phase) * effectiveGain;
515                       break;
516                     case "square":
517                       signalBuffers[id][i] = (node.phase < 0.5 ? 1 :
                            -1) * effectiveGain * Math.SQRT1_2;
518                       break;
519                     case "sawtooth":
520                       signalBuffers[id][i] = (2 * node.phase - 1) *
                            effectiveGain;
521                       break;
522                     case "triangle":
523                       signalBuffers[id][i] = (node.phase < 0.5 ? 4 *
                            node.phase - 1 : 3 - 4 * node.phase) *
                            effectiveGain;
524                       break;
525                     default:
526                       signalBuffers[id][i] = Math.sin(2 * Math.PI *
                            node.phase) * effectiveGain;
```

```
527                 }
528               }
529             } else if (node.node === "LFO2") {
530               const effectiveFrequency = getEffectiveParam(node, "
                      frequency", node.baseParams["freq cv +/-"]);
531               const effectiveGain = getEffectiveParam(node, "gain");
532               for (let i = 0; i < 128; i++) {
533                 node.phase += effectiveFrequency / sampleRate;
534                 if (node.phase >= 1) node.phase -= 1;
535                 node.output.sine[i] = Math.sin(2 * Math.PI * node.
                      phase) * effectiveGain;
536                 node.output.square[i] = (node.phase < 0.5 ? 1 : -1)
                      * effectiveGain;
537                 node.output.saw[i] = (2 * node.phase - 1) *
                      effectiveGain;
538                 node.output.tri[i] = (node.phase < 0.5 ? 4 * node.
                      phase - 1 : 3 - 4 * node.phase) * effectiveGain;
539               }
540             } else if (node.node === "VCA") {
541               const effectiveGain = getEffectiveParam(node, "gain",
                      node.baseParams["gain cv +/-"]);
542               if (!node.dcBlockState) {
543                 node.dcBlockState = { prevInput: 0, prevOutput: 0 };
544               }
545               const cutoffFreq = 5;
546               const alpha = 1 - Math.exp(-2 * Math.PI * cutoffFreq /
                      sampleRate);
547               for (let i = 0; i < 128; i++) {
548                 const inputSample = inputBuffer[i] || 0;
549                 let modulatedGain = (node.modulatedParams["gain"] ||
                        0) * node.baseParams["gain cv +/-"];
550                 const filteredModulation = alpha * (modulatedGain -
                      node.dcBlockState.prevInput) + node.dcBlockState.
                      prevOutput;
551                 node.dcBlockState.prevInput = modulatedGain;
552                 node.dcBlockState.prevOutput = filteredModulation;
553                 const finalGain = this.clamp(effectiveGain +
                      filteredModulation, 0, 1);
554                 signalBuffers[id][i] = inputSample * finalGain;
555               }
556             } else if (node.node === "Gain") {
557               for (let i = 0; i < 128; i++) signalBuffers[id][i] =
                      inputBuffer[i] * node.baseParams.gain;
558             } else if (node.node === "feedbackDelayNode") {
559               if (!node.delayBuffer) {
560                 node.delayBuffer = new Float32Array(128);
561                 node.delayIndex = 0;
562               }
563               for (let i = 0; i < 128; i++) {
564                 const feedbackInput = feedbackBuffers[id]?.[i] || 0;
```

```
565                 const delayIndex = (node.delayIndex - 1 + 128) %
                        128;
566                 const delayedSample = node.delayBuffer[delayIndex]
                        || 0;
567                 signalBuffers[id][i] = delayedSample;
568                 node.delayBuffer[node.delayIndex] = feedbackInput;
569                 node.delayIndex = (node.delayIndex + 1) % 128;
570               }
571             } else if (node.node === "Delay") {
572               let cubicInterpolate = function(buffer, index) {
573                 const len = buffer.length;
574                 let intIndex = Math.floor(index);
575                 let frac = index - intIndex;
576                 const i0 = (intIndex - 1 + len) % len;
577                 const i1 = intIndex % len;
578                 const i2 = (intIndex + 1) % len;
579                 const i3 = (intIndex + 2) % len;
580                 const sample0 = buffer[i0];
581                 const sample1 = buffer[i1];
582                 const sample2 = buffer[i2];
583                 const sample3 = buffer[i3];
584                 const a0 = -0.5 * sample0 + 1.5 * sample1 - 1.5 *
                        sample2 + 0.5 * sample3;
585                 const a1 = sample0 - 2.5 * sample1 + 2 * sample2 -
                        0.5 * sample3;
586                 const a2 = -0.5 * sample0 + 0.5 * sample2;
587                 const a3 = sample1;
588                 return ((a0 * frac + a1) * frac + a2) * frac + a3;
589               };
590               if (!node.delayBuffer) {
591                 node.delayBuffer = new Float32Array(sampleRate);
592                 node.delayIndex = 0;
593                 node.lpfCutoff = node.baseParams.lpfCutoff || 3e3;
594                 node.lpfPreviousSample = 0;
595                 node.currentDelayTime = Math.min(
596                   Math.max(getEffectiveParam(node, "delayTime", node
                          .baseParams["time cv +/-"]), 0),
597                   999
598                 );
599                 node.targetDelayTime = node.currentDelayTime;
600                 node.crossfadeActive = false;
601                 node.crossfade = 0;
602                 node.crossfadeSamples = 128;
603                 node.crossfadeIncrement = 1 / node.crossfadeSamples;
604                 node.allpassMem1 = 0;
605                 node.allpassMem2 = 0;
606               }
607               const newDelayTimeParam = Math.min(
608                 Math.max(getEffectiveParam(node, "delayTime", node.
                        baseParams["time cv +/-"]), 0),
609                 999
```

```
610                 );
611                 if (newDelayTimeParam !== node.targetDelayTime &&
                        !node.crossfadeActive) {
612                   node.targetDelayTime = newDelayTimeParam;
613                   node.crossfadeActive = true;
614                   node.crossfade = 0;
615                 }
616                 const RC = 1 / (2 * Math.PI * node.lpfCutoff);
617                 const alpha = sampleRate / (sampleRate + RC);
618                 for (let i = 0; i < 128; i++) {
619                   let delayedSample;
620                   if (node.crossfadeActive) {
621                     const delaySamplesOld = node.currentDelayTime / 1
                            e3 * sampleRate;
622                     const delaySamplesNew = node.targetDelayTime / 1e3
                            * sampleRate;
623                     let readIndexOld = node.delayIndex -
                            delaySamplesOld;
624                     if (readIndexOld < 0) readIndexOld += node.
                            delayBuffer.length;
625                     let readIndexNew = node.delayIndex -
                            delaySamplesNew;
626                     if (readIndexNew < 0) readIndexNew += node.
                            delayBuffer.length;
627                     const delayedOld = cubicInterpolate(node.
                            delayBuffer, readIndexOld);
628                     const delayedNew = cubicInterpolate(node.
                            delayBuffer, readIndexNew);
629                     delayedSample = (1 - node.crossfade) * delayedOld
                            + node.crossfade * delayedNew;
630                     node.crossfade += node.crossfadeIncrement;
631                     if (node.crossfade >= 1) {
632                       node.currentDelayTime = node.targetDelayTime;
633                       node.crossfadeActive = false;
634                       node.crossfade = 0;
635                     }
636                   } else {
637                     const delaySamples = node.currentDelayTime / 1e3 *
                            sampleRate;
638                     let readIndex = node.delayIndex - delaySamples;
639                     if (readIndex < 0) readIndex += node.delayBuffer.
                            length;
640                     delayedSample = cubicInterpolate(node.delayBuffer,
                            readIndex);
641                   }
642                   let filteredFeedback = alpha * delayedSample + (1 -
                          alpha) * node.lpfPreviousSample;
643                   node.lpfPreviousSample = filteredFeedback;
644                   const feedbackInput = feedbackBuffers[id]?.[i] || 0;
```

```
645                 const feedbackParam = typeof node.baseParams.
                        feedback === "number" ? node.baseParams.feedback
                        : 0.5;
646                 const feedbackSample = filteredFeedback *
                        feedbackParam + feedbackInput;
647                 const inputSample = inputBuffer[i] || 0;
648                 const wetMix = Math.min(Math.max(getEffectiveParam(
                        node, "dryWet", node.baseParams["dryWet cv +/-"])
                        , 0), 1);
649                 const dryMix = 1 - Math.min(Math.max(
                        getEffectiveParam(node, "dryWet", node.baseParams
                        ["dryWet cv +/-"]), 0), 1);
650                 const wetSignal = delayedSample + feedbackSample;
651                 const drySignal = inputSample;
652                 signalBuffers[id][i] = this.clamp(drySignal * dryMix
                        + wetSignal * wetMix, -1, 1);
653                 const feedbackAmount = Math.min(
654                   Math.max(getEffectiveParam(node, "feedback", node.
                          baseParams["feedback cv +/-"]), 0),
655                   1
656                 ) || 0.3;
657                 const feedbackSignal = this.clamp(drySignal +
                        wetSignal * feedbackAmount, -1, 1);
658                 node.delayBuffer[node.delayIndex] = feedbackSignal;
659                 node.delayIndex = (node.delayIndex + 1) % node.
                        delayBuffer.length;
660               }
661             } else if (node.node === "HighPassFilter") {
662               if (!node.coefficients) {
663                 node.coefficients = { a0: 0, a1: 0, a2: 0, b0: 0, b1
                        : 0, b2: 0 };
664                 node.inputHistory1 = [0, 0];
665                 node.outputHistory1 = [0, 0];
666                 node.inputHistory2 = [0, 0];
667                 node.outputHistory2 = [0, 0];
668               }
669               const effectiveFreq = this.clamp(
670                 getEffectiveParam(node, "freq", node.baseParams["
                        freq cv +/-"]),
671                 80,
672                 1e4
673               );
674               const effectiveQ = this.clamp(
675                 getEffectiveParam(node, "Q", node.baseParams["Q cv
                        +/-"]),
676                 0.1,
677                 20
678               );
679               const omega = 2 * Math.PI * effectiveFreq / sampleRate
                      ;
680               const alpha = Math.sin(omega) / (2 * effectiveQ);
```

