Flocking: A Framework for Declarative Music-Making on the Web

Colin Clark
OCAD University
cclark@ocadu.ca

Adam Tindale
OCAD University
atindale@faculty.ocadu.ca

ABSTRACT

Flocking[1] is a framework for audio synthesis and music composition written in JavaScript. It takes a unique approach to solving several of the common architectural problems faced by computer music environments, emphasizing a declarative style that is closely aligned with the principles of the web.

Flocking's goal is to enable the growth of an ecosystem of tools that can easily parse and understand the logic and semantics of digital instruments by representing the basic building blocks of synthesis declaratively. This is particularly useful for supporting generative composition (where programs generate new instruments and scores algorithmically), graphical tools (for programmers and non-programmers alike to collaborate), and new modes of social programming that allow musicians to easily adapt, extend, and rework existing instruments without having to "fork" their code.

Flocking provides a robust, optimized, and well-tested architecture that explicitly supports extensibility and long-term growth. Flocking runs in nearly any modern JavaScript environment, including desktop and mobile browsers (Chrome, Firefox, and Safari), as well as on embedded devices with Node.js.

1. INTRODUCTION

A prominent stream in computer music research over the past few decades has focused on the creation of specialized languages for expressing musical and time-based constructs programmatically [1, 2, 3, 4]. This emphasis on new forms of syntax and language-level expression has produced noteworthy computer music environments and useful results for many use cases such as live coding. Nonetheless, there is also a risk associated with the proliferation of isolated, specialist programming languages for music and art: an increased gap between creative coders and the resources available to mainstream software developers. For example, in many self-contained computer music environments, it continues to be difficult to create polished user interfaces or to connect with web-based services and sources of data—tasks that are routinely addressed in mainstream programming environments such as JavaScript. As artists and musicians increasingly use networked devices, sensors, and collaboration in their work, these limitations take an increasing toll on the complexity and scalability of creative coding.

Flocking is an open source JavaScript framework that aims to address some of these concerns by connecting musicians and artists with the cross-platform, distributed delivery model of the web, and with the larger pool of libraries, user interface components, and tutorials that are available to the web development community. Further, it emphasizes an approach to interoperability in which declarative instruments and compositions can be broadly shared, manipulated, and extended across traditional technical subcultural boundaries.

1.1 Interoperability in Context

A primary motivating concern for Flocking is that the tendency towards music-specific programming languages shifts focus away from interoperability amongst tools and systems. The term "interoperability" is used here to describe a specific concept: the ability to share a single instance of a computer music artifact (i.e. an instrument or score) *bidirectionally* amongst human coders, generative or transformational algorithms, and authoring or graphical tools. Bidirectionality implies that a software artifact needs to preserve sufficient semantics and landmarks that it can be inspected, overridden, and extended by humans and programs not only at creation time but throughout the process of being used and maintained.

Today, a prospective computer musician often must choose from the outset whether or not she wants to use a code-based environment (such as SuperCollider or ChucK) or a graphical one (Max/MSP, Pd, or AudioMulch, for example). Since imperative programming code can't easily be parsed, generated, and understood by tools outside the chosen environment, the code and graphical paradigms rarely interoperate. This compounds the difficulty of collaborating on a musical project across modalities.

Interoperability amongst computer music systems has been addressed in a number of ways and to varying degrees. Open Sound Control [5], for example, helps support cross-system, message-based interoperability at runtime. Some graphical environments such as Max and Pd support the embedding of programmatic "externals" within an otherwise graphical instrument. FAUST offers unidirectional code generators for a variety of target languages, enabling programs to be written in the FAUST language but deployed within other environments. The Music-N family's simple textual format has fostered a variety of third-

[1] http://flockingjs.org/

party compositional tools that can process and generate score and orchestra files.

Some computer music environments also provide APIs for manipulating the language's parsing and compilation artifacts. One of CSound 6's new features includes an abstract syntax tree API, enabling a user to write C code that manipulates an orchestra prior to compilation [6]. Max/MSP's Patcher API supports the programmatic traversal and generation of a Max patch using Java or JavaScript code [2]. Lisp-based languages such as Extempore go further towards potential interoperability, providing macro systems that allow for more robust generative algorithms to be created within the facilities of the language itself.

