Satie: A Creativity Support Tool for Authoring Spatial Generative Audio

Mateo Larrea
Stanford University
Stanford, CA, USA

Yuhao Zhang
Independent Researcher
Palo Alto, CA, USA

Richard Boulanger
Berklee College of Music
Boston, MA, USA

Jerry Chen
Independent Researcher
Palo Alto, CA, USA

Pedro Sodre
New York University
New York, NY, USA

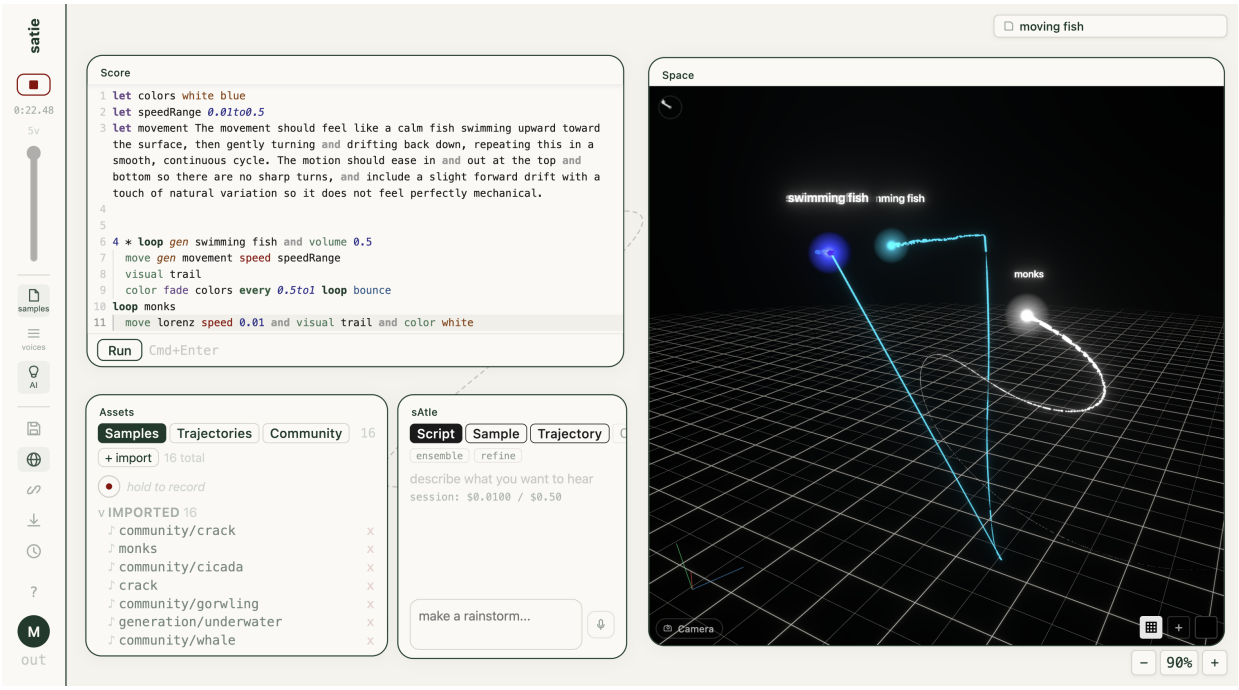

Figure 1: A Satie sketch in the browser. The Score panel (left) holds an eleven-line script: `let` bindings introduce a color palette, a speed range, and a natural-language movement description; `4 * loop gen swimming fish` spawns four independent voices whose audio is generated from the prompt; `move gen movement` compiles the natural-language description into a procedural trajectory; `color fade ... loop bounce` oscillates each voice between the bound colors. A second voice, `loop monks`, plays a community sample on a `lorenz` attractor trajectory. The Space panel (right) shows the four `swimming fish` voices tracing the AI-generated path alongside the white `monks` voice on its lorenz curve, all rendered in 3D space with HRTF panning.

Abstract

Composers and sound designers working in immersive spatial audio often face a gap between creative intent and implementation. Core behaviors such as stochastic timing, per-event parameter variation, and smooth 3D motion are conceptually straightforward but require substantial programming to realize in a game engine or DAW. Generative AI can help: large language models (LLMs) can produce code, and audio-generation models can synthesize source material from text prompts. However, these tools typically produce outputs the designer cannot fully read, understand, or selectively revise. We present Satie, a creativity support tool (CST) that lets designers harness generative AI while retaining ownership of what they produce. Satie's core is an audio-first domain-specific language (DSL) whose keywords mirror sound-designer vocabulary (e.g., `volume`, `pitch`, `fade`, `move fly`). Because the language is readable plain text with a line-per-property structure, it serves as a shared interface between the designer and AI tools: an LLM can generate Satie code, the runtime can generate audio from text prompts embedded in that code (via the `gen` keyword), the same keyword can also generate procedural spatial trajectories, and the designer can always read, understand, and revise individual properties without cascading changes to unrelated parts of the scene. Satie runs as an interpreted layer inside a web application built on the Web Audio API, enabling in-browser iteration with live editing and real-time visualization of sound-source trajectories. Sketches can be saved, forked, and shared through a public gallery, and `gen` statements

NIME '26, June 23–26, 2026, London, UK

are resolved against a community-uploaded sample library before falling back to AI synthesis. We demonstrate Satie with a set of examples that include immersive scenes, compositions, and scripts for live coding. We evaluate expressivity by comparing lines of code against matched game-engine implementations and report informal feedback from sound designers.

Keywords

creativity support tool, domain-specific language, spatial audio, generative audio, human-AI co-creation, sound design, ambisonics, web audio

1 Introduction

Sound designers working in games, virtual reality, and immersive media face a persistent challenge: translating creative vision into working implementations. Consider the task of creating a forest ambience with birds. Table 1 lists the requirements: randomized scheduling, spatial motion, and playback control. Most sound designers would readily understand what these requirements describe, yet implementing them in a game engine or a Digital Audio Workstation (DAW) requires substantial programming expertise: coroutines or event schedulers, audio source pooling, noise functions, trail renderers, and careful fade logic. In practice, sound designers often lack this programming background, forcing them to delegate implementation to programmers. This delegation introduces friction, iteration delays, and a loss of creative control.

Generative AI offers new flexibility: large language models (LLMs) can produce implementation code, and audio-generation models can synthesize source material from text prompts. But these tools typically produce outputs the designer cannot fully read, understand, or selectively revise. When an LLM generates hundreds of lines of game-engine code, the designer gains a working result but loses ownership of it.