```
681                 let b0, b1, b2, a0, a1, a2;
682                 b0 = (1 + Math.cos(omega)) / 2;
683                 b1 = -(1 + Math.cos(omega));
684                 b2 = (1 + Math.cos(omega)) / 2;
685                 a0 = 1 + alpha;
686                 a1 = -2 * Math.cos(omega);
687                 a2 = 1 - alpha;
688                 b0 /= a0;
689                 b1 /= a0;
690                 b2 /= a0;
691                 a1 /= a0;
692                 a2 /= a0;
693                 node.coefficients = { b0, b1, b2, a1, a2 };
694                 for (let i = 0; i < 128; i++) {
695                   const inputSample = inputBuffer[i];
696                   let filtered1 = node.coefficients.b0 * inputSample +
                          node.coefficients.b1 * node.inputHistory1[0] +
                         node.coefficients.b2 * node.inputHistory1[1] -
                         node.coefficients.a1 * node.outputHistory1[0] -
                         node.coefficients.a2 * node.outputHistory1[1];
697                   node.inputHistory1[1] = node.inputHistory1[0];
698                   node.inputHistory1[0] = inputSample;
699                   node.outputHistory1[1] = node.outputHistory1[0];
700                   node.outputHistory1[0] = filtered1;
701                   let filtered2 = node.coefficients.b0 * filtered1 +
                         node.coefficients.b1 * node.inputHistory2[0] +
                         node.coefficients.b2 * node.inputHistory2[1] -
                         node.coefficients.a1 * node.outputHistory2[0] -
                         node.coefficients.a2 * node.outputHistory2[1];
702                   node.inputHistory2[1] = node.inputHistory2[0];
703                   node.inputHistory2[0] = filtered1;
704                   node.outputHistory2[1] = node.outputHistory2[0];
705                   node.outputHistory2[0] = filtered2;
706                   signalBuffers[id][i] = filtered2;
707                 }
708               } else if (node.node === "Pulses") {
709                 const effectiveTempo = getEffectiveParam(node, "tempo"
                      , node.baseParams["tempo cv +/-"]);
710                 const stepDuration = 60 / effectiveTempo * sampleRate;
711                 const effectivePulseWidth = Math.min(Math.max(
                     getEffectiveParam(node, "pulseWidth"), 0), 0.999);
712                 const pulseSamples = effectivePulseWidth > 0 ? Math.
                     max(1, Math.round(effectivePulseWidth *
                     stepDuration)) : 0;
713                 node.clockPhase = node.clockPhase || 0;
714                 node.stepIndex = node.stepIndex || 0;
715                 node.pulseCounter = node.pulseCounter || 0;
716                 for (let i = 0; i < 128; i++) {
717                   node.clockPhase += 1;
718                   if (node.clockPhase >= stepDuration) {
719                     node.clockPhase = 0;
```

```
720                     node.stepIndex = (node.stepIndex + 1) % node.
                            baseParams.stepCount;
721                     node.pulseCounter = pulseSamples;
722                   }
723                   signalBuffers[id][i] = node.pulseCounter > 0 ? 1 :
                        0;
724                   if (node.pulseCounter > 0) {
725                     node.pulseCounter -= 1;
726                   }
727                 }
728               } else if (node.node === "Euclid") {
729                 let generateEuclideanPattern = function(pulses, steps)
                        {
730                   let pattern2 = [];
731                   let bucket = 0;
732                   for (let i = 0; i < steps; i++) {
733                     bucket += pulses;
734                     if (bucket >= steps) {
735                       bucket -= steps;
736                       pattern2.push(1);
737                     } else {
738                       pattern2.push(0);
739                     }
740                   }
741                   return pattern2;
742                 };
743                 const effectiveTempo = getEffectiveParam(node, "tempo"
                        , node.baseParams["tempo cv +/-"]);
744                 const stepDuration = 60 / effectiveTempo * sampleRate;
745                 const effectivePulseWidth = Math.min(Math.max(
                        getEffectiveParam(node, "ratchet"), 0), 1);
746                 const pulseSamples = Math.max(1, Math.round(sampleRate
                        * 5e-4));
747                 const numStepsParam = Math.max(1, Math.floor(
                        getEffectiveParam(node, "numSteps")));
748                 const activeStepsParam = Math.max(0, Math.min(
                        numStepsParam, Math.floor(getEffectiveParam(node, "
                        activeSteps"))));
749                 const pattern = generateEuclideanPattern(
                        activeStepsParam, numStepsParam);
750                 const effectiveRatchet = Math.max(1, Math.floor(
                        getEffectiveParam(node, "ratchet", node.baseParams[
                        "ratchet cv +/-"])));
751                 let ratchetInterval = stepDuration;
752                 let ratchetPulseSamples = pulseSamples;
753                 if (effectiveRatchet > 1) {
754                   ratchetInterval = stepDuration / effectiveRatchet;
755                   ratchetPulseSamples = effectivePulseWidth > 0 ? Math
                          .max(1, Math.round(effectivePulseWidth *
                          ratchetInterval)) : pulseSamples;
756                 }
```

```
757               node.clockPhase = node.clockPhase || 0;
758               node.stepIndex = node.stepIndex || 0;
759               node.currentStepActive = pattern[node.stepIndex] ===
                      1;
760               for (let i = 0; i < 128; i++) {
761                 node.clockPhase += 1;
762                 if (node.clockPhase >= stepDuration) {
763                   node.clockPhase -= stepDuration;
764                   node.stepIndex = (node.stepIndex + 1) %
                          numStepsParam;
765                   node.currentStepActive = pattern[node.stepIndex]
                          === 1;
766                 }
767                 let outputVal = 0;
768                 if (node.currentStepActive) {
769                   if (effectiveRatchet > 1) {
770                     const subphase = node.clockPhase %
                            ratchetInterval;
771                     if (subphase < ratchetPulseSamples) {
772                       outputVal = 1;
773                     }
774                   } else {
775                     if (node.clockPhase < pulseSamples) {
776                       outputVal = 1;
777                     }
778                   }
779                 }
780                 signalBuffers[id][i] = outputVal;
781               }
782             } else if (node.node === "Mixer") {
783               const gainA = node.baseParams["gainA"];
784               const gainB = node.baseParams["gainB"];
785               const sourceA = node.inputA;
786               const sourceB = node.inputB;
787               const bufferA = signalBuffers[node.inputA] || new
                      Float32Array(128);
788               const bufferB = signalBuffers[node.inputB] || new
                      Float32Array(128);
789               if (!signalBuffers[id]) {
790                 signalBuffers[id] = new Float32Array(128);
791               }
792               for (let i = 0; i < 128; i++) {
793                 signalBuffers[id][i] = bufferA[i] * gainA + bufferB[
                        i] * gainB;
794               }
795             }
796           };
797           state.outputConnections.forEach((id) => processNode(id.
                  split(".")[0]));
798         };
799
```

```
800           processGraph(this.currentState, this.signalBuffersCurrent,
                  this.feedbackBuffers);
801           if (this.nextState.outputConnections) processGraph(this.
                  nextState, this.signalBuffersNext, this.feedbackBuffers);
802           for (let i = 0; i < output.length; i++) {
803             let currentSample = 0;
804             let nextSample = 0;
805             if (this.currentState.outputConnections.length > 0) {
806               const currentOutputId = this.currentState.
                      outputConnections[0].split(".")[0];
807               currentSample = this.signalBuffersCurrent[
                      currentOutputId]?.[i] || 0;
808             }
809             if (this.nextState.outputConnections.length > 0) {
810               const nextOutputId = this.nextState.outputConnections
                      [0].split(".")[0];
811               nextSample = this.signalBuffersNext[nextOutputId]?.[i]
                      || 0;
812             }
813             output[i] = this.crossfadeInProgress ? (1 - this.
                    crossfadeProgress) * currentSample + this.
                    crossfadeProgress * nextSample : currentSample;
814             output[i] *= this.outputVolume;
815           }
816           if (this.analyze === true) {
817             let sumSq = 0;
818             for (let i = 0; i < output.length; i++) {
819               sumSq += output[i] * output[i];
820             }
821             // log rms if player has activate it
822             const rms = Math.sqrt(sumSq / output.length);
823             this.analyzerFrameCount = (this.analyzerFrameCount || 0) +
                    1;
824             if (this.analyzerFrameCount % 100 === 0) {
825               this.port.postMessage({
826                 cmd: "analyzerData",
827                 rms
828               });
829             }
830           }
831           return true;
832       }
833   };
834   registerProcessor("DSP", DSP);
```

Listing F.1: Custom DSP code running in an AudioWorklet

Appendix G

Module Specifications

```
1
2  {
3    "AudioDestination": {
4      "parameters": {},
5      "paramNames": [],
6      "cv": {},
7      "inputs": ["IN"],
8      "outputs": []
9    },
10   "feedbackDelayNode": {
11     "parameters": {},
12     "paramNames": [],
13     "cv": {},
14     "inputs": ["IN"],
15     "outputs": ["OUT"]
16   },
17   "Gain": {
18     "parameters": {
19       "gain": {
20         "min": 0,
21         "max": 1,
22         "default": 1,
23         "units": "linear gain",
24         "description": "Controls the amount of amplification applied
               to the input signal.",
25         "modulatable": true,
26         "modulationDescription": "Audio-rate modulation of gain to
              create effects like tremolo or dynamic volume changes."
27       }
28     },
29     "paramNames": ["gain"],
30     "cvNames": ["gain"],
31     "cv": {
32       "gain": {
```