Within this context, Flocking aims to provide a framework that supports extended interoperability via a declarative programming model where the intentions of code are expressed as JavaScript Object Notation (JSON) data structures. JSON is a subset of the JavaScript language that is used widely across the web for exchanging data [3]. Flocking's approach combines *metaprogramming* with an emphasis on publically-visible state and structural landmarks that help to support the alignment, sharing, and extension of musical artifacts across communities of programmers and tools.

2. HOW FLOCKING WORKS

2.1 The Framework

The core of the Flocking framework consists of several interconnected components that provide the essential behaviour of interpreting and instantiating unit generators, producing streams of samples, and scheduling changes. Flocking's primary components include:

1. the *Flocking interpreter*, which parses and instantiates synths, unit generators, and buffers
2. the *Environment*, which represents the overall audio system and its configuration settings
3. *Audio Strategies*, which are pluggable audio output adaptors (binding to backends such as the Web Audio API or ALSA on Node.js)
4. *Unit Generators* (ugens), which are the sample-generating primitives used to produce sound
5. *Synths*, which represent instruments and collections of signal-generating logic
6. the *Scheduler*, which manages time-based change events on a synth

Figure 1 shows the runtime relationships between these components, showing an example of how multiple synths and unit generators are composed into a single Web Audio ScriptProcessorNode.

[2] http://cycling74.com/docs/max5/vignettes/js/jspatcherobject.html
[3] http://json.org

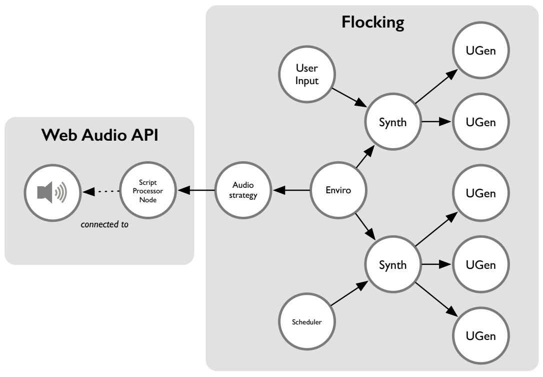


Figure 1. A diagram showing Flocking's primary components and how they relate to each other and to the Web Audio API.

2.2 Declarative Programming

Above, we described Flocking as a declarative framework. This characteristic is essential to understanding its design. Declarative programming can be understood in the context of Flocking as having two essential aspects:

1. it emphasizes a high-level, semantic view of a program's logic and structure
2. it represents programs as data structures that can be understood by other programs

J.W. Lloyd informally describes declarative programming as "stating *what* is to be computed but not necessarily *how* it is to be computed" [7]. The emphasis here is on the logical or semantic aspects of computation, rather than on low-level sequencing and control flow. Traditional imperative programming styles are typically intended for an "audience of one"—the compiler. Though code is often shared amongst multiple developers, it can't typically be understood or manipulated by programs other than the compiler.

In contrast, declarative programming involves the ability to write programs that are represented in a format that can be processed by other programs as ordinary data. The Lisp family of languages are a well-known example of this approach. Paul Graham describes the declarative nature of Lisp, saying it "has no syntax. You write programs in the parse trees... [that] are fully accessible to your programs. You can write programs that manipulate them... programs that write programs." Though Flocking is written in ordinary JavaScript, it shares with Lisp the approach of expressing programs within data structures that are fully available for manipulation by other programs.

2.3 JSON

The key to Flocking's declarative approach is JSON, the JavaScript Object Notation format. JSON is a lightweight data interchange format based on a subset of JavaScript that can be parsed and manipulated in nearly any programming language. JSON provides several primary data types and structures that are available across programming languages. The following table describes these data structures and their syntax:

Type	Syntax	Description
Object	`{}`	Dictionary of key/value pairs
Array	`[]`	An ordered list
String	`"cat"`	A character sequence
Number	`440.4`	A floating point number

Since JSON's syntax and semantics are identical to JavaScript's own type literals, JSON is a convenient language for representing data in web applications without imposing additional parsing complexity. All of Flocking's musical primitives are expressed as trees of JSON objects. These objects can be easily serialized, traversed, manipulated, and merged with other objects. In comparison to other music programming environments, which often describe themselves as functional or object-oriented, Flocking weaves the two approaches together in a manner that could be called "document-oriented."