We present Satie, a creativity support tool (CST) [17, 18] built around an audio-first domain-specific language (DSL). Rather than generating opaque application code that only a programmer can maintain, Satie uses keywords drawn from standard sound-designer vocabulary (`volume`, `pitch`, `fade`, `reverb`), producing scripts that both humans and LLMs can read and edit. Using Satie, a soundscape meeting the requirements in Table 1, which a matched LLM-generated game-engine implementation expresses in over 140 lines, can be expressed in 7 lines (Listing 1). Each requirement maps directly to a DSL keyword:

Category	Requirement	DSL keyword(s)
Randomization	Five different bird sounds	`5 * oneshot gen`
	Stochastic intervals (2–10 s)	`every 2to10`
	Varying pitch and volume	`pitch`, `volume` with ranges
Spatial motion	3D smooth-noise movement	`move fly`
	Trajectory visualization	`visual trail`
Playback control	Fade in over 5 s	`fade 5`
	Stop after 60 s	`end 60`

Table 1: Forest soundscape requirements and corresponding Satie DSL keywords.

Listing 1: Complete forest soundscape in Satie. Seven lines satisfy all requirements from Table 1: `gen` generates audio from a text prompt, `5 *` creates five unique variants, randomized `volume` and `pitch` ranges add per-event variation, `move fly` drives smooth-noise trajectories, and `end 60` stops playback after one minute.

```
5 * oneshot gen bird chirping every 2to10
    volume 0.1to0.5
    pitch 0.5to1.1
    fade 5
    move fly
    visual trail
    end 60
```

Each line in Listing 1 addresses a specific requirement from Table 1. The `gen` keyword triggers inline audio generation: at runtime, Satie first searches a community sample library for a matching sound and, if no match is found, calls a sound-generation API[1] to synthesize a clip from the prompt and caches it in the browser. When combined with a multiplier (`5 *`), the system instantiates five independent voices from the same prompt, so each bird plays a distinct sound. The `every` keyword schedules stochastic retriggering, `volume` and `pitch` ranges randomize each event, `move fly` assigns smooth-noise 3D trajectories, `visual trail` renders them, and `end 60` tears down playback after one minute. On subsequent runs, cached audio files are reused without regeneration.

Authors can write scripts directly in a Monaco-based editor, prompt them in natural language through an in-app chat or side-panel AI request, or fork another author's published sketch from a public gallery. All interaction modes produce Satie code as output. A key design goal is *targeted editing*. Consider an analogy from image generation: a designer prompts an AI to produce a horse wearing shoes, and the result is appealing except that the shoes should be red instead of white. Regenerating with a revised prompt risks losing the horse that was already satisfactory, because the model treats the entire image as a single opaque artifact. Satie's DSL avoids this problem by construction. Each sound behavior occupies an independent block of plain-text properties, so authors can revise a specific property (for example, changing the bird volume) without cascading changes to unrelated elements. This per-block structure preserves intent across iterations and is the same property that lets users selectively re-prompt one region of the script while leaving the rest intact.

Our contributions are:

- An audio-first domain-specific language whose plain-text, line-oriented structure is legible to both designers and large language models, enabling AI-assisted authoring while keeping every generated element readable and editable by hand.
- A web-native runtime built on the Web Audio API that supports in-browser iteration via live editing, real-time 3D visualization, head-related transfer function (HRTF) panning, five built-in spatial trajectory primitives, AI-generated procedural trajectories, and offline first-order ambisonic (FOA) export.
- A human-in-the-loop workflow in which AI-generated code is inspectable and selectively editable, with automatic audio retrieval from a community-uploaded sample library and synthesis fallback for sounds not present locally.

[1] Audio generation is provided by the ElevenLabs sound-generation API, https://elevenlabs.io/sound-effects.

- A demonstration through three immersive scenes; a lines-of-code comparison against LLM-generated game-engine code under matched conditions; and informal feedback from sound designers on steering AI outputs.
- An open-source web application requiring no installation, with a public gallery in which sketches can be saved, forked, liked, and embedded.

2 Related Work

2.1 Digital Audio Workstations

Digital audio workstations (DAWs) are optimized for timeline-based composition and mixing, not generative or procedural audio behaviors. Implementing randomized scheduling in a DAW typically requires workarounds such as clip follow-actions, LFO-driven parameters, or MIDI scripting, all of which impose grid quantization and timing jitter unsuitable for precise generative systems. True per-event randomization (independent pitch and volume for each triggered sound) is awkward without custom devices such as Max for Live patches, Logic Scripter, or JSFX plugins. Smooth 3D spatial motion has no native representation in most DAWs; achieving it requires piping Open Sound Control (OSC) messages to external visualization applications [1]. The result is that generative soundscapes in DAWs involve "fighting the tool," assembling complex multi-track setups that are difficult to scale or reuse.

2.2 Game Engine Native Audio

Game-engine audio systems (Unity's built-in audio, Unreal's MetaSounds) provide the necessary primitives (audio sources, clips, spatialization settings) but require extensive scripting to achieve generative behaviors. Our reference implementation of a comparable soundscape (with randomized timing, per-event parameter variation, smooth-noise 3D motion, trail visualization, and fade logic) required over 140 lines of C# code even when explicitly minimized (see Section 6). This implementation demands familiarity with object-oriented programming, coroutines, the engine's component system, and real-time audio considerations. For sound designers without this background, the cognitive overhead is overwhelming.

2.3 Audio Middleware

Professional audio middleware such as Wwise and FMOD offer sophisticated authoring environments with features like random containers, real-time parameter controls (RTPCs), and spatial audio pipelines [20]. However, these tools lack a unified authoring-to-runtime interface: designers must coordinate separate authoring applications, SDKs, plugins, SoundBank management, and engine-side wiring via PostEvent calls and callbacks. Version control becomes challenging because middleware exports binary SoundBank files that resist diffing and merging, unlike plain-text formats such as Satie's scripts. Debugging requires coordinating two applications. Crucially, neither Wwise nor FMOD provides built-in support for sound source motion; emitters must be moved programmatically each frame, reintroducing the programming burden. For small teams or individual practitioners, the overhead often exceeds the benefit.

2.4 Spatial Audio Authoring Tools

A more direct precedent for object-based spatial authoring comes from environments developed in the Max ecosystem. The IRCAM Spat library [10] has been the canonical authoring tool for object-based spatial audio for over two decades, exposing source positions, room properties, and ambisonic encoding through a visual patcher. The ICST Ambisonics toolkit for Max [11] provides an analogous decoder and encoder set, and Spatium [12] contributes a modular open-source toolset for ambisonic and amplitude-panned spatialization. These tools are powerful but expert-oriented: they assume facility with Max patching and OSC, and they decouple authoring from playback. Satie targets a different audience (sound designers who do not work in Max but still want object-level spatial behaviors with declarative syntax) and complements rather than replaces these environments.

A particularly relevant line of work treats spatial trajectories as autonomous behaviors rather than hand-keyframed paths. Reynolds' classical models of flocking and steering [13, 14] have been adapted by Carpentier and Gerzso [15] into the IRCAM Spat ecosystem, allowing composers to specify high-level autonomous trajectories (seek, wander, path-follow, flock) for sound sources. Dziwis [16] extends the approach to autonomous audiovisual agents for performance contexts. Satie shares this philosophy, exposing a small set of named trajectory primitives (`walk`, `fly`, `spiral`, `orbit`, `lorenz`, plus AI-generated custom paths) as first-class properties of the DSL.

