SOVA: A LIBRE / OPEN SOURCE POLYGLOT AND COLLABORATIVE LIVE CODING ENVIRONMENT FOR PEDAGOGY AND RESEARCH ON LIVE CODING

Raphaël Maurice Forment
Université Jean Monnet,
Laboratoire ECLLA

Tanguy Dubois
Nantes Université,
École Centrale de Nantes,
CNRS, LS2N, UMR 6004

Loïg Jezequel
Nantes Université,
École Centrale de Nantes,
CNRS, LS2N, UMR 6004

Résumé

Sova (Coba) est un environnement de programmation libre et *open source* (licence AGPL 3.0) pensé pour la pratique collaborative et polyglotte du *live coding* musical. Implémenté en Rust, Sova se compose d'une machine virtuelle dédiée à la création de langages musicaux événementiels, d'une interface client/serveur, d'un moteur de synthèse sonore et d'échantillonnage, et de plusieurs interfaces utilisateur (GUI et TUI). Sova est également capable de synchronisation entre pairs sur le réseau local via le protocole Ableton Link. Quatre langages de programmation musicale conçus à des fins de démonstration illustrent la polyvalence de l'architecture : Bali, Bob, Boinx et Cagire. Sova est le fruit d'un projet de recherche-création initié par l'Athénor CNCM entre Raphaël Forment et le laboratoire LS2N de l'Université de Nantes (2025). Le projet a été au cœur d'une initiative de médiation art-sciences portée par l'Athénor CNCM au sein de plusieurs établissements scolaires de la région Pays de la Loire.

Abstract

Sova (Coba) is a free and open-source programming environment (AGPL 3.0 licensed [8]) designed for collaborative and multi-language live coding. Implemented in Rust and leveraging the Ableton Link synchronization protocol, Sova comprises a virtual machine dedicated to hosting event-based musical programming languages, a client/server communication layer, a dedicated audio engine capable of synthesis and creative sampling, and multiple user-facing graphical interfaces (GUI & TUI). Four musical programming languages — Bali, Bob, Boinx, and Cagire — each adopting a different programming paradigm, have been built to illustrate the versatility of the architecture. Sova is the result of a research-creation project established by Athénor CNCM between Raphaël Forment and the LS2N laboratory at Université de Nantes (2025). The project has been at the heart of an art-science outreach program run across several high-schools in the Pays de la Loire region.

1. INTRODUCTION

Live coding, the practice of writing and performing with code in real time to generate visual and musical performances, typically with the performer's screen projected for the audience, has emerged as a significant field of research in computer music [5] and the arts [11]. Although the practice predates its formal naming, the founding of TOPLAP[1] in 2004 marked a pivotal moment of collective self-organization soon followed by the rise of the *Algorave* movement[2] [4] in the early 2010s which brought live coding into clubs, festival settings and into popular culture. On the academic front, the *International Conference on Live Coding* (ICLC), held with regularity since 2015, has consolidated an interdisciplinary research community comprehensively documented by Blackwell et al. [3]. A defining trait of live coding is that practitioners tend to blur the boundaries between composition, performance, and instrument-making [7, 16]. Rooted in free software culture and open to DIY practices, the community has long valued the creation of bespoke systems [3]. In this context, code and software are not merely seen as a means to produce a result but are themselves considered as aesthetic materials and expressive mediums.

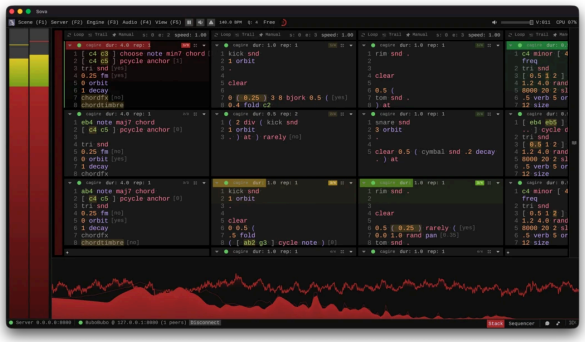

Figure 1. Sova user interface (`sova-frontend`) playing an internal demo file, in its latest iteration (May 2026).

They are projected and performed as an integral part of the creative act. As a consequence, the choice and design of one's tools and languages becomes a deeply personal matter. Each environment is seen as capable to shape

[1] TOPLAP (Temporal Organisation for the Promotion of Live Algorithm Programming) was established in Hamburg in 2004. See the organisation's wiki: https://toplap.org/. Last consulted: February 20, 2026.

[2] The term *algorave* was coined in 2011 by Nick Collins and Alex McLean. The first self-proclaimed algorave was held in London as a warm-up event for the SuperCollider Symposium in 2012.

how a musician thinks about, conceptualize and expresses music [12]. Over the past two decades, a rich ecosystem of dedicated environments has matured to support this diversity of approaches — from education-oriented platforms such as Sonic Pi [1] to deeply specialized pattern languages such as TidalCycles [14] and its web-based port Strudel [20]. Yet, despite this wealth of tools, the ability to rapidly design custom languages and interfaces — shaped to fit specific creative visions or pedagogical contexts — remained difficult to achieve with existing software and frameworks.

2. THE ORIGINS OF SOVA

Pedagogy has long been a concern of the live coding community. Sonic Pi [1, 2] was designed from the ground up to teach programming in schools through music. ChucK [24, 26], with its roots in laptop orchestras, has a well-established tradition of pedagogical use in academic settings. Gibber [18] and EarSketch [25] have been deployed in contexts ranging from middle-school summer camps to university ensembles. In each of these cases, however, the students receive a ready-made environment — language, working paradigm, libraries — and the pedagogical effort centres on learning to use it. Comparatively little attention has been given to involving learners in the design of the tools themselves: shaping the languages, extending the environment or adapting its behaviour to their own musical practice and setup. This dimension of the practice — digital lutherie, instrument building — is typically cared for by advanced practitioners. Yet live coding, as a practice rooted in hacker ethics [10] and open-source culture [3, 6], carries with it the premise that the performer understands and can modify the software. It follows that learning to live code does not need to stop at mastering a given tool but can extend to participating in its design — reading, questioning, and contributing to the software that underpins the performance environment.