2.4 Unit Generator Definitions

Musicians working with Flocking don't typically instantiate unit generators directly. Instead, they compose JSON objects into trees. Each node in the tree, called a *unit generator definition* (ugenDef), describes a unit generator instance and its connection to others in the signal-processing graph. A ugenDef includes the following information:

1. the type of unit generator to be instantiated
2. a named set of inputs (key/value pairs), which can consist of either literal values (floats) or other unit generator specifications
3. the rate at which the unit generator will be evaluated (audio, control, or constant); this defaults to `"audio"` if omitted
4. a named set of static options, which describe how the unit generator should be configured

Below is a simple example of a sine wave oscillator, illustrating how Flocking unit generators are defined in JSON:

```
{
  ugen: "flock.ugen.sinOsc",
  rate: "audio",
  inputs: {
    freq: 440,
    mul: 0.25
  },
  options: {
    interpolation: "linear"
  }
}
```

Unit generator types are expressed as dot-separated strings called *key paths* or *EL expressions*. These strings are bound to creator functions at instantiation time by Flocking. All type expressions refer to a global namespace hierarchy so that developers can easily contribute their own unit generator implementations (using their own namespace to avoid conflicts) and have the Flocking framework manage them in the same manner as any of the built-in types.

2.5 Synth Definitions

A collection of unit generator definitions form the basis of a *synth definition* (synthDef). Synth definitions describe a complete instrument to be instantiated by the Flocking framework. Synths typically include a connection to an output bus—either the speakers or one of the environment's shared "interconnect" buses. In this respect, Flocking's architecture is inspired by the SuperCollider server [8, pp.25]. Here is a simple example of a synthDef that outputs two sine waves, one in each stereo channel:

```
{
  ugen: "flock.ugen.out",
  sources: [
    {
      ugen: "flock.ugen.sinOsc"
    },
    {
      ugen: "flock.ugen.sinOsc",
      freq: 444
    }
  ]
}
```

This example also illustrates a key aspect of Flocking's interpreter and its document-merging approach. In the case of the first unit generator, we have omitted all input values. When the synth is instantiated, it will automatically be given a frequency of 440 Hz and an amplitude of 1.0. This is due to the fact that every built-in unit generator declares a set of default values. The Flocking interpreter, prior to instantiating the unit generator, will merge the user's ugenDef values on top of the defaults. If a property is omitted, the default value will be retained; if a user specifies a property, it will be used in place of the default. To save typing, the interpreter will also handle input names correctly when they aren't nested inside an "inputs" container. Notably, this defaulting and permissiveness is implemented in a publicly visible way (as JSON *defaults specifications*), helping to ensure that these programming conveniences wont't restrict interoperability with other tools.

To instantiate a Synth, its creator function must be called. In Flocking, a component creator function typically takes only one argument—the component's *options* structure—and returns an instance of the component. For all synths, the options object must include a synthDef as well as any other settings needed to appropriately configure the synth instance. Figure 2 shows how a Flocking synth is created programmatically.

By default, synths are automatically added to the tail of the Environment's list of nodes to evaluate, so they will start sounding immediately if the Environment has been started.

```
var synth = flock.synth({
  synthDef: {
    id: "carrier",
    ugen: "flock.ugen.sinOsc",
    freq: 440,
    phase: {
      id: "mod",
      ugen: "flock.ugen.sinOsc",
      freq: 34.0,
      mul: {
        ugen: "flock.ugen.sinOsc",
        freq: 1/20,
        mul: Math.PI
      },
      add: Math.PI
    },
    mul: 0.25
  }
});
```

Figure 2. Instantiating a custom phase modulation synth.

2.6 Updating Values

Once a synth has been instantiated, its inputs can be changed on the fly. Flocking supports a highly dynamic signal processing pipeline; unit generators can be added or swapped out from a synth at any time, even while it's playing. Behind the scenes, everything in the signal graph is a unit generator, even static values.

In order to direct changes at a particular unit generator, it has to be given an identifying name. In the example shown in figure 2, the carrier and modulator unit generators are each given an `id` property that exposes them publicly. These names represent "cutpoints" into the overall tree that provide easier access to a particular unit generator. Synths keep track of all their named unit generators and provide `get` and `set` methods for making programmatic changes to their inputs.