```
        "min": 0,
        "max": 1,
        "default": 1,
        "units": "linear gain",
        "description": "Controls the amount of amplification applied to the input signal.",
        "modulatable": true,
        "modulationDescription": "Audio-rate modulation of gain to create effects like tremolo or dynamic volume changes."
      }
    },
    "inputs": ["IN"],
    "outputs": ["OUT"]
  },
  "VCA": {
    "parameters": {
      "gain": {
        "min": 0,
        "max": 1,
        "default": 1,
        "units": "linear gain",
        "description": "Sets the base gain applied to the input signal.",
        "modulatable": true,
        "modulationDescription": "Audio-rate modulation of gain for tremolo or dynamic volume changes."
      },
      "gain cv +/-": {
        "min": 0,
        "max": 1,
        "default": 0.5,
        "units": "linear scale",
        "description": "Attenuates the effect of the CV input on gain modulation.",
        "modulatable": false,
        "modulationDescription": "N/A"
      }
    },
    "paramNames": ["gain", "gain cv +/-"],
    "cvNames": ["gain"],
    "cv": {
      "gain": {
        "min": 0,
        "max": 1,
        "default": 0,
        "units": "linear gain",
        "description": "Control Voltage input for dynamic gain modulation.",
        "modulatable": true,
        "modulationDescription": "Audio-rate modulation to control gain dynamically."
```

```
      }
    },
    "inputs": ["IN"],
    "outputs": ["OUT"]
  },

  "HighPassFilter": {
    "parameters": {
      "freq": {
        "min": 80,
        "max": 10000,
        "default": 1000,
        "units": "Hz",
        "description": "The cutoff frequency of the high-pass filter.",
        "modulatable": true,
        "modulationDescription": "Modulating the cutoff frequency allows for dynamic filtering effects."
      },
      "Q": {
        "min": 0.1,
        "max": 20,
        "default": 10,
        "units": "resonance",
        "description": "The resonance or quality factor of the filter.",
        "modulatable": true,
        "modulationDescription": "Higher Q values emphasize frequencies around the cutoff point."
      },
      "freq cv +/-": {
        "min": 80,
        "max": 10000,
        "default": 1000,
        "units": "Hz",
        "description": "Control Voltage input for dynamically modulating the cutoff frequency.",
        "modulatable": true,
        "modulationDescription": "Allows for sweeps or envelope-controlled filtering."
      },
      "Q cv +/-": {
        "min": 0.1,
        "max": 20,
        "default": 1,
        "units": "resonance",
        "description": "Control Voltage input for dynamically modulating the resonance.",
        "modulatable": true,
        "modulationDescription": "Modulating the Q value changes how sharply the filter emphasizes the cutoff frequency."
```

```
120         }
121       },
122       "paramNames": ["freq", "Q", "freq cv +/-", "Q cv +/-"],
123       "cvNames": ["freq", "Q"],
124       "cv": {
125         "freq": {
126           "min": 80,
127           "max": 10000,
128           "default": 1000,
129           "units": "Hz",
130           "description": "Control Voltage input for dynamically
                modulating the cutoff frequency.",
131           "modulatable": true,
132           "modulationDescription": "Allows for sweeps or envelope-
                controlled filtering."
133         },
134         "Q": {
135           "min": 0.1,
136           "max": 20,
137           "default": 10,
138           "units": "resonance",
139           "description": "Control Voltage input for dynamically
                modulating the resonance.",
140           "modulatable": true,
141           "modulationDescription": "Modulating the Q value changes how
                 sharply the filter emphasizes the cutoff frequency."
142         }
143       },
144       "inputs": ["IN"],
145       "outputs": ["OUT"]
146     },
147
148     "Oscillator": {
149       "parameters": {
150         "frequency": {
151           "min": 110,
152           "max": 3000,
153           "default": 440,
154           "units": "Hz",
155           "description": "The frequency of the oscillator.",
156           "modulatable": true,
157           "modulationDescription": "Audio-rate modulation of the
                frequency to create vibrato or FM synthesis effects."
158         },
159         "type": {
160           "values": ["sine", "square", "sawtooth", "triangle"],
161           "default": "sine",
162           "description": "The waveform shape of the oscillator."
163         },
164         "freq cv +/-": {
165           "min": -1000,
```

```
        "max": 1000,
        "default": 100,
        "units": "Hz",
        "description": "The attenuverter of the frequency CV input",
        "modulatable": false,
        "modulationDescription": "The attenuverter of the frequency CV input"
      }
    },
    "paramNames": ["frequency", "freq cv +/-", "type"],
    "cvNames": ["frequency"],
    "cv": {
      "frequency": {
        "min": 110,
        "max": 9000,
        "default": 440,
        "units": "Hz",
        "description": "The frequency of the oscillator.",
        "modulatable": true,
        "modulationDescription": "Audio-rate modulation of the frequency to create vibrato or FM synthesis effects."
      }
    },
    "inputs": [],
    "outputs": ["OUT"]
  },
  "Delay": {
    "parameters": {
      "delayTime": {
        "min": 0.01,
        "max": 999,
        "default": 500,
        "units": "milliseconds",
        "description": "The delay time applied to the input signal.",
        "modulatable": true,
        "modulationDescription": "Audio-rate modulation of the delay time. Connect an AudioNode to modulate this parameter."
      },
      "time cv +/-": {
        "min": -1000,
        "max": 1000,
        "default": 100,
        "units": "Hz",
        "description": "The attenuverter of the delayTime CV input",
        "modulatable": false,
        "modulationDescription": "The attenuverter of the delayTime CV input"
      },
      "feedback": {
        "min": 0,
```

```
212           "max": 1,
213           "default": 0.5,
214           "units": "float",
215           "description": "feedback into the delay buffer",
216           "modulatable": true,
217           "modulationDescription": "feedback into the delay buffer"
218         },
219         "feedback cv +/-": {
220           "min": -1,
221           "max": 1,
222           "default": 0,
223           "units": "float",
224           "description": "attenuverter for the feedback into the delay
                buffer",
225           "modulatable": true,
226           "modulationDescription": "attenuverter for the dfeedback
                into the delay buffer"
227         },
228         "dryWet": {
229           "min": 0,
230           "max": 1,
231           "default": 0.5,
232           "units": "float",
233           "description": "dry/wet ratio of the output signal",
234           "modulatable": true,
235           "modulationDescription": "dry/wet ratio of the output signal
                "
236         },
237         "dryWet cv +/-": {
238           "min": -1,
239           "max": 1,
240           "default": 0,
241           "units": "float",
242           "description": "attenuverter for the dry/wet ratio of the
                output signal",
243           "modulatable": true,
244           "modulationDescription": "attenuverter for the dry/wet ratio
                of the output signal"
245         }
246       },
247       "paramNames": ["delayTime", "time cv +/-", "feedback", "feedback
            cv +/-", "dryWet", "dryWet cv +/-"],
248       "cvNames": ["delayTime", "feedback", "dryWet"],
249       "cv": {
250         "delayTime": {
251           "min": 0,
252           "max": 999,
253           "default": 100,
254           "units": "seconds",
255           "description": "The delay time applied to the input signal."
                ,
```

```
256         "modulatable": true,
257         "modulationDescription": "Audio-rate modulation of the delay
                time. Connect an AudioNode to modulate this parameter."
258       },
259       "feedback": {
260         "min": 0,
261         "max": 1,
262         "default": 0.5,
263         "units": "float",
264         "description": "feedback into the delay buffer",
265         "modulatable": true,
266         "modulationDescription": "feedback into the delay buffer"
267       },
268       "dryWet": {
269         "min": 0,
270         "max": 1,
271         "default": 0.5,
272         "units": "float",
273         "description": "dry/wet ratio of the output signal",
274         "modulatable": true,
275         "modulationDescription": "Audio-rate modulation of the dry/
              wet ratio of the output signal"
276       }
277     },
278     "inputs": ["IN"],
279     "outputs": ["OUT"]
280   },
281
282   "Flutter": {
283     "parameters": {
284       "delayTime": {
285         "min": 0.01,
286         "max": 50,
287         "default": 30,
288         "units": "milliseconds",
289         "description": "The delay time applied to the input signal."
              ,
290         "modulatable": true,
291         "modulationDescription": "Audio-rate modulation of the delay
                time. Connect an AudioNode to modulate this parameter."
292       },
293       "time cv +/-": {
294         "min": -50,
295         "max": 50,
296         "default": 2,
297         "units": "Hz",
298         "description": "The attenuverter of the delayTime CV input",
299         "modulatable": false,
300         "modulationDescription": "The attenuverter of the delayTime
              CV input"
301       },
```

```
302         "feedback": {
303           "min": 0,
304           "max": 1,
305           "default": 0.5,
306           "units": "float",
307           "description": "feedback into the delay buffer",
308           "modulatable": true,
309           "modulationDescription": "feedback into the delay buffer"
310         },
311         "feedback cv +/-": {
312           "min": -1,
313           "max": 1,
314           "default": 0,
315           "units": "float",
316           "description": "attenuverter for the feedback into the delay
                buffer",
317           "modulatable": true,
318           "modulationDescription": "attenuverter for the dfeedback
                into the delay buffer"
319         },
320         "dryWet": {
321           "min": 0,
322           "max": 1,
323           "default": 0.5,
324           "units": "float",
325           "description": "dry/wet ratio of the output signal",
326           "modulatable": true,
327           "modulationDescription": "dry/wet ratio of the output signal
                "
328         },
329         "dryWet cv +/-": {
330           "min": -1,
331           "max": 1,
332           "default": 0,
333           "units": "float",
334           "description": "attenuverter for the dry/wet ratio of the
                output signal",
335           "modulatable": true,
336           "modulationDescription": "attenuverter for the dry/wet ratio
                of the output signal"
337         }
338       },
339       "paramNames": ["delayTime", "time cv +/-", "feedback", "feedback
            cv +/-", "dryWet", "dryWet cv +/-"],
340       "cvNames": ["delayTime", "feedback", "dryWet"],
341       "cv": {
342         "delayTime": {
343           "min": 0.01,
344           "max": 50,
345           "default": 30,
346           "units": "milliseconds",
```