In winter 2024, a collaboration between Athénor CNCM and the LS2N provided the opportunity to work on live coding with high school students, with the goal of preparing for a public networked live performance (Figure 2). The objective was for students — most of whom had little or no prior programming experience — to engage in live coding to its full extent after only a few sessions. Rather than introducing a single language, we sought to involve the students in selecting or shaping a language suited to their sensibility with our help to extend it further in-between sessions. It became apparent, once the students were introduced to the ethics and principles that helped to define live coding, that they could themselves contribute to the design of what a suitable live coding language might be. Our goal thus shifted to creating the necessary framework for students to be able to rapidly be involved in the design of new languages and iterate over their own ideas.

We therefore decided to develop a new platform for live coding subject to the following constraints:

Figure 2. Performance rehearsal with high-school students, (May 7th, Athénor, Saint-Nazaire). Photography credits: Hélène Desrues.

- usable for live performance (i.e. with precise timing) and supporting collaborative live coding, as the students would have to perform and play together;
- self-contained and able to run on lower-end computers, shipping everything needed to live code out of the box, including a built-in audio engine (Section 6) so as to minimise external dependencies and simplify deployment in any school;
- allowing rapid integration of new languages since we would need to incorporate the students' ideas from one session to the next, typically within less than one month between sessions;
- enabling students to produce music quickly and enjoyably from the outset, while offering enough depth for those who wish to explore further.

Thus began the development of Sova. Sova architecture aims to facilitate the integration and cohabitation of various live coding languages and user interfaces on the top of the same runtime, while providing flexible inputs and outputs (MIDI / OSC, etc). We present its current state and illustrate what it makes possible for musicians and researchers interested by live coding, generative / algorithmic music making and music language design.

3. THE MODULAR ARCHITECTURE OF SOVA

The constraints outlined in the previous section — precise timing, collaborative networked performance, rapid

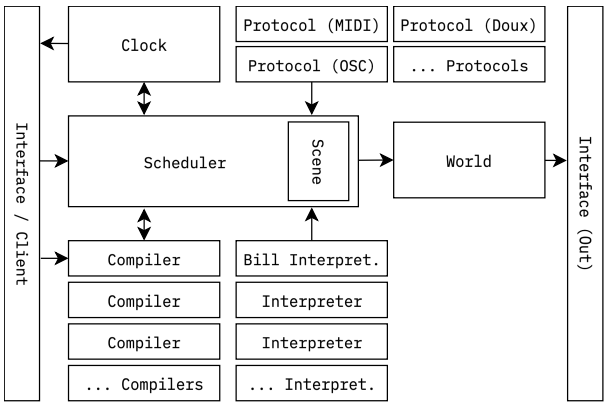

Figure 3. Software architecture overview. On the left, clients and user interfaces. On the right, output devices.

language prototyping, and self-contained deployment on lower-end hardware — call for an architecture that cleanly separates the temporal execution engine from the languages it hosts. Sova addresses this through a modular design (Figure 3) in which each major component of the system (live coding languages, user interfaces, a built-in audio engine, and communication protocols) can be developed, replaced or extended independently. Sova is built as an ecosystem of software modules coupled to a robust and thorough language design and execution system. In stark contrast with the current popularity of web-based live coding platforms, Sova is distributed as a static independent binary programmed in Rust that does not rely on a network connection to run at full capability, suitable for classroom settings with limited connectivity.

Languages made for Sova can either be compiled to bytecode or interpreted directly. They plug into the system through a common interface. They receive timing information and shared state from the scheduler, and produce timestamped events destined for output devices: hardware units, creative software and softsynths. This separation is what enables Sova's polyglot capability: because the execution engine is language-agnostic, adding a new language amounts to implementing a single compiler or interpreter module without modifying the core infrastructure. Similar experiments at building live coding environments centered on language design have already been carried out by Graham Wakefield and Charlie Roberts although with different goals [23].

At its core, Sova relies on two dedicated threads, with a shared *clock* synchronized across the network via the Ableton Link protocol [9] that provides a common temporal reference between them:

- The *scheduler* runs slightly ahead of real time, stepping through scripts written in any supported language and producing timestamped events.
- The *world* receives these events and dispatches them to the appropriate devices (audio engines, MIDI ports, OSC endpoints) at appropriate times. Events can be *immediate* — meaning that they are dispatched at their exact timestamp — or timed, meaning that they are dispatched a bit ahead of their timestamp so that the target device can handle them at the desired time.

Used as a server, this pipeline from clients inputs through a central scheduled execution to a unique real-time output is what allows multiple users to live code simultaneously using various bespoke languages while the system safely maintains internal timing guarantees. The *scheduler*, the *world* and the *clock* are the core components that underpin the capabilities offered by Sova. They are not exposed to users and cannot be modified in direct contrast with every other software component that can be altered or modified freely. In particular, one can:

- create new live coding languages through the addition of compilers and/or interpreters (Section 3.1);
- target any kind of hardware or software device by implementing the appropriate protocols (Section 3.2);
- plug their own user interface by complying with the client/server interfaces defined by Sova (Section 3.3).

3.1. Crafting live coding languages

Languages can be introduced to Sova in a few different ways: by implementing a Rust `Interpreter` or `Compiler` (compiling to bytecode). More experimentally, one can wrap an external binary with the special interpreter/compiler helpers that ship with the project. Regardless of the approach, each language must be registered with the central language registry and may optionally implement a Rust `Language` to supply syntax and documentation metadata. The simplest and most common path is to write Rust code implementing one of the two core traits. All default languages are stored in the `langs/` crate on Sova's repository.