Changes can be targeted at any unit generator within the tree using key path expressions. Here is an example of how changes can be made to different points in the unit generator tree with a single call to `Synth.set()`:

```
synth.set({
  "carrier.freq": 220,
  "mod.mul.freq": 1/30
});
```

This example lowers the frequency of the carrier oscillator by an octave while simultaneously slowing down the rate at which the modulator's amplitude is oscillating.

This hierarchical path-based scheme for addressing Flocking's graph of signal generators is inspired by Open Sound Control's *addresses*, which provide a similar means for specifying arbitrary message targets within a tree. Indeed, OSC messages can be easily adapted to Flocking change specifications; this is accomplished with only a few lines of code in the Flocking OSC library. [4]

3. SCHEDULING

3.1 Unit Generators Represent Change

Modelling the architectural distinction between different types of changes that occur at varying time scales is a common challenge faced by computer music systems. Such changes include:

1. highly optimized data flow-based changes that occur at the signal level

2. value or instrument changes scheduled at fixed or indeterminate rates (a "score")

3. messages or events sent between objects in an object-oriented system

4. user-triggered events from an OSC or MIDI controller, or from graphical user interface components such as buttons and knobs

Different systems take markedly different approaches to modelling these distinctions. Flocking attempts to unify the means for expressing both micro- and macro-level changes in a composition. Where other systems create a fundamental semantic and syntactic distinction between different sources of change (e.g. unit generators vs. patterns in SuperCollider), instruments and scheduled events alike are specified in Flocking as a tree of unit generators. The primary difference is the rate at which these unit generators are evaluated. This allows the same instruments that are used to define the note-level timbre and texture of a piece to be reused when shaping the larger-scale phrasing and structure of the music. Figure 3 provides an example of how changes are scheduled using Flocking's declarative scheduler to create a simple drum machine.

This example assumes that there is already a synth running (named "drumSynth"), which will produce a drum sound whenever its trigger input changes. First, we instantiate an asynchronous tempo scheduler—a type of scheduler that runs outside of the sample generation pipeline and that accepts time values specified in beats per minute. Currently there are only asynchronous Schedulers in Flocking; a sample-accurate implementation is in the planning stages.

The details of the desired changes are specified in the "score" section of the example. This particular score is defined with the following parameters:

- it should be repeatedly applied, every beat

- each change should be targeted at a particular instrument (specified by name)

- the value of each change should be determined by evaluating the supplied synthDef

[4] https://github.com/colinbdclark/flocking-osc/

```
flock.scheduler.async.tempo({
  bpm: 180,

  score: {
    interval: "repeat",
    time: 1,
    change: {
      synth: "drumSynth",
      values: {
        "trig.source": {
          synthDef: {
            ugen: "flock.ugen.sequence",
            list: [
              1, 1, 0, 1,
              1, 0, 1, 0
            ],
            loop: 1
          }
        }
      }
    }
  }
});
```

Figure 3. Scheduling changes with the Flocking Scheduler.

In Flocking, synths can be evaluated at different rates, including at audio, control, scheduled, and demand rate. The scheduler automatically takes care of parsing the JSON-based *change specification*, producing a *value synth* running at the specified scheduled rate, and targeting its stream of changes to the desired instrument synth. In figure 3 above, the scheduled synth will be evaluated on every beat. It produces values using a simple `sequence` unit generator, which cycles through a list of numbers in order.

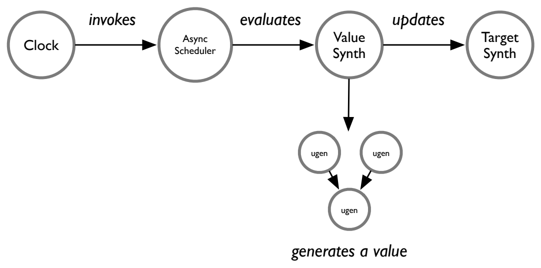


Figure 4. Diagram showing the runtime structure of Flocking's declarative scheduler.

A full version of the example in figure 3, which also illustrates how synths and schedulers can be woven together in an entirely declarative way, is available on Github [5] .