```
347         "description": "The delay time applied to the input signal."
                ,
348         "modulatable": true,
349         "modulationDescription": "Audio-rate modulation of the delay
                time. Connect an AudioNode to modulate this parameter."
350       },
351       "feedback": {
352         "min": 0,
353         "max": 1,
354         "default": 0.5,
355         "units": "float",
356         "description": "feedback into the delay buffer",
357         "modulatable": true,
358         "modulationDescription": "feedback into the delay buffer"
359       },
360       "dryWet": {
361         "min": 0,
362         "max": 1,
363         "default": 0.5,
364         "units": "float",
365         "description": "dry/wet ratio of the output signal",
366         "modulatable": true,
367         "modulationDescription": "Audio-rate modulation of the dry/
                wet ratio of the output signal"
368       }
369     },
370     "inputs": ["IN"],
371     "outputs": ["OUT"]
372   },
373
374
375   "LFO": {
376     "parameters": {
377       "frequency": {
378         "min": 0.01,
379         "max": 10,
380         "default": 1,
381         "units": "Hz",
382         "description": "The frequency of the low frequency
                oscillator.",
383         "modulatable": true,
384         "modulationDescription": "Audio-rate modulation of the
                frequency to create vibrato or FM synthesis effects."
385       },
386       "type": {
387         "values": ["sine", "square", "sawtooth", "triangle"],
388         "default": "sine",
389         "description": "The waveform shape of the oscillator."
390       },
391       "freq cv +/-": {
392         "min": -20,
```

```
        "max": 20,
        "default": 10,
        "units": "Hz",
        "description": "The attenuverter of the frequency CV input",
        "modulatable": false,
        "modulationDescription": "The attenuverter of the frequency
          CV input"
      }
    },
    "paramNames": ["frequency", "freq cv +/-", "type"],
    "cvNames": ["frequency"],
    "cv": {
      "frequency": {
        "min": 0.01,
        "max": 10,
        "default": 5,
        "units": "Hz",
        "description": "The frequency of the oscillator.",
        "modulatable": true,
        "modulationDescription": "Audio-rate modulation of the
          frequency to create vibrato or FM synthesis effects."
      }
    },
    "inputs": [],
    "outputs": ["OUT"]
  },

  "Pulses": {
    "parameters": {
      "stepCount": {
        "min": 1,
        "max": 16,
        "default": 8,
        "units": "steps",
        "description": "Number of steps in the sequence.",
        "modulatable": false
      },
      "tempo": {
        "min": 20,
        "max": 300,
        "default": 120,
        "units": "BPM",
        "description": "Controls the speed of the sequence.",
        "modulatable": true,
        "modulationDescription": "Can be modulated for tempo
          variations."
      },
      "tempo cv +/-": {
        "min": -100,
        "max": 100,
        "default": 10,
```

```
        "units": "BPM",
        "description": "The attenuverter of the tempo CV input",
        "modulatable": false,
        "modulationDescription": "The attenuverter of the tempo CV
           input"
      },
      "pulseWidth": {
        "min": 0.05,
        "max": 1,
        "default": 0.5,
        "units": "%",
        "description": "Width of the gate relative to the tempo",
        "modulatable": true,
        "modulationDescription": "Can be modulated for dynamic gate
           timing."
      }
    },
    "paramNames": ["stepCount", "tempo", "tempo cv +/-", "pulseWidth
       "],
    "cvNames": ["tempo", "pulseWidth"],
    "cv": {
      "tempo": {
        "min": 20,
        "max": 300,
        "default": 120,
        "units": "BPM",
        "description": "Modulation input for tempo control.",
        "modulatable": true
      },
      "pulseWidth": {
        "min": 0.05,
        "max": 1,
        "default": 0.5,
        "units": "%",
        "description": "Width of the gate relative to the tempo",
        "modulatable": true
      }
    },
    "inputs": [],
    "outputs": ["OUT"]
  },

  "Euclid": {
    "parameters": {
      "numSteps": {
        "min": 1,
        "max": 16,
        "default": 8,
        "units": "steps",
        "description": "Number of available steps in the sequence.",
        "modulatable": false
```

```
      },
      "activeSteps": {
        "min": 1,
        "max": 16,
        "default": 5,
        "units": "steps",
        "description": "Number of active steps in the sequence.",
        "modulatable": false
      },
      "tempo": {
        "min": 20,
        "max": 300,
        "default": 120,
        "units": "BPM",
        "description": "Controls the speed of the sequence.",
        "modulatable": true,
        "modulationDescription": "Can be modulated for tempo
           variations."
      },
      "tempo cv +/-": {
        "min": -100,
        "max": 100,
        "default": 10,
        "units": "BPM",
        "description": "The attenuverter of the tempo CV input",
        "modulatable": false,
        "modulationDescription": "The attenuverter of the tempo CV
           input"
      },
      "ratchet": {
        "min": 0,
        "max": 16,
        "default": 0,
        "units": "steps",
        "description": "Number of ratchets per active step.",
        "modulatable": false
      },
      "ratchet cv +/-": {
        "min": -16,
        "max": 16,
        "default": 0,
        "units": "steps",
        "description": "Attenuverter of number of ratchets per
           active step.",
        "modulatable": false
      }
    },
    "paramNames": ["numSteps", "activeSteps", "tempo", "tempo cv +/-
       ", "ratchet", "ratchet cv +/-"],
    "cvNames": ["tempo", "ratchet"],
    "cv": {
```

```
536         "tempo": {
537           "min": 20,
538           "max": 300,
539           "default": 120,
540           "units": "BPM",
541           "description": "Modulation input for tempo control.",
542           "modulatable": true
543         },
544         "ratchet": {
545           "min": 0,
546           "max": 16,
547           "default": 0,
548           "units": "steps",
549           "description": "modulation of the ratchets",
550           "modulatable": true
551         }
552       },
553       "inputs": [],
554       "outputs": ["OUT"]
555     },
556     "Mixer": {
557       "parameters": {
558         "gainA": {
559           "min": 0,
560           "max": 1,
561           "default": 1,
562           "units": "linear",
563           "description": "Gain level for input A.",
564           "modulatable": true,
565           "modulationDescription": "Allows modulation of input A's
                amplitude."
566         },
567         "gainB": {
568           "min": 0,
569           "max": 1,
570           "default": 1,
571           "units": "linear",
572           "description": "Gain level for input B.",
573           "modulatable": true,
574           "modulationDescription": "Allows modulation of input B's
                amplitude."
575         }
576       },
577       "paramNames": ["gainA", "gainB"],
578       "cvNames": [],
579       "cv": { },
580       "inputs": ["IN_A", "IN_B"],
581       "outputs": ["OUT"]
582     },
583     "SLOWFO": {
584       "parameters": {
```

```
585         "frequency": {
586           "min": 0.001,
587           "max": 1,
588           "default": 0.001,
589           "units": "Hz",
590           "description": "The frequency of the low frequency
                oscillator.",
591           "modulatable": true,
592           "modulationDescription": "Audio-rate modulation of the
                frequency to create vibrato or FM synthesis effects."
593         },
594         "type": {
595           "values": ["sine", "square", "sawtooth", "triangle"],
596           "default": "sine",
597           "description": "The waveform shape of the oscillator."
598         },
599         "freq cv +/-": {
600           "min": -1,
601           "max": 1,
602           "default": 0.002,
603           "units": "Hz",
604           "description": "The attenuverter of the frequency CV input",
605           "modulatable": false,
606           "modulationDescription": "The attenuverter of the frequency
                CV input"
607         }
608       },
609       "paramNames": ["frequency", "freq cv +/-", "type"],
610       "cvNames": ["frequency"],
611       "cv": {
612         "frequency": {
613           "min": 0.001,
614           "max": 1,
615           "default": 0.02,
616           "units": "Hz",
617           "description": "The frequency of the oscillator.",
618           "modulatable": true,
619           "modulationDescription": "Audio-rate modulation of the
                frequency to create vibrato or FM synthesis effects."
620         }
621       },
622       "inputs": [],
623       "outputs": ["OUT"]
624     }
625
626
627
628
629 }
```

Listing G.1: Specifications for modules written for Forking Paths. Used by the DSP AudioWorklet, the Synth Designer, and patching interface in the main synth window.

Appendix H

.fpsynth File Data Structure


```
1
2  {
3    "filename": "demo.fpsynth",
4    "visualGraph": {
5      "elements": {
6        "nodes": [
7          {
8            "data": {
9              "id": "AudioDestination_Rabbit_8bc1f09d7627",
10             "label": "AudioDestination Rabbit",
11             "kind": "module",
12             "rnboName": "AudioDestination",
13             "moduleSpec": {
14               "parameters": {},
15               "paramNames": [],
16               "cv": {},
17               "inputs": [
18                 "IN"
19               ],
20               "outputs": []
21             },
22             "structure": "webAudioNodes",
23             "bgcolour": "#E6CCFF"
24           },
25           "position": {
26             "x": 610,
27             "y": 554.5
28           },
29           "group": "nodes",
30           "removed": false,
31           "selected": false,
32           "selectable": true,
33           "locked": false,
34           "grabbable": true,
```

```
      "pannable": false,
      "classes": ":parent"
    },
    {
      "data": {
        "id": "AudioDestination_Rabbit_8bc1f09d7627.IN",
        "parent": "AudioDestination_Rabbit_8bc1f09d7627",
        "label": "IN",
        "kind": "input",
        "bgcolour": "#FC9A4F",
        "ghostCableShape": "rectangle",
        "ghostCableColour": "#5C9AE3",
        "description": null
      },
      "position": {
        "x": 610,
        "y": 564
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "pannable": false,
      "classes": ""
    },
    {
      "data": {
        "id": "LFO_Owl_59397a2b32bf",
        "label": "LFO Owl",
        "kind": "module",
        "rnboName": "LFO",
        "moduleSpec": {
          "parameters": {
            "frequency": {
              "min": 0.01,
              "max": 10,
              "default": 1,
              "units": "Hz",
              "description": "The frequency of the low frequency
                  oscillator.",
              "modulatable": true,
              "modulationDescription": "Audio-rate modulation of
                  the frequency to create vibrato or FM
                 synthesis effects.",
              "kind": "paramAnchorNode",
              "name": "frequency",
              "ui": "knob"
            },
            "type": {
```