The `Interpreter` feature basically requires to implement an `execute_next` function that executes a few lines of a program in the new language and returns the effects of these lines on the world: side effects, scheduler actions, etc. The `Compiler` feature requires to implement a `compile` function that transforms a program in the new language into a program in our internal low level language called SAIL (*Sova ASM Internal Language*). The resulting SAIL program is then interpreted by the bundled SAIL interpreter, effectively serving as a tiny virtual machine in the style of Java's JVM [22]. SAIL is low-level by design, drawing heavy inspiration from assembly languages[3].

Finally, when adding a language, one should implement the `Language` feature, making it possible to define syntax highlighting and documentation which will then be automatically integrated into consuming interfaces. Sova's languages are managed by a `LanguageCenter` structure, containing a `Transcoder` and an `InterpreterDirectory`. Compiled languages must be added beforehand to the

[3] https://github.com/sova-org/Sova/blob/main/core/src/vm/control_asm.rs gives a list of the assembly-like instructions available in SAIL and https://github.com/sova-org/Sova/blob/main/core/src/vm/event.rs gives a list of the instructions that have effects on the world.

Transcoder, while interpreted ones need to be added to the InterpreterDirectory.

3.2. I/O devices and protocol communication

Interaction between Sova and hardware or digital peripherals is handled through a DeviceMap. The goal of the DeviceMap is to manage connections to devices, assign slots addressable by musicians, but most importantly: to translate internal events into protocol messages. Each protocol usable with Sova (MIDI, OSC, etc) has to be implemented. This allows the scheduler to be able to form correct messages to devices, as the translation is done based on the targeted device type. Generated messages are then self-contained and annotated with a pointer to their targeted device. This also allows the world to know how to communicate with these devices without querying the DeviceMap. A few standard protocols are currently handled. It is up to the user to add any protocol required for a performance, in principle including DMX, serial connections, custom lighting rigs, etc. Only the most versatile protocols have been implemented for now, in accordance with our constraint of rapid prototyping and deployment in educational contexts (Section 2). DeviceMap configurations can be saved and restored, allowing to store complex configurations that form a performer's setup. In a remote networked session, each musician can configure its own DeviceMap, allowing to either centralize the setup on one computer or let each musician decide how to dispatch device messages locally.

3.3. User interface coupling

Sova was conceived from the outset as a library: any Rust project can depend on it and drive the scheduler via inter-thread channels, receiving status notifications in return. In parallel a lightweight TCP server was implemented, exposing the identical control API over a socket-based protocol and effectively offering the same functionality via remote procedure calls. This dual interface — an in-process crate plus a networked daemon — permits an external process to spawn a Sova instance, issue commands, and poll its state using a simple serialization format. The server emits structured messages that clients decode locally to update their views, encapsulating timing and event data in an application-level protocol. This client/server architecture greatly simplifies the construction of user interfaces (see Section 7) and decouples front-ends from the core execution engine. Optionally, Sova is also capable of sending *presence indicators* (cursor position, compilation visual feedback, chat messages) to remote peers, allowing one musician to form a better mental state of other musicians actions and of Sova's server state during a remote jam session.

4. SCHEDULER BEHAVIOR AND DETAILS

The scheduler is the central component of Sova, on which all other subsystems depend. Its internal clock runs roughly 30ms ahead of real time so that events are precomputed and timestamped before they need to fire; these messages are then dispatched by the *world* thread with sub-millisecond precision via a priority queue and active polling. The scheduler organizes execution around objects called *scenes*, *lines* and *frames*, a sequencer-like grid immediately familiar to electronic-music practitioners, yet equally suited to live coding, where scripts are edited mid-performance, and to offline algorithmic composition, where a scene is authored, stored, and rendered without real-time interaction[4].

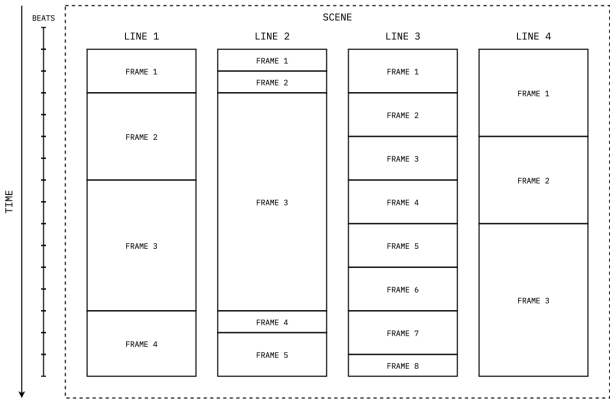

Figure 4. Diagram of Sova's scene model with Frames and Lines hosting scripts written in various languages.

4.1. Data model: Frames, Lines, Scenes (and scripts)

A *scene* (Figure 4) is split into one or more *lines*, each line being constituted of one or more *frames*. Each of these frames is associated to a *script*: a program in any live coding language supported by Sova. In order to execute its scene, the scheduler concurrently executes all the lines of this scene. The execution of a *line* consists in the sequential execution of all its *frames*. Finally, executing a *frame* consists in starting an execution of its *script* after compilation and/or interpretation.

4.2. Durations

Each frame carries an explicit duration; a line's total length is the sum of its frames' durations. When and how lines restart is governed by the scene's execution mode, of which three exist: *Free* — each line loops independently when its own duration elapses; *AtQuantum* — all lines resynchronize at the next beat-grid quantum boundary provided by Ableton Link; *LongestLine* — all lines wait for the longest one to complete before restarting together. A line may additionally operate in *trailing* mode, where a new execution begins while the previous one continues, producing overlapping layers of code execution. Frame durations serve a different purpose: each frame occupies a span of time within its line, and the next frame fires once the previous frame's duration has elapsed. A frame's execution is instantaneous — it merely launches the cor-

[4] Each client ultimately decides how to render the scene on screen, leading to vastly different interfaces, ranging from standard timeline arrangements to tracker-like and/or more creative layouts.

responding script, whose actual running time depends entirely on the program it contains.

