Web Audio Core Sonification Library

Description

This document describes the requirements and design of a small library to support portable Web Audio API-based sonification instruments. It is very inspired by Flocking, but attempts to provide a minimalistic environment for instantiating and wiring native Web Audio nodes into signal-processing graphs and updating values on them.

The goal of this library is to provide a small core for developing an inclusive sonification toolkit that is usable across projects such as FLOE, PhET, Flocking, and others.

Requirements

- Support for modular synthesis-style signal processing graphs, which are composed of low-level signal units (*Nodes*, in Web Audio terminology)
    - This should include signal processing primitives that are missing or implicit in the Web Audio API, such as envelopes, value nodes, common mathematical operators, etc.
- The ability to specify graphs of Web Audio API nodes declaratively (i.e. without requiring repetitive, imperative code to instantiate or connect nodes)
- Signal graphs should be authorable using visual and textual tools so that non-programmers can design sonification instruments
- **Signal graph specifications** should be easy to serialize, interpret, and modify *as data structures* using standard tools (i.e. JSON)
- Signals should provide viable defaults values such that users need only define their own overrides, which will be merged in by the framework
- It should be possible to export signal graphs created in a sonification authoring tool or editing environment into a format that can be easily instantiated in a "player" environment
- Can enable the creation of separate, higher level tooling that supports personalization and user authoring environments in which "end users" can customize or change a sonification's instrumental characteristics and mappings to data
- Minimal dependence on third party frameworks/libraries—*where such minimalism doesn't end up leading towards the creation of ad hoc or implicit framework functionality*—but which can be viably integrated into larger frameworks such as Flocking and Infusion
- The ability to change live signal graphs declaratively by address (i.e. by targeting signal specification data structures at named paths) such as:
    - inserting new nodes into the graph
    - removing nodes
    - replacing one node with another
    - updating audio parameter values
- The ability to define custom-typed nodes that are implemented using JavaScript-based signal processors (via the current ScriptProcessorNode API and eventually the AudioWorklet API when it is implemented)
    - This should include, for example, the ability for Flocking to implement its own custom "flocking" type node
- Built-in, sensible defaults for all (or as many as is conceivably possible) node parameters and input values, such that nodes can be easily used in an "always live" authoring or playback environment
- The ability to define shared audio buffers that can be loaded from a URL or generated from a set of built-in function generators
- Where appropriately, normalization of Node parameters to signal ranges (e.g. 0-1.0 for AudioBufferSourceNode's loopEnd param) to make these values more easily modulatable and reduce repetitive manual calculation code.

What's Not in Scope (Yet)?

- Scheduling (although we should look at using or extracting a portion of the Bergson scheduler and applying it to the lookahead scheduling scheme for Web Audio described by Chris Wilson
- Support for interconnecting separate, higher-level compositions of signal graph modules in a declarative way (this should be accomplished by Flocking)
- Musical abstraction such as tempo mapping, MIDI, polyphonic synthesizer, etc. (this is also Flocking's realm)
- Data mapping strategies for sonfication (this is acutely needed, but should be the domain of a sonification-specific library that uses this one)
- Signal graph authoring tools, visual or otherwise (also the realm of Flocking and/or a sonification authoring tool)

How Does This Compare to Other Libraries (Tone.js, Flocking, etc.) or the Web Audio API Directly? Why Build Another?

Tone.js is an awesome and widely used Web Audio-based music library. It provides many high-level musical abstractions (patterns, loops, tempo-based scheduling, pre-built synthesizers, etc.). Tone.js makes it very easy to get set up to make generative browser-based music, but its object-oriented abstractions often hide away the modules of synthesis in favour of over-arching musical objects such as Synths and Patterns. It provides no support for authoring, since it is entirely imperative and can not be easily mapped into a portable, declarative specification.

Flocking is a framework for creative sound programming. It is highly declarative in nature, and provide full support for composing signal graphs from modular, low-level pieces without sacrificing higher level abstractions such as synths, note events, and scheduling. However, its current implementation makes it very difficult to declare complex wiring of signals (in, say, a diamond-shaped configuration where a unit generator's output is routed as input to multiple downstream unit generators). Flocking currently also has only rudimentary support for native nodes, making it unsuitable for use in graphics-intensive environments. A future release of Flocking will be updated to use the services provided by this library.

The Web Audio API, used by itself, encourages brittle, highly sequential code that cannot be exported or used outside of the context of the application in which the code was written. It provides no ability to easily modify, rewire, or interpose new nodes on an existing graph. It provides limited means for introspecting an instantiated graph—for example, to determine and render visualizations of the connections between nodes. Notably, the specification itself makes clear that, on its own, the API is unsuitable for serialization or introspection. In other words, the Web Audio API can't be used in environments that support authoring and sharing of synthesis algorithms in a secure and flexible way.