[5] https://github.com/colinbdclark/flocking-examples/blob/master/drum-machine/drum-machine.js

3.2 Rationale

This approach was inspired by an insight in James Tenney's *Computer Music Experiences* [9], where he points out the conceptual similarity between the macrostructure of a composition—events that occur over the duration of a piece of music—and the changes that occur at the microlevel of unit generators. In the early 1960s, Tenney attempted to use Music IV's unit generator system as the basis for algorithmically specifying the large-scale time structure of his compositions. He commented that the instruments "produced results that were quite interesting to me, but it was not very efficient to use the compiler itself for these operations. . . [requiring] a separation between the compositional procedures and the actual sample-generation" [9, p.41–42]. This suggests that the architectural rift between composition-level and signal-level changes, which has been inherited by several generations of computer music systems since the 1960s, was born out of early performance issues.

Few would doubt that the performance factors of today's computer music systems are the same as they were on early mainframe systems, and the elegance and power of using unit generator for both signal- and composition-level changes is worth revisiting. Aside from simplicity, one of the main advantages of Flocking's approach to declarative scheduling is that it offers the potential to actually improve performance in the long run. A typical problem with computer music schedulers is ensuring that whatever work a user schedules is deterministic and optimized for real-time performance. Schedulers either have to trade off expressivity, limiting the types of changes that can be scheduled (such as with the Web Audio API's AudioParams), or leave it entirely up to the user to implement event producers that are sufficiently optimized. Flocking attempts to help users express changes in a way that can be optimized automatically by the framework. Unit generators are explicitly designed to be used in a real-time constrained context. As a result, the Flocking interpreter is free to take a scheduled synthDef and, if appropriate, inject its unit generator tree directly into the signal path of the target synth, ensuring that all changes occur with as little overhead as possible.

SynthDefs are similarly used in Flocking's MIDI and OSC libraries to define transformations between incoming control values and the inputs of an audio synth.

4. CURRENT STATE

4.1 Relationship to the Web Audio API

Flocking currently makes limited use of the built-in native audio processing nodes in the W3C's Web Audio API [6] . It is the opinion of the authors that the version of the Web Audio API shipping in browsers today is insufficient to support the expressivity required by creative musicians without the support of additional libraries. Many of the limitations of the API are outlined in detail in [10]. Web Audio currently provides limited options for web developers who want to create their own custom synthesis algorithms in

[6] http://www.w3.org/TR/webaudio/

JavaScript and expect them to perform well. In particular, it is difficult to mix native and JavaScript-based nodes in the same signal graph without imposing latency and synchronization issues.

Flocking predates the first Web Audio API implementation, and was architected specifically to allow web developers to contribute their own first-class signal processing implementations in an open way. As a result of this philosophy, and due to the performance and developer experience issues of the current Web Audio specification, Flocking uses only small parts of the API. Instead, it takes full control of the sample-generation process and provides musicians with an open palette of signal-generating building blocks that can be used to assemble sophisticated digital instruments.

4.2 Comparison with Web Audio Libraries

Several other libraries also take a similar "all JavaScript" approach. *Gibberish* [11] and *CoffeeCollider* [12] are two prominent alternatives to Flocking. CoffeeCollider attempts to replicate the SuperCollider environment as closely as possible using the CoffeeScript programming language [13], while Gibberish takes a more traditional object-oriented approach. Although these environments each offer their own unique features, neither has attempted to stray far from the conventional models established by existing music programming environments.

Flocking, too, has taken architectural inspiration from several existing music programming systems, particularly the design of the SuperCollider 3 synthesis server. Flocking shares with it a simple "functions and state" architecture for unit generators, as well as a strict (conceptual) separation between the realtime constraints of the signal-processing world and the more dynamic and event-driven application space, manifested in the architectural distinction between unit generators and synths [14, pp. 64].

4.3 Performance

Much has been written about web audio performance issues related to the current generation of JavaScript runtimes generally (lack of deterministic, incremental garbage collection) and the Web Audio API specifically (the requirement for ScriptProcessorNodes to run on the main browser thread) [10, 11]. If history is any indication, it seems likely that the performance characteristics of the JavaScript language will keep improving as the browser performance wars continue to rage between Mozilla, Google, and Apple. In addition, Web Worker-based strategies for sample generation are currently being discussed for inclusion in the Web Audio API specification [7], which will significantly improve the stability of JavaScript-based signal generators.