2.5 Audio Programming Languages

Languages designed specifically for audio and music computation offer powerful abstractions but present their own barriers:

SuperCollider [2] provides a powerful and mature environment for real-time audio synthesis and algorithmic composition. However, it is a standalone audio language; integrating it with a game engine requires OSC communication bridges, introducing latency and architectural complexity.

Sonic Pi [5] emphasizes accessibility and live coding for education, but targets standalone performance rather than embedded use in interactive applications.

ChucK [3], embedded in Unity via Chunity [4] and recently ported to the browser as WebChucK [8], offers strongly-timed, concurrent audio programming directly in the host. However, ChucK's syntax represents a distinct paradigm unfamiliar to sound designers:

```
SinOsc s => dac;
440 => s.freq;
0.5 => s.gain;
2::second => now;
```

The "chuck operator" (`=>`) and time-advancing syntax (`2::second => now`) are powerful for audio programmers but require learning a new conceptual model. **TidalCycles** [6] and **Gibber** [7] are equally powerful but make rhythmic patterns or live audiovisual programming the central abstraction. Satie instead uses vocabulary sound designers already know: `volume`, `pitch`, `loop`, `fade`.

2.6 Human-AI Co-Creation

Recent work has explored large language models for code generation in creative contexts. However, when LLMs generate application code directly (e.g., game-engine C#), the output is often

opaque to designers with limited programming experience, leading to poor maintainability and scalability. Practitioners cannot easily inspect, understand, or selectively edit the generated code. This creates a "black box" problem where regenerating any portion risks losing previous work. Recent work in human-computer interaction (HCI) on novice-AI music co-creation has addressed the related problem of unsteerable generative output: Louie et al.'s Cococo [19] introduced AI-steering tools (voice lanes, semantic sliders, multiple alternatives) and showed they increase novices' trust, control, and sense of ownership.

Satie addresses the same problem through a different mechanism: it interposes a domain-specific language as an intermediate symbolic representation between the designer and the runtime. The LLM generates Satie code, not opaque application code, and Satie code is readable by sound designers. Because the DSL serves as a transparent intermediate layer, designers always have access to the current state of the generative process: they can inspect what was generated, revise specific regions without cascading changes, and preserve progress when regenerating portions of the composition.

3 Language Design

3.1 Design Principles

Satie's language design follows five principles:

(1) **Readable syntax**: Code should be understandable at a glance, even by those unfamiliar with programming.
(2) **Declarative statements**: Statements describe *what* should happen, not *how* to implement it.
(3) **Sound-designer vocabulary**: Keywords use terms from DAWs and mixing consoles (`volume`, `pitch`, `fade`, `reverb`) rather than programming abstractions.
(4) **Low floor, high ceiling**: Simple tasks should be approachable; complex behaviors should remain possible.
(5) **Web-first, zero installation**: The runtime, editor, and gallery are accessible from any modern browser without external dependencies.

3.2 Core Syntax

Satie scripts consist of *statements* that declare sound behaviors. Each statement begins with a playback type (`loop` or `oneshot`), followed by a clip name, optional timing modifiers, and indented properties:

```
loop ambient/forest
    volume 0.6
    fade 3
```

3.2.1 Playback Types. `loop` plays a clip continuously until stopped. `oneshot` plays a clip once. Adding `every` to a oneshot creates a repeating trigger:

```
oneshot footstep every 0.5to1.5
```

3.2.2 Multipliers. Prefixing a statement with `N *` spawns N independent instances, each with independently sampled random values:

```
5 * loop chirp
    volume 0.3to0.6
    pitch 0.8to1.3
```

3.2.3 Ranges and Randomization. The `to` keyword creates ranges for per-event randomization. Each value is sampled independently per instance:

```
oneshot bird every 2to10     - Random interval
    volume 0.5to0.9          - Random between
        0.5 and 0.9
    pitch 0.8to1.2           - Random pitch
        variation
```

3.2.4 Modulation. Parameters can be animated over time using two primitives: `fade` interpolates continuously between values, `jump` steps discretely. Both accept a period and an optional loop mode (`restart` or `bounce`):

```
volume fade 0 1 every 5                  -
    Fade from 0 to 1 over 5s
volume fade 0.2 0.8 every 5 loop bounce  -
    Oscillating between 0.2 and 0.8
pitch jump 0.5 1 1.5 2 every 1to3        -
    Random step pattern
```

3.2.5 Spatial Motion. The `move` property attaches a 3D trajectory to a voice. Built-in trajectories cover the most common cases; AI-generated trajectories (Section 3.2.9) are referenced by name:

```
move fly                    - 3D smooth-noise
    motion
move walk                   - Ground-plane
    motion
move spiral speed 0.5       - Analytical spiral
move orbit                  - Horizontal circle
move lorenz                 - Lorenz attractor
move a bird flying around - Trajectory from a
    gen block
```

3.2.6 Groups. The `group` construct applies shared properties to multiple statements. Group-level properties are written at the same indentation as the `group` keyword; child statements are indented below:

```
group background
volume 0.4
    loop wind
    loop rain
        volume 0.6
endgroup
```

Properties defined on the group are inherited by children; children can override with their own values. In this example, both `wind` and `rain` inherit `volume 0.4`, but `rain` overrides its volume to `0.6`.

3.2.7 DSP Effects. Built-in effects use familiar parameters, ranging from simple to detailed:

```
loop synth
    reverb wet 0.5
    delay wet 0.3 time 0.375 feedback 0.5
    filter mode lowpass cutoff 0.4 resonance
        0.6
```

3.2.8 Inline Audio Generation. The `gen` keyword, placed between the playback type and the prompt, marks a clip as generated rather than referenced from the local library:

```
loop gen fire with crackles
    volume 1
    pitch 0.5
```

```
oneshot gen thunder rumble every 5to15
    volume 0.8
```

When the runtime encounters a gen statement, it (1) checks a community sample library for a matching sound, (2) checks a per-prompt cache in the browser, and (3) calls the sound-generation API only if no match is found. Cached clips load instantly. Duplicate prompts are deduplicated so that only a single API call is made. Duration defaults are chosen automatically: 10 seconds for loops (suitable for seamless looping) and 5 seconds for one-shots. The gen keyword composes freely with all other properties: every, move, DSP effects, groups, and multipliers all work as expected.

3.2.9 Inline Trajectory Generation. The same gen keyword extends to spatial motion. A standalone gen block declares a custom trajectory by name with optional generation parameters; subsequent move references treat the result as if it were a built-in primitive:

```
gen a bird flying around
    smoothing 0.3
    seed 42
    ground 0.1

oneshot gen swallow
    move a bird flying around speed 0.6
    visual trail
```

3.3 Grammar Summary

Table 2 summarizes the core syntax constructs.