```
                "values": [
                  "sine",
                  "square",
                  "sawtooth",
                  "triangle"
                ],
                "default": "sine",
                "description": "The waveform shape of the
                   oscillator.",
                "kind": "paramAnchorNode",
                "name": "type",
                "ui": "menu"
              },
              "freq cv +/-": {
                "min": -20,
                "max": 20,
                "default": 10,
                "units": "Hz",
                "description": "The attenuverter of the frequency
                   CV input",
                "modulatable": false,
                "modulationDescription": "The attenuverter of the
                   frequency CV input",
                "kind": "paramAnchorNode",
                "name": "freq cv +/-",
                "ui": "knob"
              }
            },
            "paramNames": [
              "frequency",
              "freq cv +/-",
              "type"
            ],
            "cvNames": [
              "frequency"
            ],
            "cv": {
              "frequency": {
                "min": 0.01,
                "max": 10,
                "default": 5,
                "units": "Hz",
                "description": "The frequency of the oscillator.",
                "modulatable": true,
                "modulationDescription": "Audio-rate modulation of
                    the frequency to create vibrato or FM
                   synthesis effects.",
                "kind": "input",
                "name": "frequency",
                "ui": "knob"
              }
```

```
            },
            "inputs": [],
            "outputs": [
              "OUT"
            ]
          },
          "structure": "webAudioNodes",
          "bgcolour": "#CCCCCC"
        },
        "position": {
          "x": 162.5,
          "y": 262.5
        },
        "group": "nodes",
        "removed": false,
        "selected": false,
        "selectable": true,
        "locked": false,
        "grabbable": true,
        "pannable": false,
        "classes": ":parent"
      },
      {
        "data": {
          "id": "LFO_Owl_59397a2b32bf_frequency",
          "parent": "LFO_Owl_59397a2b32bf",
          "label": "frequency",
          "nameSpace": "LFO_Owl_59397a2b32bf_frequency.frequency",
          "kind": "paramAnchorNode",
          "shape": "ellipse",
          "bgcolour": "#CCCCCC",
          "hash": "LFO_Owl_59397a2b32bf_frequency",
          "ui": "knob",
          "min": 0.01,
          "max": 10,
          "value": 1,
          "menuOptions": "none",
          "description": "The frequency of the low frequency
             oscillator.",
          "units": "Hz"
        },
        "position": {
          "x": 82,
          "y": 172
        },
        "group": "nodes",
        "removed": false,
        "selected": false,
        "selectable": true,
        "locked": false,
        "grabbable": true,
```

```
      "pannable": false,
      "classes": "paramAnchorNode"
    },
    {
      "data": {
        "id": "LFO_Owl_59397a2b32bf_freq cv +/-",
        "parent": "LFO_Owl_59397a2b32bf",
        "label": "freq cv +/-",
        "nameSpace": "LFO_Owl_59397a2b32bf_freq cv +/-.freq cv
          +/-",
        "kind": "paramAnchorNode",
        "shape": "ellipse",
        "bgcolour": "#CCCCCC",
        "hash": "LFO_Owl_59397a2b32bf_freq cv +/-",
        "ui": "knob",
        "min": -20,
        "max": 20,
        "value": 10,
        "menuOptions": "none",
        "description": "The attenuverter of the frequency CV
          input",
        "units": "Hz"
      },
      "position": {
        "x": 82,
        "y": 277
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "pannable": false,
      "classes": "paramAnchorNode"
    },
    {
      "data": {
        "id": "LFO_Owl_59397a2b32bf_type",
        "parent": "LFO_Owl_59397a2b32bf",
        "label": "type",
        "nameSpace": "LFO_Owl_59397a2b32bf_type.type",
        "kind": "paramAnchorNode",
        "shape": "ellipse",
        "bgcolour": "#CCCCCC",
        "hash": "LFO_Owl_59397a2b32bf_type",
        "ui": "menu",
        "min": 0,
        "max": 1,
        "value": "sine",
        "menuOptions": [
```

```
            "sine",
            "square",
            "sawtooth",
            "triangle"
          ],
          "description": "The waveform shape of the oscillator.",
          "units": "unspecified"
        },
        "position": {
          "x": 82,
          "y": 382
        },
        "group": "nodes",
        "removed": false,
        "selected": false,
        "selectable": true,
        "locked": false,
        "grabbable": true,
        "pannable": false,
        "classes": "paramAnchorNode"
      },
      {
        "data": {
          "id": "LFO_Owl_59397a2b32bf.frequency",
          "parent": "LFO_Owl_59397a2b32bf",
          "label": "frequency",
          "kind": "input",
          "bgcolour": "#FC9A4F",
          "ghostCableShape": "rectangle",
          "ghostCableColour": "#5C9AE3",
          "description": "Audio-rate modulation of the frequency
             to create vibrato or FM synthesis effects."
        },
        "position": {
          "x": 222,
          "y": 172
        },
        "group": "nodes",
        "removed": false,
        "selected": false,
        "selectable": true,
        "locked": false,
        "grabbable": true,
        "pannable": false,
        "classes": ""
      },
      {
        "data": {
          "id": "LFO_Owl_59397a2b32bf.OUT",
          "parent": "LFO_Owl_59397a2b32bf",
          "label": "OUT",
```

```
278             "kind": "output",
279             "bgcolour": "#6FB1FC",
280             "ghostCableShape": "triangle",
281             "ghostCableColour": "#E68942",
282             "description": null
283           },
284           "position": {
285             "x": 222,
286             "y": 277
287           },
288           "group": "nodes",
289           "removed": false,
290           "selected": false,
291           "selectable": true,
292           "locked": false,
293           "grabbable": true,
294           "pannable": false,
295           "classes": ""
296         },
297         {
298           "data": {
299             "id": "Oscillator_Snake_3ef7749e23ea",
300             "label": "Oscillator Snake",
301             "kind": "module",
302             "rnboName": "Oscillator",
303             "moduleSpec": {
304               "parameters": {
305                 "frequency": {
306                   "min": 110,
307                   "max": 3000,
308                   "default": 440,
309                   "units": "Hz",
310                   "description": "The frequency of the oscillator.",
311                   "modulatable": true,
312                   "modulationDescription": "Audio-rate modulation of
                        the frequency to create vibrato or FM
                       synthesis effects.",
313                   "kind": "paramAnchorNode",
314                   "name": "frequency",
315                   "ui": "knob"
316                 },
317                 "type": {
318                   "values": [
319                     "sine",
320                     "square",
321                     "sawtooth",
322                     "triangle"
323                   ],
324                   "default": "sine",
325                   "description": "The waveform shape of the
                       oscillator.",
```

```
                "kind": "paramAnchorNode",
                "name": "type",
                "ui": "menu"
              },
              "freq cv +/-": {
                "min": -1000,
                "max": 1000,
                "default": 100,
                "units": "Hz",
                "description": "The attenuverter of the frequency
                    CV input",
                "modulatable": false,
                "modulationDescription": "The attenuverter of the
                    frequency CV input",
                "kind": "paramAnchorNode",
                "name": "freq cv +/-",
                "ui": "knob"
              }
            },
            "paramNames": [
              "frequency",
              "freq cv +/-",
              "type"
            ],
            "cvNames": [
              "frequency"
            ],
            "cv": {
              "frequency": {
                "min": 110,
                "max": 9000,
                "default": 440,
                "units": "Hz",
                "description": "The frequency of the oscillator.",
                "modulatable": true,
                "modulationDescription": "Audio-rate modulation of
                    the frequency to create vibrato or FM
                    synthesis effects.",
                "kind": "input",
                "name": "frequency",
                "ui": "knob"
              }
            },
            "inputs": [],
            "outputs": [
              "OUT"
            ]
          },
          "structure": "webAudioNodes",
          "bgcolour": "#CCCCCC"
        },
```

```
      "position": {
        "x": 438.5,
        "y": 260.5
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "pannable": false,
      "classes": ":parent"
    },
    {
      "data": {
        "id": "Oscillator_Snake_3ef7749e23ea_frequency",
        "parent": "Oscillator_Snake_3ef7749e23ea",
        "label": "frequency",
        "nameSpace": "Oscillator_Snake_3ef7749e23ea_frequency.
           frequency",
        "kind": "paramAnchorNode",
        "shape": "ellipse",
        "bgcolour": "#CCCCCC",
        "hash": "Oscillator_Snake_3ef7749e23ea_frequency",
        "ui": "knob",
        "min": 110,
        "max": 3000,
        "value": 440,
        "menuOptions": "none",
        "description": "The frequency of the oscillator.",
        "units": "Hz"
      },
      "position": {
        "x": 358,
        "y": 170
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "pannable": false,
      "classes": "paramAnchorNode"
    },
    {
      "data": {
        "id": "Oscillator_Snake_3ef7749e23ea_freq cv +/-",
        "parent": "Oscillator_Snake_3ef7749e23ea",
        "label": "freq cv +/-",
```

```
            "nameSpace": "Oscillator_Snake_3ef7749e23ea_freq cv +/-.
               freq cv +/-",
            "kind": "paramAnchorNode",
            "shape": "ellipse",
            "bgcolour": "#CCCCCC",
            "hash": "Oscillator_Snake_3ef7749e23ea_freq cv +/-",
            "ui": "knob",
            "min": -1000,
            "max": 1000,
            "value": 100,
            "menuOptions": "none",
            "description": "The attenuverter of the frequency CV
               input",
            "units": "Hz"
          },
          "position": {
            "x": 358,
            "y": 275
          },
          "group": "nodes",
          "removed": false,
          "selected": false,
          "selectable": true,
          "locked": false,
          "grabbable": true,
          "pannable": false,
          "classes": "paramAnchorNode"
        },
        {
          "data": {
            "id": "Oscillator_Snake_3ef7749e23ea_type",
            "parent": "Oscillator_Snake_3ef7749e23ea",
            "label": "type",
            "nameSpace": "Oscillator_Snake_3ef7749e23ea_type.type",
            "kind": "paramAnchorNode",
            "shape": "ellipse",
            "bgcolour": "#CCCCCC",
            "hash": "Oscillator_Snake_3ef7749e23ea_type",
            "ui": "menu",
            "min": 0,
            "max": 1,
            "value": "sine",
            "menuOptions": [
              "sine",
              "square",
              "sawtooth",
              "triangle"
            ],
            "description": "The waveform shape of the oscillator.",
            "units": "unspecified"
          },
```