4.3. Scripts

A script pairs a text source with a designated language and is bound to exactly one frame. When a user submits or updates a script, the change can be applied immediately or deferred to the next beat or quantum boundary. In either case, any running execution of the previous version continues undisturbed; it is only at the next frame trigger that the old executions are discarded and the new code takes effect, ensuring glitch-free transitions during live performance. Once started, a script runs until it terminates. All the running scripts are executed concurrently by the scheduler, following a simple round-robin algorithm: the scheduler executes one *computation step* of each active script execution in turn. Because a frame can be re-triggered before its current execution finishes, multiple executions of the same script may coexist and are all serviced by the round-robin. For a compiled language running on the virtual machine, a computation step consists of executing up to a configurable batch of control instructions (16 by default), yielding immediately upon the first effect instruction (i.e. an instruction that produces an event). For an interpreted language, the interpreter itself decides what constitutes one step through the common `Interpreter` trait. This batching ensures that scripts make meaningful progress per round while guaranteeing that no single execution can live-lock the scheduler.

Each computation step yields one of four outcomes:

- *event emission*: the appropriate protocol builds a time-stamped message and forwards it to the world;
- *silent progress*: the script advanced its internal state without producing an observable effect;
- *idle request*: the script suspends for a specified duration, during which all other scripts continue to run;
- *termination*: the script has reached its end and is removed from the execution pool.

5. VARIABLES AND EVALUATION CONTEXT

Programs run by Sova share access to internal data structures that allow inter-program communication (1), manage execution context (2) and/or emit musical events (3). Sova internally uses typed variables — both primitive and composite — so that new languages can be developed without exposing classical pointers or dynamic allocation. The supported types are: integers, floating-point numbers, high-precision decimals, booleans, strings, durations (representable as microseconds, beats, or fractions of the current frame length), dynamic vectors, dynamic maps with string keys, first-class functions, binary blobs, and generators (values that evolve over time according to a configurable waveform). Conversions and standard operators are defined for each type. Variables are partitioned into four scopes: *global* (shared across all scripts), *line*, *frame*, and *instance* (local to a single script execution).

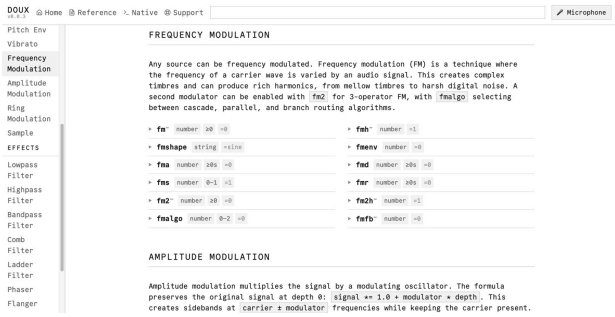

Figure 5. Screen capture of the audio engine demonstration website (WebAssembly version), available at: https://doux.livecoding.fr. Last consulted: February 27, 2026.

5.1. Evaluation context

Each script execution receives an evaluation context that allows it to interact with its environment. This context exposes:

- the current logic-time;
- variables at all four scopes: global, line, frame and instance as well as environment values;
- a double-ended stack local to the execution;
- the shared clock for conversions between time units;
- the indices of the current line and frame, together with their iteration counts;
- the length of the frame in beats;
- the structure of the scene and the device map.

Errors raised during execution are collected rather than causing a crash, providing the user with a diagnostic that can be displayed accordingly in software clients.

5.2. Events

In order to achieve maximal modularity and be as protocol-agnostic as possible, events that can be triggered from scripts are either specific to protocols (in order to perform complex actions) or generic. Most things can be done with generic events that are then translated internally into protocol messages by the `DeviceMap`. The same event type could then be used for internal engines, MIDI or OSC devices for instance, without the script needing to know which protocol will be used.

6. AUDIO ENGINE

Sova development started with no perceived need for built-in synthesis, signal processing or sampling. The rationale was that audio duties could be handled by external software such as SuperCollider or Pure Data targeted as external devices. However, the imperative to produce a self-contained piece of software that could be readily deployed across a diverse fleet of machines gradually lead to design a lightweight and autonomous audio engine. To that end, Sova drew on the research carried out by Felix Roos toward the design of a new audio engine for Strudel [20]. Over the course of 2025, contributors to the Strudel project began crafting a self-contained engine tailored

```
/sound/saw/note/48~60:0.5/delay/0.5
/delaytime/0.1/delayfb/0.8/coarse/12
/gain/1/width/2/release/8
```

Listing 1. Example of a command message sent to the Doux engine for configuring a fixed-duration synthesis voice featuring delay, sample depth reduction, stereo width panning and pitch audio-rate modulation.

for live coding written in the C programming language called Dough[5], capable of running both natively and in the browser as a WebAssembly module. This engine was subsequently ported to Rust by Raphaël before being extended and tailored to meet the specific constraints of the Sova project. The result of this initiative is available to download as Doux[6] (Figure 5), drawn as a dependency by the Sova project. Doux has been gradually extended to offer more specialized sound design capabilities for users running the native version.

6.1. General architecture

By embedding synthesis, sampling and creative DSP effects directly into a single binary/library, Doux addresses the self-containment constraint from Section 2, eliminating the dependency on an external sound server. Unlike dynamically assembled node-graph engines such as SuperCollider [13], Doux employs a fixed-topology signal chain: every voice traverses an identical processing pipeline where unused stages are bypassed, eliminating graph traversal and dynamic dispatch from the audio thread. Each voice is a stereo signal chain that sequentially couples a sound generation module with various processing modules such as filters, distortion units, per-voice effects, amplitude envelope (DAHDSR), stereo panning, etc. These voices are then routed to one of eight stereo *orbit* buses carrying effects shared by voices such as delay, reverb, comb filter and feedback units. The audio path thus processes sample blocks without heap allocations, in an efficient manner. Any synth voice configuration can be speedily described using an OSC-like message scheme as shown in Listing 1, forming a lightweight audio-engine control DSL suitable for spawning and/or controlling existing audio voices.