Table 2: Satie syntax summary

Construct	Example
Loop	`loop ambient`
One-shot	`oneshot click`
Repeat	`oneshot beep every 2to5`
Multiplier	`3 * loop rain`
Volume	`volume 0.8` or `volume 0.5to1.0`
Pitch	`pitch 0.9to1.1`
Start delay	`start 2.0`
End with fade	`end 60 fade 5`
Fade in	`fade 3`
Fade modulation	`volume fade 0 1 every 5`
Step modulation	`pitch jump 0.5 1 2 every 1to3`
Motion	`move fly speed 1`
Visual	`visual trail`
Group	`group name ... endgroup`
Generate audio	`loop gen fire with crackles`
Generate trajectory	`gen a bird flying around`

4 System Architecture

4.1 Overview

Satie is implemented as a web application that runs entirely in the browser. The user interface is organized around several views: a chat-driven landing page for natural-language soundscape generation, an editor that combines a Monaco-based code panel with a 3D viewport and side panels for samples, AI requests, voices, and export, a public gallery for browsing community sketches, and embeddable per-sketch players. All views share a single audio engine instance and reach the same backend for storage and AI proxying.

The audio engine itself consists of four main components: (1) a regex-based parser that transforms scripts into a list of statement objects, (2) a runtime that interprets these statements and manages audio voices, (3) a sample-accurate scheduler synchronized to the Web Audio clock, and (4) a multi-provider LLM pipeline for natural-language input. The engine is built entirely on the Web Audio API [9], with no native dependency or plugin to install.

A strict architectural rule separates the audio engine from the React-based UI: the engine has zero React dependencies, and the 3D viewport reads engine state from a mutable reference rather than from React state, so that 30 Hz position updates do not trigger re-renders.

4.2 Parser Implementation

The Satie parser processes scripts line by line using regular expressions to identify what each line represents. For each line, the parser determines whether it is a *statement* (`loop` or `oneshot`), a *property* (such as `volume` or `pitch`), a *group* definition, a *comment*, or a blank line.

Indentation is significant: it determines which properties belong to which statement. Before the main pass, the parser performs several preprocessing steps. It strips comments, expands and-joined property lines, substitutes `let`-bound variables, and expands multi-clip statements. It then extracts gen blocks for trajectories and audio separately, so that a gen clip prompt (e.g., `loop gen fire with crackles`) is rewritten into a canonical clip path with the original prompt preserved as metadata. This preprocessing is transparent to the rest of the pipeline: properties, groups, every syntax, and multipliers all work unchanged.

The parser then collects all indented lines below each statement as its properties. Each statement and its properties are packaged into a `Statement` object containing all relevant data: clip path, playback type, timing, volume, pitch, spatial movement, DSP effects, generation metadata, color, visuals, and other parameters.

The `RangeOrValue` struct represents both static values (`0.5`) and ranges (`0.5to1.0`), enabling uniform handling of randomization. The `InterpolationData` class captures animated parameters via the `fade` and `jump` primitives, with optional looping (`restart` or `bounce`).

Groups allow shared properties across multiple statements. When a group closes, its properties are merged into each child statement, with child values taking precedence.

The parser's output is a list of `Statement` objects. This list serves as the data structure that the runtime interprets to produce audio. Unknown properties are surfaced as inline warnings in the editor with a Levenshtein-distance "did you mean?" suggestion.

4.3 Runtime and Scheduler

The `SatieEngine` takes the list of `Statement` objects produced by the parser and interprets them, mapping each statement to a Web Audio voice. For each statement, the engine instantiates a `SatieTrack` object that manages the corresponding source nodes, applies the specified properties, and handles spatial positioning.

Because Satie is an interpreted language, there is no compilation step. The runtime reads the statement list directly and executes the instructions in real time. This architecture enables **live editing**: scripts can be modified while playback is active,

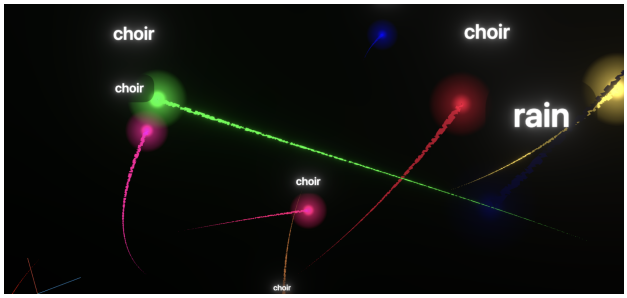

Figure 2: The Satie 3D viewport, displaying a multi-voice scene in real time. Each voice is rendered as a labeled marker; moving voices leave a colored trail that traces their trajectory. The listener sits at the origin, indicated by the axis gizmo (X/Y/Z). The camera can be flown with mouse and keyboard, and the listener orientation can be slaved to device orientation or webcam-based head tracking.

and the runtime re-parses and reconciles the resulting statement list against the live track set, preserving voices whose statements are unchanged. Critically, there is no experiential gap between the authored script and the running result: the designer hears exactly what the code describes, in the same environment where the final experience will be delivered. This tight feedback loop enables rapid iteration.

The `SatieScheduler` maintains a sample-accurate event timeline using a sorted array indexed by sample time, with O(log n) insertion via binary search. Timing precision is achieved through `SatieDSPClock`, which reads the Web Audio thread's clock (`AudioContext.currentTime`) rather than frame-based timing. This provides sample-accurate scheduling independent of frame rate fluctuations.

4.4 DSP Effects

Satie includes built-in DSP processors implemented as per-track Web Audio node chains. The chain order per voice is source → gain → filter → distortion → delay → reverb → EQ → panner, and the master bus is fed through a brick-wall `DynamicsCompressorNode` configured as a limiter to prevent clipping when many voices overlap.

- **Delay**: Configurable time, feedback, and stereo ping-pong
- **Reverb**: Convolutional reverb with procedural noise impulse, room size and damping controls
- **Filter**: Lowpass, highpass, bandpass, notch, and peak modes with cutoff and resonance
- **Distortion**: Softclip, hardclip, tanh, cubic, and asymmetric modes
- **EQ**: Three-band parametric equalizer

All effect parameters support static values, ranges (per-instance randomization), and `fade/jump` modulation (time-varying automation).

4.5 Spatial Audio

Spatial audio is a first-class concern in Satie, handled by the engine rather than by an external pipeline. This subsection describes the four pillars of the spatial layer. Figure 2 shows the live 3D viewport that visualizes them.

4.5.1 HRTF Panning and Listener Model. Each voice connects to a `PannerNode` configured with HRTF panning, an inverse distance model, and standard reference and rolloff parameters. Position parameters are smoothed with a 20 ms time constant to suppress zipper noise on parameter updates. The global `AudioListener` forward and up vectors are smoothed with a 10 ms time constant so that head turning, driven by the viewport camera or by an external orientation source, feels continuous.