In the interim, many claims have been made about the relative performance merits of various optimization strategies used in toolkits such as Gibberish [11]. Most of these claims, however, focus on micro-benchmarks that measure the cost of small-scale operations in isolation, rather than taking into account the performance of real-world signal processing algorithms.

[7] https://github.com/WebAudio/web-audio-api/issues/113

Avoiding the temptation to focus on micro-benchmarking and premature optimization, the approach we have taken in Flocking is to build an architecture and framework that can serve as a flexible, long-term foundation on which to continually evolve new features and improved performance. Significant effort has been invested in developing automated unit and performance tests for Flocking that measure the real-world costs of its approach.

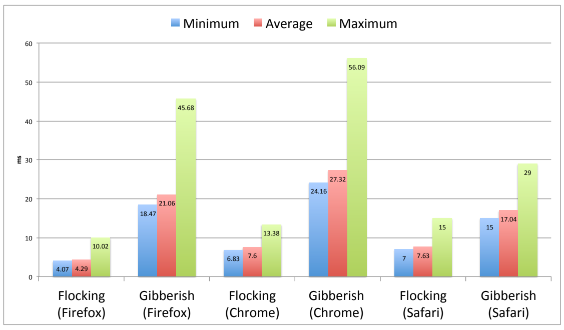


Figure 5. A comparison of performance between Flocking and Gibberish. Smaller bars are faster.

With just-in-time compilers such as Google's V8 [8] and Mozilla's IonMonkey [9], we believe that real-world performance is best achieved by using simple algorithms that represent stable "hot loops" that can be quickly and permanently compiled into machine code by the runtime. The risk of micro-optimization efforts such as the code-generation techniques promoted by Gibberish is "lumpy" (i.e. of an unpredictable duration) real-world performance caused by the JavaScript runtime having to re-trace and recompile code. This is particularly an issue when code needs to be dynamically generated and evaluated whenever the signal graph changes, such as the introduction of new synths or unit generators into the pipeline. Flocking avoids this risk while maintaining competitive performance by using a simple algorithm for traversing and evaluating unit generators. Synth nodes and unit generators are stored in flat, ordered lists. Flocking is able to quickly iterate through these lists and evaluate each signal generator in order. Synth nodes and unit generators can be added or removed from the pipeline at any time without forcing the JavaScript runtime to spill its caches when evaluating a new piece of code. This helps to ensure that Flocking's performance profile remains stable and consistent at runtime.

Despite very little optimization effort to date, preliminary benchmarks [10] suggest that Flocking's approach is promising both from the perspective of good performance as well as greater simplicity and maintainability in comparison to systems that use more complex code generation techniques. Figure 5 shows a simple test where one second's worth of samples were generated and timed for an FM synth consisting of three sine oscillators. This test

[8] https://code.google.com/p/v8/
[9] https://wiki.mozilla.org/IonMonkey/Overview
[10] https://github.com/colinbdclark/webaudio-performance-benchmarks

was performed 10000 times to illustrate realistic VM behaviour. The minimum, average, and maximum times are graphed in milliseconds. The tests were carried out on an Apple MacBook Pro laptop with a 2.3 GHz Intel Core i7 processor. Many factors can influence benchmark results, but Flocking's performance appears to be significantly better than Gibberish on every browser.

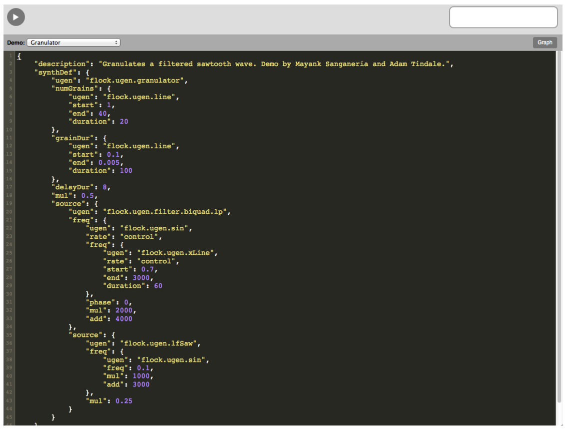
4.4 The Flocking Playground

Flocking's data-oriented approach can be useful for a variety of musical and social purposes. For example, a generative music application can algorithmically produce JSON synthDefs on the fly that introduce new instruments or variations on existing instruments into the system. Similarly, a visualization and editing environment can traverse the source code of a synthDef and produce a rendering that allows users to inspect or edit their instruments visually.