6.2. Sound generation and modulation

Sound sources include a collection of band-limited and non band-limited oscillators with a phase-shaping pipeline, additive and 3-OP frequency modulation synthesis, white/pink/brown noise sources, seven synthesized drum types, lazy-loaded sample playback via a lock-free registry with optional wavetable synthesis capabilities. Experimental support for soundfonts (via `.sf2` files) is also currently being tested, allowing users to use soundfonts as raw materials to transform. Live input (microphones, etc) and live samping with optional overdub is also being tested, allowing musicians to generate, use and overwrite audio samples live. We do believe that the thorough timbral palette offered by the engine suffices for musically complete performances without relying on more complex — and versatile — graph-based audio engines.

Each voice carries its own voice level effect chain before being summed into its assigned *orbit* audio bus. Orbit audio busses each apply time sensitive effects such as delay (standard, ping-pong, tape, or multitap), reverb (switchable between a Dattorro plate algorithm and a feedback delay network), comb filter, and modulable feedback delay. Per-voice effects shape individual timbres; orbit effects provide shared spatial processing across all voices routed to the same bus. Any numerical parameter across both layers can additionally be modulated at audio rate through an inline mini-language embedded directly in parameter values (e.g. `200~4000:2` oscillates between two bounds over a given period), collapsing the traditional modulation matrix into the parameter specification itself.

Doux is a standalone library. The bridge crate `doux-sova` converts Sova's key–value event payloads into Doux command strings, mapping microsecond timetags to engine-local time via the shared clock. Within Sova, Doux registers as a `ProtocolDevice`, sharing the same dispatch path as MIDI and OSC. Adding the engine required no modification to the *scheduler* or *world*. Doux has already been used pedagogically and creatively in another software (Cagire[7]) for an intensive one-week long live coding workshop at the Ecole Nationale Supérieure des Arts Décoratifs (Paris, PSL) in March 2026 and for an *Algorave* in May 2026. It also comes with a standalone REPL, used for testing, and with a non realtime CLI usable to generate sound samples as `.wav` files.

7. INTERFACES

The separation between Sova's core and its user-facing layer means that interfaces are not privileged components. They are ordinary clients that either embed the core library directly or connect through the server's API, enabling both solo and networked operation[8]. Designing these interfaces presents a particular challenge. Experienced live coders expect rich, feedback-driven editors that tightly couple code representation with musical structure [15, 19] and playback state. Because the performer's screen is typically projected for the audience, the interface is itself part of the performance and its visual identity (fonts, color schemes, layout) must be customizable to reflect each artist's stage presence. High school students encountering

[5] See the project page on Codeberg: https://codeberg.org/uzu/dough or the demo website: https://dough.strudel.cc. Last seen: February 21, 2026.

[6] See the dedicated website: https://doux.livecoding.fr and the GitHub repository: https://github.com/sova-org/doux. Last consulted: February 21, 2026.

[7] See the dedicated website about this project, derived from implementation work done on Sova: https://cagire.raphaelforment.fr. Last consulted: March 15, 2026.

[8] Client/server architectures are well established in the live coding field. SuperCollider's separation of language and synthesis server [13] remains unrivalled in flexibility and is the most influential example of this pattern.

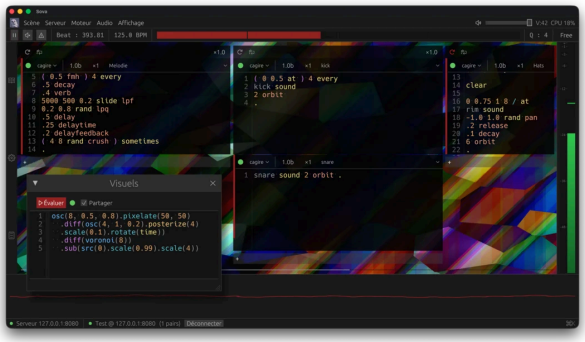

Figure 6. Live session using `sova-frontend` GUI (March 14th). On screen: the scene view, multiple code editors, various widgets (oscilloscope, VUMeter, etc).

Figure 7. `solo-tui` main view displaying the current `Scene` using a grid-like organizational pattern.

programming for the first time, by contrast, will require an environment that is immediately approachable and that uses sensible defaults.

In both cases, users must navigate a system whose components — scheduler, audio engine, devices, languages — are deeply interlinked. The interface must expose this complexity without overwhelming or hindering musical action. Beginners should be able to produce music quickly while receiving useful and immediate feedback to guide them. Advanced users, on the opposite side, must retain full control and customization capabilities. In practice, this calls for domain-specific code editors and domain-specific clients that borrow familiar affordances from general-purpose programming IDEs (syntax highlighting, code completion, search) while specializing them for the live coding context. Real-time transport controls, scene visualization, audio monitoring and tight integration with the scheduler and audio engine are as useful as the traditional debugger panel found in a conventional IDEs.

Multiple interfaces have been developed in order to make Sova usable by musicians. Each tool is conceived to address a different level of access / detail and user feedback in the Sova ecosystem:

- (1) `sova-server`: a command line interface for managing Sova server instances with a rich flag configuration system. Used mainly as tooling for testing and/or deploying jam sessions on local networks.
- (2) `solo-tui`: a terminal user interface (TUI) dedicated to running as close as possible to Sova's internals without any network stack.
- (3) `sova-frontend`: a general purpose client / server binary that contains all of Sova's components and can be used as a graphical client and/or server interface. It is aimed to be used by almost all musicians interested by Sova.