4.5.2 Built-in Trajectories. Five trajectory primitives are provided. `walk` produces smooth-noise motion constrained to the ground plane, evaluated analytically as a sum of sinusoids with optional noise perturbation. `fly` extends this to three dimensions. `spiral` and `orbit` are analytical trajectories with configurable speed and radius. `lorenz` is a precomputed lookup table generated by integrating the Lorenz system (σ=10, ρ=28, β=8/3) with fourth-order Runge-Kutta. Each trajectory exposes the same parameters: a per-axis bounding box, a speed (or speed range), and an optional noise scalar. Position updates are rate-limited to 30 Hz.

4.5.3 AI Trajectory Generation. A natural-language description such as `bird that occasionally stops at a branch`, `lazy figure-eight over a lake`, or `drunken bee` (Figure 1 shows a complete example) can be compiled into a procedural trajectory in three stages.

First, the prompt is sent to an LLM with a system prompt that constrains output to a JSON object containing a `name` and `code` body of a JavaScript function. The function is asked to fill an interleaved `Float32Array` of length $SIZE \times 3$ with normalized $[0, 1]$ XYZ samples, given a deterministic seeded random source, and to support optional behavioral state machines (e.g., flying versus perching) and parameters for duration, smoothing, ground-locking, and variation.

Second, the returned code is executed inside a sandboxed `Function` constructor with only `SIZE`, `SEED`, and `Math` in scope, returning the filled buffer. The result is validated for length and finite values; on failure, a fast-tier model is asked to repair the code given the parser's error message.

Third, the buffer is post-processed: an optional moving-average smoothing window controlled by `smoothing`, and an optional ground-lock controlled by `ground` that flattens Y to a fixed value. The result is registered in the engine under the user's chosen name, persisted to IndexedDB so that reloads play instantly, and made available to any `move` property by name.

From the engine's perspective, an AI-generated trajectory is indistinguishable from a built-in one: the same evaluation routine reads the same lookup-table format. This gives the LLM a structured way to author motion without delegating real-time positioning to an opaque model: the trajectory itself is explicit code, the property in the script that selects it is plain text, and both can be edited.

4.5.4 Offline Ambisonic Export. Satie includes an offline rendering pipeline that delivers a complete sketch as a single audio file in three formats: stereo (equal-power panning), binaural (HRTF), and first-order ambisonics (FOA). The FOA path uses an `OfflineAudioContext` with a custom encoder that computes per-source AmbiX gains (ACN channel ordering with SN3D normalization). Distance attenuation matches the live panner's inverse model. The encoded audio is written as a multichannel WAV (16- or 24-bit) using `WAVE_FORMAT_EXTENSIBLE` so that ambisonic-aware

players preserve the channel order. Designers can therefore deliver a complete spatial soundscape as a portable file directly from a sketch, without leaving the Satie environment.

5 Human-AI Co-Creation

5.1 Three Authoring Surfaces

Satie supports three ways to author soundscapes, all producing Satie code as output:

(1) **Direct code**: Authors write scripts in a Monaco-based editor with syntax highlighting, context-aware completion, hover documentation for every keyword and property, and live validation. Edits are reflected in the running soundscape immediately.
(2) **Natural language via chat**: The default landing page is a chat interface in which authors describe the desired soundscape in plain English (e.g., "a quiet forest with flying birds around me"). The LLM produces or revises the script across multiple turns, with the recent exchanges passed back so that follow-ups ("add more reverb", "slow the birds down") refine the existing sketch rather than restart it.
(3) **In-editor AI panel**: A side panel allows targeted AI requests against three surfaces: scripts, audio samples, and trajectories. The same provider-agnostic backend services all three.

The critical insight is that all three surfaces produce *the same artifact*: human-readable Satie code. Authors can seamlessly switch between them, inspect AI-generated code, and make manual edits.

5.2 LLM Pipeline

The LLM pipeline uses a provider-agnostic abstraction implemented for Anthropic Claude, OpenAI GPT, and Google Gemini, with both a primary tier (Sonnet, GPT-4o, Gemini Flash) and a fast tier (Haiku, GPT-4o-mini, Flash Lite). A short heuristic classifies each prompt as *simple* (a localized edit to an existing script) or *complex* (a fresh composition or substantial restructure). Simple prompts route to the fast tier; complex prompts use the primary tier. Where the provider supports it, the long static portion of the system prompt is cached across calls to reduce cost.

The system prompt bundles the DSL grammar, compositional patterns (variation through multiplication, layering, evolving parameters, rhythmic patterns, spatial depth), and a list of common mistakes (legacy keywords, equals signs, quoted clip names). It also enforces a spatial-visual mapping that helps keep generated scripts legible: moving voices receive `visual trail`, static voices receive `visual sphere`. The same compositional patterns guide the LLM to assign motion based on real-world plausibility: voices that would naturally travel (birds, footsteps, vehicles) receive `move` properties, while voices that would not (room tone, fire, distant rumble) are left static. The author can always override either choice in the script. A dynamic suffix is appended at call time, listing the audio samples currently loaded by the user, the current script if any, and the user's prompt.

The model's response is stripped of markdown fencing and prose preambles, then validated by re-running the parser. On parser failure, a fast-tier model is invoked with a *repair* prompt that lists the strict syntax rules and the parser's error message; up to two repair attempts are made before the failure is surfaced. For users without their own API keys, a serverless proxy routes calls through server-side credentials with a per-session budget guard.

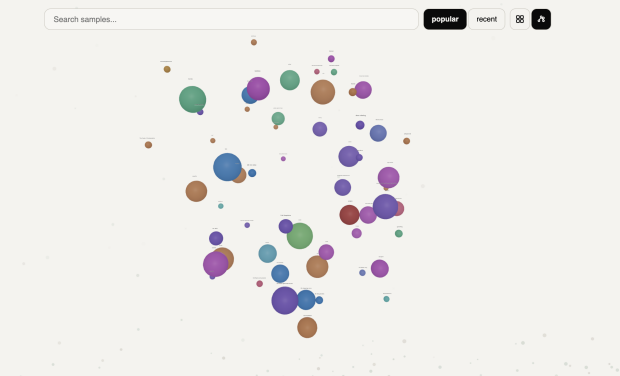

Figure 3: The community sample library, visualized as a similarity graph. Each bubble is a user-uploaded sample; size encodes download count and color indicates the dominant tag cluster. Samples close to one another in the graph share embedding-space neighborhoods. `gen` statements at runtime query this library before falling back to text-to-audio synthesis.