```
      "position": {
        "x": 358,
        "y": 380
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "pannable": false,
      "classes": "paramAnchorNode"
    },
    {
      "data": {
        "id": "Oscillator_Snake_3ef7749e23ea.frequency",
        "parent": "Oscillator_Snake_3ef7749e23ea",
        "label": "frequency",
        "kind": "input",
        "bgcolour": "#FC9A4F",
        "ghostCableShape": "rectangle",
        "ghostCableColour": "#5C9AE3",
        "description": "Audio-rate modulation of the frequency
           to create vibrato or FM synthesis effects."
      },
      "position": {
        "x": 498,
        "y": 170
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "pannable": false,
      "classes": ""
    },
    {
      "data": {
        "id": "Oscillator_Snake_3ef7749e23ea.OUT",
        "parent": "Oscillator_Snake_3ef7749e23ea",
        "label": "OUT",
        "kind": "output",
        "bgcolour": "#6FB1FC",
        "ghostCableShape": "triangle",
        "ghostCableColour": "#E68942",
        "description": null
      },
      "position": {
        "x": 498,
```

```
            "y": 275
          },
          "group": "nodes",
          "removed": false,
          "selected": false,
          "selectable": true,
          "locked": false,
          "grabbable": true,
          "pannable": false,
          "classes": ""
        },
        {
          "data": {
            "id": "SLOWFO_Cucumber_dad3fab4af09",
            "label": "SLOWFO Cucumber",
            "kind": "module",
            "rnboName": "SLOWFO",
            "moduleSpec": {
              "parameters": {
                "frequency": {
                  "min": 0.001,
                  "max": 1,
                  "default": 0.001,
                  "units": "Hz",
                  "description": "The frequency of the low frequency
                      oscillator.",
                  "modulatable": true,
                  "modulationDescription": "Audio-rate modulation of
                      the frequency to create vibrato or FM
                     synthesis effects.",
                  "kind": "paramAnchorNode",
                  "name": "frequency",
                  "ui": "knob"
                },
                "type": {
                  "values": [
                    "sine",
                    "square",
                    "sawtooth",
                    "triangle"
                  ],
                  "default": "sine",
                  "description": "The waveform shape of the
                     oscillator.",
                  "kind": "paramAnchorNode",
                  "name": "type",
                  "ui": "menu"
                },
                "freq cv +/-": {
                  "min": -1,
                  "max": 1,
```

```
                    "default": 0.002,
                    "units": "Hz",
                    "description": "The attenuverter of the frequency
                       CV input",
                    "modulatable": false,
                    "modulationDescription": "The attenuverter of the
                       frequency CV input",
                    "kind": "paramAnchorNode",
                    "name": "freq cv +/-",
                    "ui": "knob"
                  }
                },
                "paramNames": [
                  "frequency",
                  "freq cv +/-",
                  "type"
                ],
                "cvNames": [
                  "frequency"
                ],
                "cv": {
                  "frequency": {
                    "min": 0.001,
                    "max": 1,
                    "default": 0.02,
                    "units": "Hz",
                    "description": "The frequency of the oscillator.",
                    "modulatable": true,
                    "modulationDescription": "Audio-rate modulation of
                        the frequency to create vibrato or FM
                       synthesis effects.",
                    "kind": "input",
                    "name": "frequency",
                    "ui": "knob"
                  }
                },
                "inputs": [],
                "outputs": [
                  "OUT"
                ]
              },
              "structure": "webAudioNodes",
              "bgcolour": "#E6CCFF"
            },
            "position": {
              "x": 998.5,
              "y": 261.5
            },
            "group": "nodes",
            "removed": false,
            "selected": false,
```

```
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "pannable": false,
      "classes": ":parent"
    },
    {
      "data": {
        "id": "SLOWFO_Cucumber_dad3fab4af09_frequency",
        "parent": "SLOWFO_Cucumber_dad3fab4af09",
        "label": "frequency",
        "nameSpace": "SLOWFO_Cucumber_dad3fab4af09_frequency.
           frequency",
        "kind": "paramAnchorNode",
        "shape": "ellipse",
        "bgcolour": "#CCCCCC",
        "hash": "SLOWFO_Cucumber_dad3fab4af09_frequency",
        "ui": "knob",
        "min": 0.001,
        "max": 1,
        "value": 0.001,
        "menuOptions": "none",
        "description": "The frequency of the low frequency
           oscillator.",
        "units": "Hz"
      },
      "position": {
        "x": 918,
        "y": 171
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "pannable": false,
      "classes": "paramAnchorNode"
    },
    {
      "data": {
        "id": "SLOWFO_Cucumber_dad3fab4af09_freq cv +/-",
        "parent": "SLOWFO_Cucumber_dad3fab4af09",
        "label": "freq cv +/-",
        "nameSpace": "SLOWFO_Cucumber_dad3fab4af09_freq cv +/-.
           freq cv +/-",
        "kind": "paramAnchorNode",
        "shape": "ellipse",
        "bgcolour": "#CCCCCC",
        "hash": "SLOWFO_Cucumber_dad3fab4af09_freq cv +/-",
        "ui": "knob",
```

```
        "min": -1,
        "max": 1,
        "value": 0.002,
        "menuOptions": "none",
        "description": "The attenuverter of the frequency CV
           input",
        "units": "Hz"
      },
      "position": {
        "x": 918,
        "y": 276
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
      "selectable": true,
      "locked": false,
      "grabbable": true,
      "pannable": false,
      "classes": "paramAnchorNode"
    },
    {
      "data": {
        "id": "SLOWFO_Cucumber_dad3fab4af09_type",
        "parent": "SLOWFO_Cucumber_dad3fab4af09",
        "label": "type",
        "nameSpace": "SLOWFO_Cucumber_dad3fab4af09_type.type",
        "kind": "paramAnchorNode",
        "shape": "ellipse",
        "bgcolour": "#CCCCCC",
        "hash": "SLOWFO_Cucumber_dad3fab4af09_type",
        "ui": "menu",
        "min": 0,
        "max": 1,
        "value": "sine",
        "menuOptions": [
          "sine",
          "square",
          "sawtooth",
          "triangle"
        ],
        "description": "The waveform shape of the oscillator.",
        "units": "unspecified"
      },
      "position": {
        "x": 918,
        "y": 381
      },
      "group": "nodes",
      "removed": false,
      "selected": false,
```

```
            "selectable": true,
            "locked": false,
            "grabbable": true,
            "pannable": false,
            "classes": "paramAnchorNode"
          },
          {
            "data": {
              "id": "SLOWFO_Cucumber_dad3fab4af09.frequency",
              "parent": "SLOWFO_Cucumber_dad3fab4af09",
              "label": "frequency",
              "kind": "input",
              "bgcolour": "#FC9A4F",
              "ghostCableShape": "rectangle",
              "ghostCableColour": "#5C9AE3",
              "description": "Audio-rate modulation of the frequency
                 to create vibrato or FM synthesis effects."
            },
            "position": {
              "x": 1058,
              "y": 171
            },
            "group": "nodes",
            "removed": false,
            "selected": false,
            "selectable": true,
            "locked": false,
            "grabbable": true,
            "pannable": false,
            "classes": ""
          },
          {
            "data": {
              "id": "SLOWFO_Cucumber_dad3fab4af09.OUT",
              "parent": "SLOWFO_Cucumber_dad3fab4af09",
              "label": "OUT",
              "kind": "output",
              "bgcolour": "#6FB1FC",
              "ghostCableShape": "triangle",
              "ghostCableColour": "#E68942",
              "description": null
            },
            "position": {
              "x": 1058,
              "y": 276
            },
            "group": "nodes",
            "removed": false,
            "selected": false,
            "selectable": true,
            "locked": false,
```

```
      "grabbable": true,
      "pannable": false,
      "classes": ""
    },
    {
      "data": {
        "id": "VCA_Skinks_2263572604f5",
        "label": "VCA Skinks",
        "kind": "module",
        "rnboName": "VCA",
        "moduleSpec": {
          "parameters": {
            "gain": {
              "min": 0,
              "max": 1,
              "default": 1,
              "units": "linear gain",
              "description": "Sets the base gain applied to the input signal.",
              "modulatable": true,
              "modulationDescription": "Audio-rate modulation of gain for tremolo or dynamic volume changes.",
              "kind": "paramAnchorNode",
              "name": "gain",
              "ui": "knob"
            },
            "gain cv +/-": {
              "min": 0,
              "max": 1,
              "default": 0.5,
              "units": "linear scale",
              "description": "Attenuates the effect of the CV input on gain modulation.",
              "modulatable": false,
              "modulationDescription": "N/A",
              "kind": "paramAnchorNode",
              "name": "gain cv +/-",
              "ui": "knob"
            }
          },
          "paramNames": [
            "gain",
            "gain cv +/-"
          ],
          "cvNames": [
            "gain"
          ],
          "cv": {
            "gain": {
              "min": 0,
              "max": 1,
```