7.1. Command-line server interface

The `sova-server` binary is a headless server accessible through the command line in a terminal. It exposes a rich flag-based configuration interface covering network, timing, and audio engine / resources settings. A single invocation command can fully capture a particular running configuration. The server exposes the full Sova API: any number of clients can connect simultaneously, each sending editing commands and receiving real-time state updates. In a classroom or workshop setting, this client can be used to deploy a server on a central machine handling audio synthesis and device routing while students connect from lightweight clients that need not ship the audio engine or manage hardware. Only the graphical interface and a network connection are required to participate in a collaborative session. The server is also available as a Rust library crate[9], allowing third-party applications to embed a Sova instance programmatically.

7.2. Solo Terminal User Interface (TUI)

For situations where a graphical desktop is unnecessary, Sova provides a standalone terminal user interface built with the Ratatui framework[10] (Figure 7). The TUI bypasses the server entirely. It embeds both the core library and the audio engine, and does not require any networking layer. `solo-tui` is particularly suited to lower-end machines and/or experienced users. The interface is organized into navigable pages: a scene view renders lines and frames on a canvas, highlighting active playback positions in real time. A code editor provides line numbers, undo/redo, clipboard integration, and per-frame language selection. A configuration page exposes a save/load mechanism, a switch for execution mode, and per-line looping and trailing controls. Additional pages display the `DeviceMap`, server logs, and shared variables. All interaction is keyboard-driven, with transport controls accessible from any page.

7.3. Graphical User Interface

The primary interface for both performance and pedagogy is `sova-frontend`, a desktop application built

[9] In the Rust ecosystem, a *crate* is a compilation unit that can be shared and reused as a package. See https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html. Last consulted: February 23, 2026.

[10] Ratatui is a Rust library for building terminal user interfaces. See https://ratatui.rs/. Last consulted: February 23, 2026.

with the egui immediate-mode GUI library[11] (Figure 1, Figure 6). It connects to an instance of `sova-server` or can start an embedded server from within the application itself. It is currently the most featureful client implementation available for Sova. `sova-frontend` provides two ways of visualizing the current server scene: as an hybrid step-sequencer / code editor view or as a grid of tiles, each embedding a code editor. In any case, lines and frames are animated visually to display real-time playback progress. Multiple side panels and/or widgets can be shown, hidden, or detached into separate windows depending on the performer's needs. A transport bar provides playback controls, tempo display, phase visualization, and execution-mode selection. The built-in code editor supports syntax highlighting and code hints (inlined documentation) for all demonstration languages, with multiple color themes, search, line numbers, and word wrapping. For audio feedback, an oscilloscope, a VU-meter and a spectrum analyzer run alongside the scene view (Figure 6). A sample browser lets users navigate and preview the audio files available to the Doux engine. A log panel with severity filtering displays server and client messages, and a searchable command palette offers quick access to all available actions. An early experimental port of Hydra [11] displayed over `egui` can be used to enrich the performance by projecting live-codable background visual shaders. Students involved in the pedagogical workshops described in Section 2 have already been introduced to it and are familiar with its usage, allowing them to jam both musically and visually using the scene view.

The collaborative dimension of the graphical interface draws on the experience gained by one of the authors as a contributor to Flok[12], a popular web-based collaborative editor for live coding developed by Damián Silvani [21]. In `sova-frontend`, connected peers can see each other and interact with each other through various actions. Each peer's cursor position and current editing activity is indicated within the scene grid, and a built-in text chat allows communication during performance. An in-app documentation panel provides getting-started guides and per-language references. Both the interface and the documentation are internationalized — currently in English and French — so that users from different countries can adopt the tool without a language barrier.

Despite this range of interfaces, several limitations remain due to the project's early stage of development. The code editor, while functional, lacks features that experienced programmers have come to expect from mature environments — notably inline error reporting and language-aware navigation. More broadly, all three interfaces remain code-centric: interaction with external controllers — MIDI surfaces, gamepads, or gestural devices — is limited to what the server exposes, with no dedicated mapping or binding per interface/client. A web-based client, which would eliminate installation entirely and open the door to mobile devices and browser-only workflows, has not yet been developed. These are potentials areas for future work. Feedback from experienced computer musicians would prove beneficial for the future stages of development.

```
EU 3 8 0.125:
  >> [note: ADD G.root MUL I 7
      vel: RRAND 60 127]
END
```

Listing 2. Bob syntax exemplified through the composition of an Euclidean rhythm applied to a MIDI sequence.

8. LANGUAGES

Four demonstration languages, in various stages of completeness, have been developed for Sova. Each language is adopting a distinct programming paradigm: imperative (Bob), declarative (Bali), pattern-based (Boinx), and concatenative (Cagire). Together they serve as a proof of concept that the architecture described in Section 3.1 is genuinely language-agnostic. Because all four share the same variable stores, clock, and device map, they interoperate freely within a single scene: a global variable set by a Bob script can be read by a Cagire script running on another line. Two of the languages (Bob and Bali) are compiled to bytecode for the shared virtual machine; the other two (Boinx and Cagire) provide their own interpreters and produce events directly through the common trait interface.

8.1. Compiled languages: Bob and Bali

Bob is an imperative, expression-oriented language that uses Polish (prefix) notation with fixed-arity operators, eliminating the need for parentheses. Inspired by the Monome Teletype hardware sequencer[13] design choices, it is designed for brevity and simplicity. Events are described as key-value maps and emitted with `>>`, while `WAIT` advances a virtual clock through the script. Bob provides four variable scopes (global, line, frame, and instance), a complete set of arithmetic and logical operators, and control structures including conditionals, counted loops, while loops, and a switch statement.

Beyond basic control flow, Bob offers built-in euclidean and binary rhythm generators, probabilistic execution (`PROB`), concurrent branching (`FORK`), lambdas, and higher-order list operations (`MAP`, `FILTER`, `REDUCE`). Source code is parsed by a LALRPOP grammar into an abstract syntax tree, then compiled to bytecode for the Sova virtual machine.