5.3 Material Retrieval

A persistent question for any spatial-audio system that uses generative AI is how the language model bridges between an abstract intent ("a forest with crackling fire") and concrete sonic material on disk. Satie routes every `gen` request through an explicit ordered cascade:

(1) **Loaded samples**: If the user has uploaded local samples to the sketch, the LLM is instructed to reference them by name. The dynamic system prompt includes the file list, so the model never invents a nonexistent path.
(2) **Community library**: For each `gen` clip name at runtime, the engine issues a parallel three-channel query against the community sample library: exact-tag overlap, full-text search over names and descriptions, and cosine similarity over an embedding column (`pgvector`). Results are deduplicated and re-ranked by combined match score, recency, and download count.
(3) **Synthesis fallback**: If no community match clears the similarity threshold, the engine calls the sound-generation API with the prompt, caches the resulting audio in the browser keyed by prompt and duration, decodes it, and plays it.

This ordering matters because community retrieval is essentially free relative to API-based synthesis, and because the community library accumulates over time. From the designer's perspective, the difference is invisible: a single `gen` statement may resolve to a community sample in one session and a freshly synthesized clip in another, but the script reads identically in both cases. Figure 3 shows the library users actually see: a graph of uploaded samples positioned by embedding similarity, with bubble size reflecting download count.

5.4 Targeted Editing via the DSL

A persistent challenge with generative AI tools is the loss of creative agency. When an LLM generates complex application code

directly (e.g., game-engine C#), designers with limited programming experience often cannot understand or modify the output. Regenerating any portion risks cascading changes that break unrelated behaviors, a frustrating experience that undermines creative flow.

Satie addresses this by using the DSL as a transparent intermediate representation between the designer's intent and the runtime. Because the LLM generates Satie code rather than application code, the current state of the composition is always visible and editable. Authors can:

- **Inspect** what the AI produced and understand its structure at a glance.
- **Edit** specific DSL regions (e.g., change `volume 0.5` to `volume 0.3`) without cascading changes to unrelated scene elements.
- **Preserve progress** when regenerating portions. For example, if the birds sound right but the wind layer needs work, the author can regenerate only the wind group while keeping the birds intact.
- **Fork and remix**: any public sketch can be forked into the user's account in one click; the new sketch is editable, the original remains intact.

This targeted-editing property helps maintain human authorship and creative control even when leveraging AI assistance.

5.5 Public Gallery and Sketch Sharing

Sketches are first-class persistent objects. On save, the script and a thumbnail are written to a backend store, and any audio buffers held by the engine, including AI-generated audio decoded from the synthesis API, are uploaded as 16-bit WAV samples. A public-private toggle controls whether a sketch appears in the public gallery and is reachable at a stable URL; a fork action duplicates the sketch under the current user. The result is that any visitor can play, like, fork, or embed any public sketch without API keys, because the sketch carries its rendered samples with it. This addresses the longstanding question of how to share procedural audio: the recipient does not need to reproduce the author's environment to hear the work. Figure 4 shows the public gallery.

6 Evaluation

We evaluate Satie along three axes: (1) a lines-of-code comparison against LLM-generated game-engine code under identical requirements, (2) informal feedback from sound designers on editing LLM-generated game-engine code versus LLM-generated Satie code, and (3) the end-to-end latency of the LLM pipeline from prompt to verified script.

6.1 Demonstration Scenes

To illustrate the range of behaviors expressible in Satie, we authored three immersive scenes, each exercising a distinct combination of DSL features (Table 3). Each scene is published as a public sketch and is playable directly in the browser without any API keys, because the saved sketch carries its pre-rendered sample WAVs.

The three scenes span common spatial audio authoring patterns: stochastic scheduling with per-event parameter variation (all three), smooth spatial motion across walk, fly, and per-axis modes (across the three scenes), grouped behaviors with interpolated fade-ins (all three), DSP effect chains including reverb, delay, and animated filter sweeps (Forest, City), and animated color channels (Forest, Underwater).

Table 3: Demonstration scenes authored in Satie. Each scene exercises a distinct combination of DSL features.

Scene	Key Idioms
Underwater	Groups, `gen` multipliers, walk/fly/axis movement, reverb, visual sphere and trail, color
Enchanted Forest	Filter sweep (`fade`), delay, animated color channels, walk/fly movement, reverb, visual sphere and trail
City Street	Volume `fade`, pitch `jump`, lowpass + reverb combo, walk/axis movement, visual sphere and trail

6.2 Expressivity: Lines of Code vs. LLM-Generated Game-Engine Code

To evaluate the DSL's expressivity, we compare the Satie scripts for each demonstration artifact against functionally matched game-engine implementations in C#. Critically, both the Satie scripts and the C# implementations were produced by an LLM (Claude Sonnet) given identical natural-language requirements. The C# prompt included an explicit objective to minimize code length, ensuring the comparison reflects the DSL's structural advantage rather than incidental verbosity in the C# baseline. The complete scripts, prompts, and C# excerpts are provided in Appendix A.

Table 4: Lines of code: both columns were generated by the same LLM from identical natural-language requirements. The C# prompt included an explicit objective to minimize code length.

Scene	Satie (LLM)	C# (LLM)	Ratio
Underwater	30	141	4.7×
Enchanted Forest	36	173	4.8×
City Street	32	165	5.2×

Even when the LLM is explicitly instructed to minimize C# length, Satie scripts are approximately 5× shorter. This gap reflects a structural property of the DSL: behaviors that require boilerplate in C# (coroutine scheduling, audio-source pooling, smooth-noise motion, fade logic) are first-class primitives in Satie.

6.3 Informal Feedback: Steering AI Outputs

To assess whether the DSL's targeted-editing property translates to a practical advantage when steering AI-generated code, we conducted informal feedback sessions with five sound designers (three professionals working in game audio and two graduate students in music technology). Each participant was given the same spatial audio scene implemented in two forms: (1) LLM-generated game-engine C# and (2) LLM-generated Satie code. Both were produced by the same LLM from identical instructions.

Participants were asked to perform a targeted edit (e.g., "change the flying range of the birds," "add reverb to the wind layer") in each representation and were then asked whether they could identify the relevant region, whether they felt confident the edit would not break unrelated behaviors, and which representation they would prefer for iterative refinement.