```
                    "default": 0,
                    "units": "linear gain",
                    "description": "Control Voltage input for dynamic
                       gain modulation.",
                    "modulatable": true,
                    "modulationDescription": "Audio-rate modulation to
                        control gain dynamically.",
                    "kind": "input",
                    "name": "gain",
                    "ui": "knob"
                  }
                },
                "inputs": [
                  "IN"
                ],
                "outputs": [
                  "OUT"
                ]
              },
              "structure": "webAudioNodes",
              "bgcolour": "#E6CCFF"
            },
            "position": {
              "x": 720.75,
              "y": 262.5
            },
            "group": "nodes",
            "removed": false,
            "selected": false,
            "selectable": true,
            "locked": false,
            "grabbable": true,
            "pannable": false,
            "classes": ":parent"
          },
          {
            "data": {
              "id": "VCA_Skinks_2263572604f5_gain",
              "parent": "VCA_Skinks_2263572604f5",
              "label": "gain",
              "nameSpace": "VCA_Skinks_2263572604f5_gain.gain",
              "kind": "paramAnchorNode",
              "shape": "ellipse",
              "bgcolour": "#CCCCCC",
              "hash": "VCA_Skinks_2263572604f5_gain",
              "ui": "knob",
              "min": 0,
              "max": 1,
              "value": 1,
              "menuOptions": "none",
```

```
            "description": "Sets the base gain applied to the input signal.",
            "units": "linear gain"
          },
          "position": {
            "x": 655,
            "y": 167
          },
          "group": "nodes",
          "removed": false,
          "selected": false,
          "selectable": true,
          "locked": false,
          "grabbable": true,
          "pannable": false,
          "classes": "paramAnchorNode"
        },
        {
          "data": {
            "id": "VCA_Skinks_2263572604f5_gain cv +/-",
            "parent": "VCA_Skinks_2263572604f5",
            "label": "gain cv +/-",
            "nameSpace": "VCA_Skinks_2263572604f5_gain cv +/-.gain cv +/-",
            "kind": "paramAnchorNode",
            "shape": "ellipse",
            "bgcolour": "#CCCCCC",
            "hash": "VCA_Skinks_2263572604f5_gain cv +/-",
            "ui": "knob",
            "min": 0,
            "max": 1,
            "value": 0.5,
            "menuOptions": "none",
            "description": "Attenuates the effect of the CV input on gain modulation.",
            "units": "linear scale"
          },
          "position": {
            "x": 655,
            "y": 272
          },
          "group": "nodes",
          "removed": false,
          "selected": false,
          "selectable": true,
          "locked": false,
          "grabbable": true,
          "pannable": false,
          "classes": "paramAnchorNode"
        },
        {
```

```
907             "data": {
908               "id": "VCA_Skinks_2263572604f5.gain",
909               "parent": "VCA_Skinks_2263572604f5",
910               "label": "gain",
911               "kind": "input",
912               "bgcolour": "#FC9A4F",
913               "ghostCableShape": "rectangle",
914               "ghostCableColour": "#5C9AE3",
915               "description": "Audio-rate modulation to control gain
                    dynamically."
916             },
917             "position": {
918               "x": 655,
919               "y": 377
920             },
921             "group": "nodes",
922             "removed": false,
923             "selected": false,
924             "selectable": true,
925             "locked": false,
926             "grabbable": true,
927             "pannable": false,
928             "classes": ""
929           },
930           {
931             "data": {
932               "id": "VCA_Skinks_2263572604f5.IN",
933               "parent": "VCA_Skinks_2263572604f5",
934               "label": "IN",
935               "kind": "input",
936               "bgcolour": "#FC9A4F",
937               "ghostCableShape": "rectangle",
938               "ghostCableColour": "#5C9AE3",
939               "description": null
940             },
941             "position": {
942               "x": 795,
943               "y": 167
944             },
945             "group": "nodes",
946             "removed": false,
947             "selected": false,
948             "selectable": true,
949             "locked": false,
950             "grabbable": true,
951             "pannable": false,
952             "classes": ""
953           },
954           {
955             "data": {
956               "id": "VCA_Skinks_2263572604f5.OUT",
```

```
            "parent": "VCA_Skinks_2263572604f5",
            "label": "OUT",
            "kind": "output",
            "bgcolour": "#6FB1FC",
            "ghostCableShape": "triangle",
            "ghostCableColour": "#E68942",
            "description": null
          },
          "position": {
            "x": 795,
            "y": 272
          },
          "group": "nodes",
          "removed": false,
          "selected": false,
          "selectable": true,
          "locked": false,
          "grabbable": true,
          "pannable": false,
          "classes": ""
        }
      ]
    },
    "style": [
      {
        "selector": "node",
        "style": {
          "background-color": "data(bgcolour)",
          "label": "data(label)",
          "font-size": "18px",
          "width": "40px",
          "height": "40px",
          "color": "rgb(0,0,0)",
          "text-valign": "top",
          "text-halign": "center",
          "text-margin-x": "-10px"
        }
      },
      {
        "selector": "node[bgcolour]",
        "style": {
          "background-color": "data(bgcolour)"
        }
      },
      {
        "selector": "node[kind = \"input\"]",
        "style": {
          "shape": "triangle"
        }
      },
      {
```

```
        "selector": "node[kind = \"output\"]",
        "style": {
          "shape": "rectangle"
        }
      },
      {
        "selector": ":parent",
        "style": {
          "background-opacity": "0.5",
          "padding": "40px",
          "background-color": "data(bgcolour)",
          "border-color": "rgb(245,122,65)",
          "border-width": "1px",
          "label": "data(label)",
          "text-valign": "top",
          "text-halign": "center",
          "color": "rgb(0,0,0)",
          "font-size": "25px",
          "text-margin-y": "-10px"
        }
      },
      {
        "selector": "node.highlighted",
        "style": {
          "border-color": "rgb(34,139,34)",
          "border-width": "10px"
        }
      },
      {
        "selector": "node.ghostNode",
        "style": {
          "background-color": "rgb(255,255,255)",
          "opacity": "0.8",
          "width": "27px",
          "height": "27px",
          "border-width": "0px",
          "label": ""
        }
      },
      {
        "selector": "edge",
        "style": {
          "width": "6px",
          "line-color": "rgb(204,204,204)",
          "target-arrow-shape": "triangle",
          "source-arrow-shape": "triangle",
          "source-arrow-color": "rgb(0,0,0)",
          "target-arrow-color": "rgb(0,0,0)",
          "target-arrow-width": "20px",
          "source-arrow-width": "50px",
          "curve-style": "unbundled-bezier",
```

```
            "control-point-weights": "0.25 0.75",
            "control-point-distances": "20px -20px"
          }
        },
        {
          "selector": "edge.highlighted",
          "style": {
            "line-color": "rgb(255,0,0)",
            "target-arrow-color": "rgb(255,0,0)",
            "source-arrow-color": "rgb(255,0,0)",
            "width": "10px"
          }
        },
        {
          "selector": "edge.connectedEdges",
          "style": {
            "line-color": "rgb(34,139,34)",
            "target-arrow-color": "rgb(34,139,34)",
            "source-arrow-color": "rgb(34,139,34)",
            "width": "6px"
          }
        },
        {
          "selector": ".peerPointer",
          "style": {
            "background-color": "data(colour)",
            "width": "15px",
            "height": "15px",
            "shape": "star",
            "text-valign": "center",
            "text-halign": "right",
            "text-margin-x": "10px"
          }
        },
        {
          "selector": ".paramAnchorNode",
          "style": {
            "background-color": "rgb(245,164,93)",
            "background-opacity": "0",
            "shape": "ellipse",
            "width": "20px",
            "height": "20px",
            "label": "",
            "text-opacity": "0",
            "outline-width": "0px",
            "events": "no"
          }
        }
      ],
      "data": {},
      "zoomingEnabled": true,
```

```
      "userZoomingEnabled": false,
      "zoom": 1,
      "minZoom": 1e-50,
      "maxZoom": 1e+50,
      "panningEnabled": true,
      "userPanningEnabled": false,
      "pan": {
        "x": 0,
        "y": 0
      },
      "boxSelectionEnabled": true,
      "renderer": {
        "name": "canvas"
      }
    },
    "paramUIOverlays": {
      "AudioDestination_Rabbit_8bc1f09d7627": [],
      "LFO_Owl_59397a2b32bf": [
        {
          "containerDiv": {}
        },
        {
          "containerDiv": {}
        },
        {
          "containerDiv": {}
        }
      ],
      "Oscillator_Snake_3ef7749e23ea": [
        {
          "containerDiv": {}
        },
        {
          "containerDiv": {}
        },
        {
          "containerDiv": {}
        }
      ],
      "SLOWFO_Cucumber_dad3fab4af09": [
        {
          "containerDiv": {}
        },
        {
          "containerDiv": {}
        },
        {
          "containerDiv": {}
        }
      ],
      "VCA_Skinks_2263572604f5": [
```

```
      {
        "containerDiv": {}
      },
      {
        "containerDiv": {}
      }
    ]
  },
  "audioGraph": {
    "modules": {
      "AudioDestination_Rabbit_8bc1f09d7627": {
        "type": "AudioDestination",
        "params": {},
        "moduleSpec": {
          "parameters": {},
          "paramNames": [],
          "cv": {},
          "inputs": [
            "IN"
          ],
          "outputs": []
        },
        "nodeIDs": [
          "AudioDestination_Rabbit_8bc1f09d7627",
          "AudioDestination_Rabbit_8bc1f09d7627.IN"
        ],
        "structure": "webAudioNodes"
      },
      "LFO_Owl_59397a2b32bf": {
        "type": "LFO",
        "params": {
          "frequency": 1,
          "freq cv +/-": 10,
          "type": "sine"
        },
        "moduleSpec": {
          "parameters": {
            "frequency": {
              "min": 0.01,
              "max": 10,
              "default": 1,
              "units": "Hz",
              "description": "The frequency of the low frequency
                 oscillator.",
              "modulatable": true,
              "modulationDescription": "Audio-rate modulation of the
                  frequency to create vibrato or FM synthesis
                 effects.",
              "kind": "paramAnchorNode",
              "name": "frequency",
              "ui": "knob"
```