So, in short, there remains an acute need for a low-level, dependency-light library for declaratively defining Web Audio graphs, which can be used as a building block by larger-scope frameworks such as Flocking as well as applications with tight resource constraints such as PhET.

A Proposed Architecture

The requirements of a resource-constrained, real-time application are significantly different from those of an authoring tool that supports an open graph of creators in defining, reusing, and modifying signal graphs. As a result, we aim to provide a three-level architecture consisting of:

1. A very minimal, low-level representation of signal graphs as formal JSON and accompanying runtime (a graph "instantiator") that can be embedded in applications that are strictly sonification "players." This format will not typically be authored by hand by developers, but rather represents a lightweight "compile to" format or raw representation that a) ensures serializability and exchangeability of signal graph material and b) reduces the typical brittleness and sequentiality of raw Web Audio API-based code. This format could conceivably be used for simple box-and-wire visual authoring if required, or can be generated from other tooling (e.g. a Max or SuperCollider patch).
2. An authoring format that defines a more structured data model and component architecture using Fluid Infusion, which will support, in the longer term, a set of more sophisticated graph authoring and visualization tools such as the ones prototyped in the Flocking live playground, and harmonized with the state transfer semantics of the Nexus.
3. A more abstract, sound-oriented JSON-only "hand coding" format intended for users of tools such as Flocking, which will provide greater abstraction from the specifics of #2 while offering a more convenient format for expressing overridable and configurable connections amongst signal processing units and graph fragments. This layer will generate layer #1 documents via the component model of layer #2.

Sketches: Level 1 JSON Format

These examples represent in-progress sketches for a declarative format mapped to the semantics of the Web Audio API, with sufficient abstraction (if any) to make it usable for the expression of complex signal processing graphs.

We assume here, based on a working name for this library of *Signaletic* (from the French *signalétique*, a description; and via Deleuze's *signaletic material*), that the global namespace for this library will be `signal`.

Sawtooth oscillator with an amplitude envelope

```
{
    nodes: {
        carrier: {
            type: "signal.oscillator", // this refers to a built-in Web Audio node (an OscillatorNode)
            shape: "saw",
            frequency: 440
        },

        carrierAmp: {
            type: "signal.gain"
        },

        ampEnv: {
            type: "signal.adsr", // this is a custom JS component that will ship as part of the library

            // These, of course, will have sensible defaults so that a user
            // doesn't have to explicitly define all parameters.
            startLevel: 0.0,
            attackTime: 0.1,
            attackLevel: 1.0,
            decayTime: 0.1,
            sustainLevel: 0.8,
            releaseTime: 0.1,
            releaseLevel: 0.0
        },
    },

    // osc -> gain -> (env) -> destination
    connections: {
        // source: destination
        // Note that here we are explicit about specifying the input and output number.
        // The Web Audio API's Node.connect(node, inputNum, outputNum) method provides defaults for the latter
two arguments,
        // so we can (should?) also reflect this optionality in Signaletics and automatically expand to the 0th
input/output.
        "carrier.outputs.0": "carrierAmp.inputs.0",
        "ampEnv.outputs.0": "carrierAmp.gain",
        "carrierAmp.outputs.0": "output" // "output" is a reserved name representing this graph fragment's "tail
node"

                                 // and which can be routed to the AudioContext's destination node.
    }
}
```
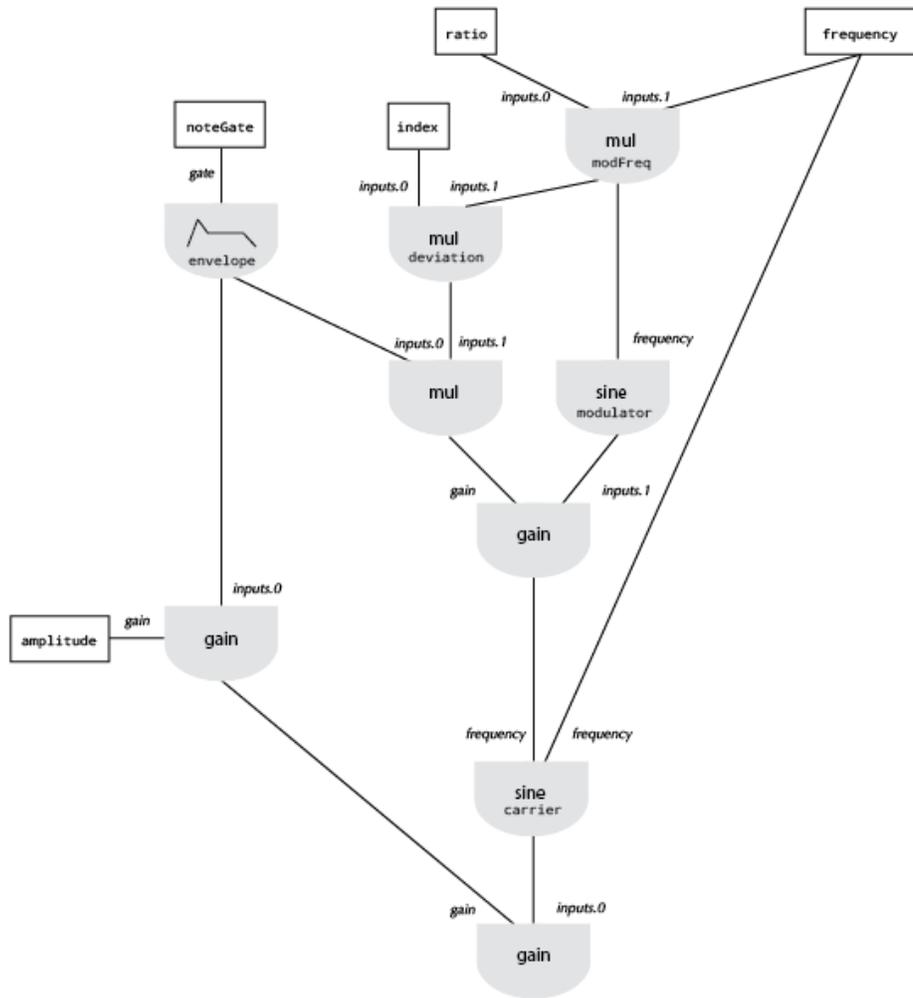
2-op FM Synthesis