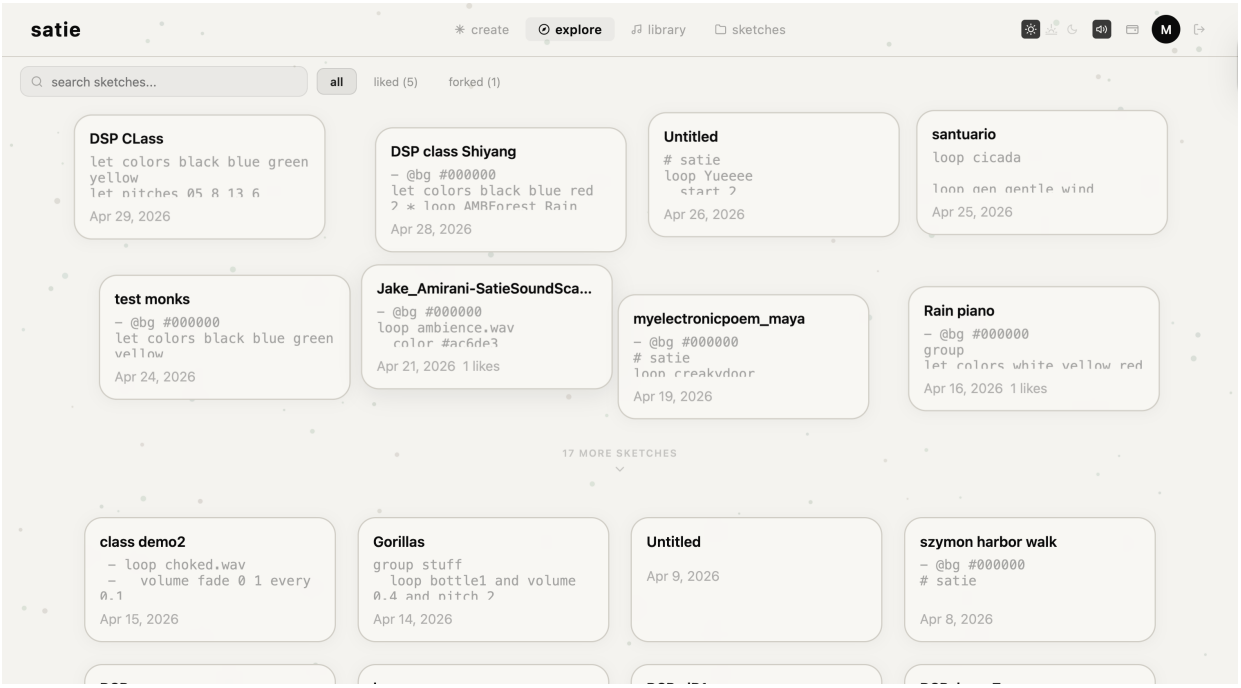

Figure 4: The public gallery (`/explore`). Each card is a saved sketch with title, the script's opening lines, save date, and like count. Sketches play directly in the browser without API keys because their pre-rendered audio is uploaded with the script on save; visitors can also fork any sketch into their own account in one click.

All five participants completed the Satie edit within seconds and reported high confidence. Representative comments included:

- "It's very readable. Feels like what I'd write on a sticky note if I were describing a sound to someone."
- "I have some C# experience, but I wasn't sure how to apply the reverb to the audio source. I couldn't tell if I had to reference it somewhere else in the code."
- "Each line is one property, so I can change the birds without worrying about the wind. In the C# code everything is tangled together in coroutines."
- "It's closer to how I think about sound design in a DAW. Layers with parameters, except it also handles the scheduling and spatial movement."

Recurring themes were: (1) participants found Satie scripts easier to navigate because each property occupies a single line using vocabulary familiar from DAW workflows; (2) in the C# versions, participants reported difficulty locating the relevant code and anxiety about unintended side effects from edits to coroutine control flow and shared state; and (3) all five preferred Satie for iterative refinement, citing the ability to make targeted edits without understanding the full program.

6.4 Pipeline Latency

We measured the end-to-end latency of the LLM pipeline from natural-language prompt to verified Satie script. Table 5 reports representative timings averaged over ten prompts of varying complexity (single-track edits to full multi-group scenes), measured against the Anthropic API (Claude Sonnet for complex prompts, Claude Haiku for simple prompts and the repair loop), with prompt caching enabled.

The dominant cost is the code-generation call. Verification is local (re-running the parser) and effectively free. The repair loop is invoked for roughly one prompt in eight; when invoked, it adds 1–2 s. The editor UI classifies responses below 2 s as "Excellent," below 5 s as "Good," and above 5 s as "Slow"; in practice, simple prompts fall in the Excellent range and complex prompts in the Good range. Audio generation via the synthesis API (triggered by the `gen` keyword when no community match is found) adds additional latency of approximately 3–8 s per clip, but this occurs asynchronously after the script is delivered and does not block non-generated tracks.

Table 5: LLM pipeline latency breakdown (mean ± std. dev. over 10 prompts).

Phase	Latency (ms)
Code generation (simple, fast tier)	1,600 ± 500
Code generation (complex, primary tier)	4,200 ± 900
Compilation verification	40 ± 20
Self-correction (when needed)	+1,200 ± 400
Total simple (no repair)	**1,700 ± 500**
Total complex (no repair)	**4,300 ± 900**

7 Discussion

7.1 Limitations

Browser audio constraints. The runtime relies on standard Web Audio nodes and does not currently use AudioWorklet, which would allow custom DSP at sample-block granularity. This is acceptable for the present feature set but constrains future synthesis work.

Spatial fidelity. The browser's HRTF model is a single anonymous head model; we do not yet support per-user HRTF measurement or SOFA file loading. For high-end spatial audio production,

designers may export FOA from Satie and decode in a more flexible host.

API costs and connectivity. The AI features depend on third-party APIs: Anthropic, OpenAI, or Google for code generation, and ElevenLabs for audio generation. Each requires an API key and incurs per-use costs. The hosted version proxies these calls and absorbs the cost up to a per-user budget; users with their own keys can override this. None of these features can be used offline.

Limited DSL control over audio-generation parameters. The `gen` keyword for audio currently accepts a prompt, an optional duration, an influence parameter, and a `loopable` flag, but does not expose model selection or seed. The trajectory variant of `gen` is more parameterized; we expect the audio variant to converge toward the same surface.

7.2 Future Work

Open and offline AI backends. To reduce cost and connectivity dependencies, we intend to support open-source alternatives for both code and audio generation, and make all AI features optional, so the DSL and runtime remain fully functional without API keys.

Pluggable spatialization. A user-loaded SOFA file would give individualized binaural rendering; an in-browser FOA-to-binaural decoder would let designers audition ambisonic exports without leaving Satie.

Trajectory-aware sample retrieval. A natural extension of the material-retrieval cascade is to condition retrieval on the trajectory: a sample tagged for a flying-by character should be preferred over a static one when `move fly` is active.

Generative score structure. Beyond per-voice generation, we plan to explore LLM-driven authoring of higher-level score structures (sections, transitions, multi-minute compositions) that compose existing sketches.

We also plan controlled user studies comparing Satie to existing workflows across metrics of task completion time, perceived creative control, and output quality.

8 Conclusion

Satie demonstrates that combining an audio-first DSL with a human-in-the-loop workflow can preserve practitioners' agency and authorship while substantially lowering the implementation barrier for spatial generative audio. The DSL lets sound designers express stochastic timing, per-event variation, smooth 3D motion, AI-generated trajectories, and DSP processing in terms they already know; the workflow ensures that AI-proposed code remains inspectable and selectively editable. The browser-based runtime removes the installation burden and includes an offline ambisonic export pipeline, so designers can render complete spatial soundscapes as first-order B-format or binaural files directly from a script. The community library and public gallery turn one author's work into another's starting material: a fork is one click, and shared sketches play without API keys because they carry their pre-rendered audio. Our evaluation shows that Satie scripts are markedly more concise than LLM-generated game-engine C# even when the LLM is explicitly instructed to minimize code length, and informal feedback sessions suggest that sound designers can more confidently steer AI outputs in the DSL than in general-purpose code.