The Flocking Playground (see figure 6) is a simple web-based development environment that serves as an evolving platform for showing Flocking's features and approach. It provides the ability to:

- browse, audition, edit, and share links to a variety of Flocking demos
- develop new instruments and compositions in the integrated code editor
- see a synchronized visual rendering of a synth's source code

The Playground's graphical mode parses a user's JSON SynthDef specifications and renders them on the fly using a combination of HTML, CSS, and SVG into a flow-based diagram that illustrates the synth's structure and signal flow.



```
{
    "description": "Granulates a filtered sawtooth wave. Demo by Mayank Sanganeria and Adam Tindale.",
    "synthDef": {
        "ugen": "flock.ugen.granulator",
        "numGrains": {
            "ugen": "flock.ugen.line",
            "start": 1,
            "end": 40,
            "duration": 20
        },
        "grainDur": {
            "ugen": "flock.ugen.line",
            "start": 0.1,
            "end": 0.005,
            "duration": 100
        },
        "delayDur": 8,
        "mul": 0.5,
        "source": {
            "ugen": "flock.ugen.filter.biquad.lp",
            "freq": {
                "ugen": "flock.ugen.sin",
                "rate": "control",
                "freq": {
                    "ugen": "flock.ugen.xLine",
                    "rate": "control",
                    "start": 0.7,
                    "end": 3000,
                    "duration": 60
                },
                "phase": 0,
                "mul": 2000,
                "add": 4000
            },
            "source": {
                "ugen": "flock.ugen.lfSaw",
                "freq": {
                    "ugen": "flock.ugen.sin",
                    "freq": 0.1,
                    "mul": 1000,
                    "add": 3000
                },
                "mul": 0.25
            }
        }
```

Figure 6. A screenshot of Flocking's interactive programming environment.

The Flocking Playground is built with Fluid Infusion [11], a JavaScript framework that supports end-user personalization and authoring [15]. Infusion's infrastructure for relaying, transforming, and firing changes across diverse models within an application are critical for maintaining

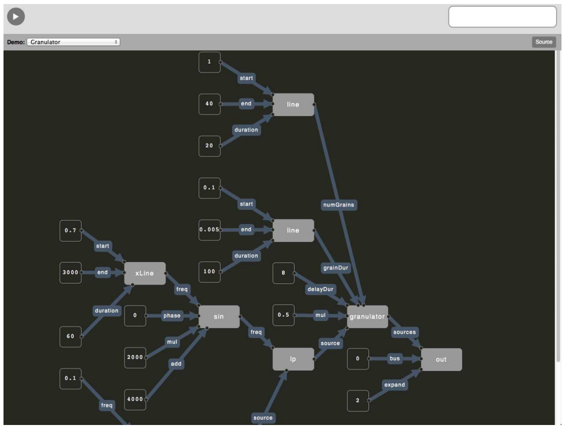


Figure 7. A screenshot of the Playground's visual view.

synchronization between the graphical and source views of the Playground. Infusion continues to be a source of significant architectural inspiration for Flocking, and the two frameworks share a common philosophy and approach.

4.5 Greater Web Audio Integration

Due to the fact that Flocking takes control of the sample generation process directly, it uses very few features of the W3C Web Audio API. As the specification evolves, plans are underway to adopt more of its features in Flocking. At the moment, Flocking consists of a single ScriptProcessorNode that is connected to the Web Audio API's destination sink. Limited support for injecting native Nodes before and after the Flocking script node is available, opening up the possiblity of using nodes such the MediaStreamSource, Panner, and Analyser nodes in tandem with Flocking. Nonetheless it remains difficult to build complex graphs that mix native and Flocking-based processors.

We are in the midst of planning an updated version of the Flocking architecture that allows Flocking unit generators to be interleaved freely with native Web Audio nodes. This approach will introduce a proxy unit generator type that adapts inputs between a native node and a Flocking unit generator.