Bali (*Basically a Lisp*) is a declarative language whose (fake) S-expression syntax places timing at the core of the notation. A Bali program is a collection of

[11] Egui is a Rust library for building portable, immediate-mode graphical interfaces. See https://www.egui.rs/. Last consulted: February 23, 2026.

[12] See the Flok project website: https://flok.cc. Last consulted: February 22, 2026.

[13] See the Monome Teletype documentation: https://monome.org/docs/teletype/. Last consulted: February 23, 2026.

```
(loop 4
  (note c3 v:90 ch:1)
  (note {e3 g3} v:70))
```

Listing 3. Short demonstration of Lisp-like Bali syntax.

```
min = (. .+3 .+7)
maj = (. .+4 .+7)

[C3 A3 E3 G3] ! [maj min min maj]
~ [. [..] _ .]
# <s: 'saw' note: .>
```

Listing 4. Short demonstration of Boinx syntax.

timed musical effects — notes, control changes, OSC messages — distributed over fractional beat positions within a frame. Structural constructs such as `(loop N ...)`, `(eucloop ...)`, `(spread ...)`, and `(binloop ...)` subdivide the frame's duration into equal fractions, producing rhythmic patterns through nesting rather than explicit wait statements.

A context system, `(with dev:1 ch:2 v:80 ...)`, propagates default values for device, channel, velocity, and duration to all nested effects, reducing redundancy. Non-determinism is available through choice brackets `{...}` (random selection), alternation brackets `<...>` (sequential cycling), and sequence brackets `[...]` inspired by *Uzulang* languages[14]. The compiler, also built on a LALRPOP grammar, expands the AST into a time-sorted list of events and emits Sova virtual machine bytecode. Bali has historically been the first language to be developed for Sova but is currently relatively unused and unmaintained.

8.2. Interpreted languages: Boinx and Cagire

Boinx is a declarative, functional and pattern-based language built in order to visually build complex patterns scheduling events over a time-span. A program is constituted of assignments and outputs (which are all executed simultaneously when the script starts). The main idea of Boinx is to compose patterns into other patterns using slots (for instance placeholders '.' or durations) and collections (simultaneous '`(...)`' and sequentials '`[...]`'). To make complex compositions, Boinx uses 5 compositional operators (`|`, `°`, `!`, `~` and `#`).

Musical theory is handled in Boinx by the use of macros (`_scalemaj`, `_maj`, `_arpmaj`...) and it has generative aspects through functions (`choice`, `maybe`, `range`...). Subprograms can be started to emulate parallelism using `{ ... }`. Lastly, devices and channels can be specified using `@ device : channel` where device and channels can be Boinx objects as well (and thus, patterns). Boinx is internally compiled in its own syntax tree, but hosts its variables in Sova's environment, and can therefore interact with other languages.

[14]To see a list of existing uzulangs, consult: https://uzu.lurk.org/t/uzulangs/5660. Last consulted: February 27, 2026.

```
[ saw tri ] choose sound
[ c4 e4 g4 ] cycle note
0.5 gain 5000 100 0.5 slide lpf
2 4 rand fm 0.5 fmh .
```

Listing 5. Short demonstration of the Cagire syntax.

Cagire is a concatenative, stack-based language inspired by classic Forth [17] implementations. It features a very lightweight and approachable syntax: programs are sequences of words and numbers separated by spaces, evaluated in postfix order. Values are pushed onto a shared stack; words consume values from the top and push results back. A central design device is the *command register*, an accumulator that builds sound events incrementally: a sound name is set (`kick snd`), parameters are added (`0.5 gain`, `c4 note`, `0.3 verb`), and nothing is sent until the emit word `.` fires the accumulated command.

Cagire provides a rich vocabulary of stack manipulation words, can handle user-defined words (`: name ... ;`), first-class quotations (`( ... )`) for deferred code, and features an extensive music theory library. Probability words (`coin`, `sometimes`, `rarely`), euclidean sequencing (`bjork`), generators, and audio-rate modulation words (`lfo`, `slide`, `env`) round out the language. Internally, Cagire compiles source code to its own dedicated opcode set — distinct from the shared Sova virtual machine — but presents itself to the scheduler through the common interpreter trait, producing events directly. Cagire is a direct adaptation of the language used by the eponymous software currently developed by Raphaël Forment[15].

The diversity — and heterodox nature — of programming paradigms currently supported is deliberate. Each language was developed in a matter of weeks / months, validating the claim that Sova's architecture supports rapid creative language prototyping. The four paradigms — imperative, declarative, pattern-based, and concatenative — cover a broad region of the design space for event-based musical languages, while sharing the same runtime infrastructure. Musicians can choose the paradigm that best fits their way of thinking about music, or design entirely new ones. Developing these languages as test implementations during our initial work on Sova served as a de-facto validation of the general soundness of the software architecture.

9. CONCLUSION

In this paper, we have presented Sova, a free and open-source environment for collaborative live coding. This environment is built around a central scheduler, a dedicated virtual machine, and a lightweight audio engine. Sova was born out of a concrete pedagogical need: enabling students to engage with live coding through languages they could help design. It serves the dual purpose of facilitating the

[15]Project website: https://cagire.raphaelforment.fr. Last consulted: February 27, 2026.

Figure 8. High school students engaged in a live coding session during a workshop organized by Athénor CNCM (March 2026).

rapid development of new event-based musical languages and providing a solid foundation for teaching and performing. The diversity of paradigms currently supported validates that the architecture is genuinely language-agnostic and that new languages can be prototyped in a matter of days.