The system is open source and runs in any modern browser at https://satie.app.

9 Ethical Standards

This paper presents a system design and implementation. The informal feedback sessions described in Section 6 involved voluntary participants who provided verbal comments during open-ended demonstrations; no personal data was collected or retained. The system is open source under the MIT license. Generative AI providers integrated by Satie include Anthropic Claude, OpenAI GPT, and Google Gemini for code generation, and ElevenLabs for optional audio generation. Community-uploaded samples are subject to a clear license declaration at upload time and are stored with attribution to the uploader. The authors declare no conflicts of interest. Development was conducted at the authors' respective institutions without external funding specific to this project.

Acknowledgements

The name "Satie" honors Erik Satie, the French composer who coined the term *musique d'ameublement* (furniture music) in 1917 [21], music intended to blend into the environment rather than demand active attention. This concept anticipated contemporary practices in ambient music, soundscape composition, and environmental audio design for games and immersive media.

References

[1] Karen Collins. An introduction to procedural music in video games. *Contemporary Music Review*, 28(1):5–15, 2009.
[2] James McCartney. Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002.
[3] Ge Wang and Perry R. Cook. ChucK: A concurrent, on-the-fly, audio programming language. In *Proceedings of the International Computer Music Conference*, pages 219–226, 2003.
[4] Jack Atherton and Ge Wang. Chunity: Integrated audiovisual programming in Unity. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, pages 102–107, 2018.
[5] Samuel Aaron and Alan F. Blackwell. From sonic Pi to overtone: creative musical experiences with domain-specific and functional languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design (FARM '13)*, pages 35–46. ACM, 2013.
[6] Alex McLean. Making programming languages to dance to: Live coding with Tidal. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design (FARM '14)*, pages 63–70. ACM, 2014.
[7] Charles Roberts, Matthew Wright, JoAnn Kuchera-Morin, and Tobias Höllerer. Gibber: Abstractions for creative multimedia programming. In *Proceedings of the 22nd ACM International Conference on Multimedia (MM '14)*, pages 67–76. ACM, 2014.
[8] Michael Mulshine, Ge Wang, Chris Chafe, Jack Atherton, Terry Feng, and Celeste Betancur. WebChucK: Computer music programming on the web. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2023.
[9] Lonce Wyse and Srikumar Subramanian. The viability of the web browser as a computer music platform. *Computer Music Journal*, 37(4):10–23, 2013.
[10] Thibaut Carpentier. A new implementation of Spat in Max. In *Proceedings of the 15th Sound and Music Computing Conference (SMC)*, 2018.
[11] Jan C. Schacher. Seven years of ICST Ambisonics tools for MaxMSP: a brief report. In *Proceedings of the 2nd International Symposium on Ambisonics and Spherical Acoustics*, 2010.
[12] Rui Penha and João Pedro Oliveira. Spatium, tools for sound spatialization. In *Proceedings of the Sound and Music Computing Conference (SMC)*, 2013.
[13] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87)*, pages 25–34. ACM, 1987.
[14] Craig W. Reynolds. Steering behaviors for autonomous characters. In *Proceedings of the Game Developers Conference (GDC)*, 1999.
[15] Thibaut Carpentier and Andrew Gerzso. Steering behaviors for spatial sound authoring. In *Proceedings of the 45th International Computer Music Conference (ICMC)*, 2019.
[16] Damian Dziwis. Revisiting Reynolds: autonomous agents for spatial audiovisual composition and performances. In *Proceedings of the 4th Conference on AI Music Creativity (AIMC)*, 2023.
[17] Ben Shneiderman. Creativity support tools: Accelerating discovery and innovation. *Communications of the ACM*, 50(12):20–32, December 2007.
[18] Jonas Frich, Lindsay MacDonald Vermeulen, Christian Remy, Michael Mose Biskjaer, and Peter Dalsgaard. Mapping the landscape of creativity support

tools in HCI. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*, Paper 389, 18 pages. ACM, 2019.
[19] Ryan Louie, Andy Coenen, Cheng Zhi Anna Huang, Michael Terry, and Carrie J. Cai. Novice-AI music co-creation via AI-steering tools for deep generative models. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*, pages 1–13. ACM, 2020.
[20] Game Audio Learning. Audio middleware and how to use it. https://www.gameaudiolearning.com/, 2024.
[21] Erik Satie. Musique d'ameublement. Composition, 1917.

A Evaluation Materials

All Satie scripts, C# implementations, and LLM prompts used in the evaluation are available in the project repository. Each demonstration scene is also published as a public sketch and is playable in any modern browser without an account. Below we include the LLM prompt template and one representative scene for reference.

A.1 LLM Prompt for Game-Engine C# Generation

Each C# implementation was generated by Claude Sonnet with the following prompt (scene-specific requirements substituted where indicated):

> Write a single C# MonoBehaviour that implements the following spatial audio scene. Use only the engine's built-in audio system. Minimize code length; use helper methods to avoid repetition. The script should: [scene-specific requirements]. Use Perlin noise for spatial motion, trail renderers for visual trails, and primitive spheres for visual sphere markers. All audio sources should be spatialized.

The explicit instruction to minimize code length ensures that the comparison reflects Satie's structural advantage rather than incidental C# verbosity. The same LLM and identical requirements were used to generate both the Satie and C# versions.

A.2 Representative Scene: Underwater

The Underwater scene (30 lines of Satie, 141 lines of C#) exercises groups, gen multipliers, three movement modes, reverb, and visual debugging.

```
group underwater
    volume fade 0 1 every 5
    loop gen underwater calm ambience
        volume 0.5
        reverb wet 0.2

    5 * oneshot gen bubbling note every 1to10
        volume 0.1to0.3
        pitch 0.7to2
        move walk speed 2to3
        visual sphere
        color blue

    5 * oneshot gen whale call every 10to15
        volume 0.6to1
        move fly speed 0.5to2
        visual trail
        color blue

    3 * oneshot gen dolphin click every 10to20
        volume 0.2to0.3
        pitch 0.8to1.5
        move spiral speed 1to3

    oneshot gen deep ocean rumble every 15to25
        volume 0.2

    2 * oneshot gen water splash every 5to12
        volume 0.01to0.1
        pitch 0.8to1.2
endgroup
```

The corresponding C# implementation requires coroutine scheduling, audio-source pooling, smooth-noise motion helpers, trail renderers, and group fade logic. The remaining two scenes (Enchanted Forest, City Street), all three C# scripts, and the public-sketch URLs are linked from the project repository.