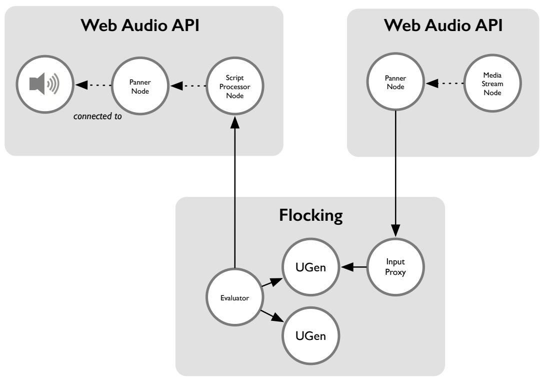


Figure 8. A diagram showing how Flocking will support mixing unit generators with native Web Audio API nodes.

This architecture change will also help prepare Flocking

[11] http://fluidproject.org/products/infusion

for Web Worker-based ScriptProcessorNodes, which are planned for a future version of the Web Audio specification [12] .

5. CONCLUSIONS

Flocking is a new framework for computer music composition that leverages existing technologies and ideas to create a robust, flexible, and expressive system. Flocking combines the unit generator pattern from many canonical computer music languages with Web Audio technologies to allow users to interact with existing and prospective web technologies. Users interact with Flocking using a declarative style of programming.

The benefit of Flocking's approach, when considering various examples of web development environments using both text and visual idioms, has been demonstrated. Flocking provides users with a clear and semantic way to represent the materials of digital music, a promising framework for growing new features and tools, and a light performance footprint.

6. REFERENCES

[1] R. B. Dannenberg, "A language for interactive audio applications," in *Proceedings of the International Computer Music Conference*. International Computer Music Society, 2002.

[2] G. Wang, P. R. Cook *et al.*, "Chuck: A concurrent, on-the-fly audio programming language," in *Proceedings of the International Computer Music Conference*. Singapore: International Computer Music Association (ICMA), 2003, pp. 219–226.

[3] J. McCartney, "Supercollider: a new real time synthesis language," in *Proceedings of the International Computer Music Conference*, 1996.

[4] Y. Orlarey, D. Fober, and S. Letz, "FAUST: an efficient functional approach to DSP programming," in *New Computational Paradigms for Computer Music*, G. Assayag, A. Gerzso, and IRCAM, Eds. Delatour, 2009, pp. 65–96.

[5] M. Wright, "Open sound control-a new protocol for communicationg with sound synthesizers," in *Proceedings of the 1997 International Computer Music Conference*, 1997, pp. 101–104.

[6] J. P. ffitch, V. Lazzarini, and S. Yi, "Csound6: old code renewed," in *LAC: Linux Audio Conference 2013: Proceedings*, I. m zmölnig and P. Plessas, Eds. Graz, Austria: Institute of Electronic Music and Acoustics (IEM), University of Music and Performing Arts Graz, May 2013, pp. 69–75. [Online]. Available: http://opus.bath.ac.uk/37389/

[7] J. W. Lloyd, "Practical advantages of declarative programming," in *Joint Conference on Declarative Programming, GULP-PRODE*, vol. 1, 1994, pp. 18–30.

[8] S. Wilson, D. Cottle, and N. Collins, *The SuperCollider Book*. The MIT Press, 2011.

[9] J. Tenney, "Computer music experiences, 1961–1964," *Electronic Music Reports*, vol. 1, pp. 23–60, 1969.

[10] L. Wyse and S. Subramanian, "The viability of the web browser as a computer music platform," *Computer Music Journal*, vol. 37, no. 4, pp. 10–23, 2013.

[11] C. Roberts, G. Wakefield, and M. Wright, "The web browser as synthesizer and interface," in *Proceedings of New Interfaces for Musical Expression (NIME)*, Daejeon, Seoul, 2013, pp. 313–318.

[12] N. Yonamine, "Coffeecollider," http://mohayonao.github.io/CoffeeCollider/ (Retrieved April 1, 2014).

[13] J. Ashkenas *et al.*, "Coffeescript," http://coffeescript.org/ (Retrieved April 1, 2014).

[14] J. McCartney, "Rethinking the computer music language: Supercollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[15] C. Clark, A. Basman, S. Bates, and K. G. Markus, "Enabling architecture: How the GPII supports inclusive software development," in *Proceedings of the International Conference on Human-Computer Interaction*, 2014, in Press.

[12] https://github.com/WebAudio/web-audio-api/issues/113