```
      },
      "type": {
        "values": [
          "sine",
          "square",
          "sawtooth",
          "triangle"
        ],
        "default": "sine",
        "description": "The waveform shape of the oscillator.",
        "kind": "paramAnchorNode",
        "name": "type",
        "ui": "menu"
      },
      "freq cv +/-": {
        "min": -20,
        "max": 20,
        "default": 10,
        "units": "Hz",
        "description": "The attenuverter of the frequency CV input",
        "modulatable": false,
        "modulationDescription": "The attenuverter of the frequency CV input",
        "kind": "paramAnchorNode",
        "name": "freq cv +/-",
        "ui": "knob"
      }
    },
    "paramNames": [
      "frequency",
      "freq cv +/-",
      "type"
    ],
    "cvNames": [
      "frequency"
    ],
    "cv": {
      "frequency": {
        "min": 0.01,
        "max": 10,
        "default": 5,
        "units": "Hz",
        "description": "The frequency of the oscillator.",
        "modulatable": true,
        "modulationDescription": "Audio-rate modulation of the frequency to create vibrato or FM synthesis effects.",
        "kind": "input",
        "name": "frequency",
```

```
            "ui": "knob"
          }
        },
        "inputs": [],
        "outputs": [
          "OUT"
        ]
      },
      "nodeIDs": [
        "LFO_Owl_59397a2b32bf",
        "LFO_Owl_59397a2b32bf_frequency-anchorNode",
        "LFO_Owl_59397a2b32bf_freq cv +/--anchorNode",
        "LFO_Owl_59397a2b32bf_type-anchorNode",
        "LFO_Owl_59397a2b32bf.frequency",
        "LFO_Owl_59397a2b32bf.OUT"
      ],
      "structure": "webAudioNodes"
    },
    "Oscillator_Snake_3ef7749e23ea": {
      "type": "Oscillator",
      "params": {
        "frequency": 440,
        "freq cv +/-": 100,
        "type": "sine"
      },
      "moduleSpec": {
        "parameters": {
          "frequency": {
            "min": 110,
            "max": 3000,
            "default": 440,
            "units": "Hz",
            "description": "The frequency of the oscillator.",
            "modulatable": true,
            "modulationDescription": "Audio-rate modulation of the
                frequency to create vibrato or FM synthesis
               effects.",
            "kind": "paramAnchorNode",
            "name": "frequency",
            "ui": "knob"
          },
          "type": {
            "values": [
              "sine",
              "square",
              "sawtooth",
              "triangle"
            ],
            "default": "sine",
            "description": "The waveform shape of the oscillator."
               ,
```

```
            "kind": "paramAnchorNode",
            "name": "type",
            "ui": "menu"
          },
          "freq cv +/-": {
            "min": -1000,
            "max": 1000,
            "default": 100,
            "units": "Hz",
            "description": "The attenuverter of the frequency CV
               input",
            "modulatable": false,
            "modulationDescription": "The attenuverter of the
               frequency CV input",
            "kind": "paramAnchorNode",
            "name": "freq cv +/-",
            "ui": "knob"
          }
        },
        "paramNames": [
          "frequency",
          "freq cv +/-",
          "type"
        ],
        "cvNames": [
          "frequency"
        ],
        "cv": {
          "frequency": {
            "min": 110,
            "max": 9000,
            "default": 440,
            "units": "Hz",
            "description": "The frequency of the oscillator.",
            "modulatable": true,
            "modulationDescription": "Audio-rate modulation of the
                frequency to create vibrato or FM synthesis
               effects.",
            "kind": "input",
            "name": "frequency",
            "ui": "knob"
          }
        },
        "inputs": [],
        "outputs": [
          "OUT"
        ]
      },
      "nodeIDs": [
        "Oscillator_Snake_3ef7749e23ea",
        "Oscillator_Snake_3ef7749e23ea_frequency-anchorNode",
```

```
        "Oscillator_Snake_3ef7749e23ea_freq cv +/--anchorNode",
        "Oscillator_Snake_3ef7749e23ea_type-anchorNode",
        "Oscillator_Snake_3ef7749e23ea.frequency",
        "Oscillator_Snake_3ef7749e23ea.OUT"
      ],
      "structure": "webAudioNodes"
    },
    "SLOWFO_Cucumber_dad3fab4af09": {
      "type": "SLOWFO",
      "params": {
        "frequency": 0.001,
        "freq cv +/-": 0.002,
        "type": "sine"
      },
      "moduleSpec": {
        "parameters": {
          "frequency": {
            "min": 0.001,
            "max": 1,
            "default": 0.001,
            "units": "Hz",
            "description": "The frequency of the low frequency
               oscillator.",
            "modulatable": true,
            "modulationDescription": "Audio-rate modulation of the
                frequency to create vibrato or FM synthesis
               effects.",
            "kind": "paramAnchorNode",
            "name": "frequency",
            "ui": "knob"
          },
          "type": {
            "values": [
              "sine",
              "square",
              "sawtooth",
              "triangle"
            ],
            "default": "sine",
            "description": "The waveform shape of the oscillator."
               ,
            "kind": "paramAnchorNode",
            "name": "type",
            "ui": "menu"
          },
          "freq cv +/-": {
            "min": -1,
            "max": 1,
            "default": 0.002,
            "units": "Hz",
```

```
            "description": "The attenuverter of the frequency CV
               input",
            "modulatable": false,
            "modulationDescription": "The attenuverter of the
               frequency CV input",
            "kind": "paramAnchorNode",
            "name": "freq cv +/-",
            "ui": "knob"
          }
        },
        "paramNames": [
          "frequency",
          "freq cv +/-",
          "type"
        ],
        "cvNames": [
          "frequency"
        ],
        "cv": {
          "frequency": {
            "min": 0.001,
            "max": 1,
            "default": 0.02,
            "units": "Hz",
            "description": "The frequency of the oscillator.",
            "modulatable": true,
            "modulationDescription": "Audio-rate modulation of the
                frequency to create vibrato or FM synthesis
               effects.",
            "kind": "input",
            "name": "frequency",
            "ui": "knob"
          }
        },
        "inputs": [],
        "outputs": [
          "OUT"
        ]
      },
      "nodeIDs": [
        "SLOWFO_Cucumber_dad3fab4af09",
        "SLOWFO_Cucumber_dad3fab4af09_frequency-anchorNode",
        "SLOWFO_Cucumber_dad3fab4af09_freq cv +/--anchorNode",
        "SLOWFO_Cucumber_dad3fab4af09_type-anchorNode",
        "SLOWFO_Cucumber_dad3fab4af09.frequency",
        "SLOWFO_Cucumber_dad3fab4af09.OUT"
      ],
      "structure": "webAudioNodes"
    },
    "VCA_Skinks_2263572604f5": {
      "type": "VCA",
```

```
        "params": {
          "gain": 1,
          "gain cv +/-": 0.5
        },
        "moduleSpec": {
          "parameters": {
            "gain": {
              "min": 0,
              "max": 1,
              "default": 1,
              "units": "linear gain",
              "description": "Sets the base gain applied to the
                 input signal.",
              "modulatable": true,
              "modulationDescription": "Audio-rate modulation of
                 gain for tremolo or dynamic volume changes.",
              "kind": "paramAnchorNode",
              "name": "gain",
              "ui": "knob"
            },
            "gain cv +/-": {
              "min": 0,
              "max": 1,
              "default": 0.5,
              "units": "linear scale",
              "description": "Attenuates the effect of the CV input
                 on gain modulation.",
              "modulatable": false,
              "modulationDescription": "N/A",
              "kind": "paramAnchorNode",
              "name": "gain cv +/-",
              "ui": "knob"
            }
          },
          "paramNames": [
            "gain",
            "gain cv +/-"
          ],
          "cvNames": [
            "gain"
          ],
          "cv": {
            "gain": {
              "min": 0,
              "max": 1,
              "default": 0,
              "units": "linear gain",
              "description": "Control Voltage input for dynamic gain
                  modulation.",
              "modulatable": true,
```

```
                "modulationDescription": "Audio-rate modulation to
                    control gain dynamically.",
                "kind": "input",
                "name": "gain",
                "ui": "knob"
              }
            },
            "inputs": [
              "IN"
            ],
            "outputs": [
              "OUT"
            ]
          },
          "nodeIDs": [
            "VCA_Skinks_2263572604f5",
            "VCA_Skinks_2263572604f5_gain-anchorNode",
            "VCA_Skinks_2263572604f5_gain cv +/--anchorNode",
            "VCA_Skinks_2263572604f5.gain",
            "VCA_Skinks_2263572604f5.IN",
            "VCA_Skinks_2263572604f5.OUT"
          ],
          "structure": "webAudioNodes"
        }
      },
      "connections": []
    }
}
```

Listing H.1: A Forking Paths `.fpsynth` file, created within the Synth Designer and able to be loaded into the Synth View

Appendix I

Configuration File

```

export const config = {
    UI:{
        moduleLayout: {
            maxRows: 3,
            cellWidth: 120,
            cellHeight: 100,
            horizontalPadding: 20,
            verticalPadding: 5
        },
        knob:{
            baseKnobSize: 60,
            thickness: 0.3,
            labelFontSize: '19px',
            labelMarginBottom: '3px',
            labelAlign: 'center',
            labelColour: 'black'
        },
        selectMenu: {

        }
    },
    indexedDB:{
        saveInterval: 1000 // how frequently to store the automerge
            document in indexedDB (ms)
    },
    patchHistory:{
        firstBranchName: 'main'
    },
    appCommunication:{
        throttleInterval: 250 // limit throughput to history
            sequencer
    },
    cytoscape:{
```

```
33          synthGraph:{
34              style: {
35                  nodeWidth: 40,
36                  nodeHeight: 40,
37                  parentNode: {
38                      textColour: 'black',
39                      fontSize: 25
40                  },
41                  edge:{
42                      arrowSize: 30
43                  }
44              }
45          },
46          historyGraph:{
47              optimizations: {
48                  hideEdgesOnViewport: false,
49                  textureOnViewPort: false // set to true only if
                        dealing with very, very large graphs. I needed
                        this before I was condensing gestures to a single
                         node
50              }
51          }
52      },
53      audio: {
54          initialVolume: 0.4 // this is overrided by localStorage
                after user changes it for the first time
55      }
56  };
```

Listing I.1: Forking Paths configuration file