Sova is already being put to use in the field. Throughout winter and spring 2026, the software has been used in a — now ended — pedagogical project led by Athénor CNCM; teaching live coding practices to high school students in Nantes, Saint-Nazaire, and Guérande (Figure 8). Two 45 minutes long public collective improvisations led by 10+ networked high-school students have been performed as a joyous conclusion to this experience. Feedback from these sessions and workshops has been collected and will inform subsequent iterations of the software and its languages. In parallel, Sova has been introduced to the international live coding community through its presentation at Equinox, an independent conference hosted by TOPLAP Italia[16]. Sova has also been used in an Algorave context in Lyon by Johann Philippe and Juliette Monzat performing as a duo (May 2nd 2026). Sova will also be put to use as a central component for musical collaboration among members of the Lab 3 during Useful Fictions #6[17] (Festival de l'Eau, Saint-Nazaire, May 2026). We are actively seeking feedback from experienced researchers and computer musicians who may wish to use Sova for their creative or academic projects.

Several challenges remain. The software has not yet reached full stability across all supported platforms. Certain bugs — particularly on lower-specification hardware — still occasionally interrupt students' workflow during workshop sessions. On the language front, continued effort is needed to lower the barrier to language integration, to mature the existing demonstration languages, and to provide richer examples and internal / external documentation throughout the codebase and clients. While this work is underway, the onboarding experience for newcomers — students in particular — could be made smoother. Addressing these reliability and usability concerns is a priority for the near-term development effort. Sova's source code is released under the AGPL 3.0 license and is available on GitHub alongside a companion documentation and research website[18]. We hope to be able to build a small community dedicated to experimentation and research around Sova.

10. ACKNOWLEDGMENTS

The authors wish to thank Athénor CNCM for initiating and supporting the workshops and work periods that have been central to Sova's development and early validation. Raphaël Forment would like to warmly thank Tanguy Dubois and Loïg Jezequel for their invaluable contributions to the design and formal specification of Sova and more broadly for their sustained engagement with the project. We would also like to thank all the students who helped us test and develop this software. Tanguy Dubois would like to thank the ANR project BisoUS ANR-22-CE48-0012 for its funding.

11. REFERENCES

[1] Aaron, S. "Sonic Pi – performance in education, technology and art", *International Journal of Performance Arts and Digital Media*, 2016.

[2] Aaron, S., Blackwell, A. F., and Burnard, P. "The development of Sonic Pi and its use in educational partnerships: Co-creating pedagogies for learning computer programming", *Journal of Music, Technology & Education*, 2016.

[3] Blackwell, A. F., Cocker, E., Cox, G., McLean, A., and Magnusson, T. *Live coding: a user's manual*. MIT Press, 2022.

[4] Collins, N., and McLean, A. "Algorave: Live Performance of Algorithmic Electronic Dance Music", *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2014.

[5] Collins, N., McLean, A., Rohrhuber, J., and Ward, A. "Live Coding in Laptop Performance", *Organised Sound*, 2003.

[6] Cox, G., and McLean, A. *Speaking Code: Coding as Aesthetic and Political Expression*. The MIT Press, Cambridge, Mass, 2013.

[7] Forment, R. *"Some thoughts have a certain sound": Esthétiques et techniques du Live Coding*

[16] See the Equinox conference website: https://equinoxtoplap.it/. Last consulted: March 15, 2026.

[17] For more information, please refer to the festival program: https://www.athenor.com/les-rendez-vous/2025-2026/useful-fictions. Last consulted: May 14, 2026.

[18] https://sova.livecoding.fr, last consulted, May 14th 2026.

en musique. Université Jean Monnet (EPSCPE), Saint-Étienne, France, 2025.

[8] Free Software Foundation. "GNU Affero General Public License, Version 3", https://www.gnu.org/licenses/agpl-3.0.html, 2007.

[9] Goltz, F. "Ableton Link–A technology to synchronize music software", *Proceedings of the Linux Audio Conference*, 2018.

[10] Himanen, P. *The Hacker Ethic and the Spirit of the Information Age*. Random House, 2001.

[11] Jack, O. "Hydra: Live Coding Networked Visuals", Zenodo, https://doi.org/10.5281/zenodo.3946269, 2019.

[12] Magnusson, T. "Algorithms as Scores: Coding Live Music", *Leonardo Music Journal*, 2011.

[13] McCartney, J. "Rethinking the computer music language: Super collider", *Computer Music Journal*, 2002.

[14] McLean, A. "Making programming languages to dance to: live coding with tidal", *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, 2014.

[15] McLean, A., Griffiths, D., Collins, N., and Wiggins, G. "Visualisation of live code", *Electronic Visualisation and the Arts (EVA 2010)*, 2010.

[16] Mori, G. *Live coding? What does it mean? An ethnographical survey on an innovative improvisational approach*. 2020.

[17] Rather, E. D., Colburn, D. R., and Moore, C. H. "The evolution of Forth", *History of Programming Languages—II*, New York, NY, USA, 1996.

[18] Roberts, C., Allison, J., Holmes, D., Taylor, B., Wright, M., and Kuchera-Morin, J. "Educational design of live coding environments for the browser", *Journal of Music, Technology & Education*, 2016.

[19] Roberts, C., Wright, M., and Kuchera-Morin, J. "Beyond editing: extended interaction with textual code fragments", *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2015.

[20] Roos, F., and McLean, A. "Strudel: Live Coding Patterns on the Web", Zenodo, https://doi.org/10.5281/zenodo.7842142, 2023.

[21] Vasilakos, K. "Exploring live coding as performance practice", *Porte Akademik Müzik ve Dans Araştırmaları Dergisi*, 2021.

[22] Venners, B. "The java virtual machine", *Java and the Java virtual machine: definition, verification, validation*, 1998.

[23] Wakefield, G., and Roberts, C. "A virtual machine for live coding language design", *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2017.

[24] Wang, G., Cook, P. R., and others. "ChucK: A concurrent, on-the-fly, audio programming language", *ICMC*, 2003.

[25] Xambó, A., Freeman, J., Magerko, B., and Shah, P. "Challenges and new directions for collaborative live coding in the classroom", *International Conference of Live Interfaces (ICLI 2016). Brighton, UK*, 2016.

[26] Zyl, M. van, and Wang, G. "What's up ChucK? Development Update 2024", *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2024.