This example is a one operator (i.e. one carrier, one modulator, and one amplitude envelope) FM brass instrument created by John Chowning, published in *Computer Music* by Dodge and Jerse. Notably, it illustrates how a node can be connected up as the input to several unit generators, and how multiple Nodes can be hooked up to a single input.

Graph Schematic

Signaletic Low-Level Document Format

```
{
    // Each node in the graph is given a unique name, and are defined declaratively by type.
    // Any "options" (static configuration), such as oscillator waveform type,
    // can be specified within the Node definition.
    nodes: {
        frequency: {
            // This will be implemented using the poorly-named ConstantOutputNode or its polyfill.
            type: "signal.value",
            value: 440
        },

        amplitude: {
            type: "signal.value",
            value: 1.0
        },

        ratio: {
            type: "signal.value",
            value: 2
        },

        index: {
            type: "signal.value",
            value: 5
```

```
        },

        noteGate: {
            type: "signal.value",
            value: 0.0
        },

        envelope: {
            // This will be a custom object that controls the AudioParam
            // it is connected when the "gate" signal transitions above/below 0.
            type: "signal.envGen",

            // This is an custom envelope with user-defined breakpoints.
            // Reusable envelope shapes will be provided and be specified by type name.
            envelope: {
                levels: [0, 1, 0.75, 0.6, 0],
                    times: [0.1, 0.1, 0.3, 0.1],
                    sustainPoint: 2
            }
        },

        modulatorFrequency: {
            // This will be implemented as a two-input GainNode
            // that assigns its second input to the "gain" AudioParam
            type: "signal.mul"
        },

        deviation: {
            type: "signal.mul"
        },

        modulatorAmplitudeGain: {
            type: "signal.mul"
        },

        modulator: {
            type: "signal.oscillator",
            shape: "sine"
        },

        modulatorOutputGain: {
            type: "signal.gain"
        },

        carrier: {
            type: "signal.oscillator",
            shape: "sine"
        },

        envelopeGain: {
            type: "signal.gain"
        },

        outputGain: {
            type: "signal.gain"
        }
    },

    // The "connections" block declares the wiring
    // between Nodes in the graph.
    // Connections are declared with the source Node (i.e. the Node you want to connect up to the input of
another) as the key,
    // and a path into the target node on the right (or an array of them for multiple connections).
    connections: {
        "noteGate.outputs.0": "envelope.gate",

        // This is an example of one node being connected
        // as an input to two others.
        "frequency.outputs.0": [
            "carrier.frequency",
            "modulatorFrequency.inputs.0"
        ],

        "ratio.outputs.0": "modulatorFrequency.inputs.1",

        "modulatorFrequency.outputs.0": [
            "deviation.inputs.0",
            "modulator.frequency"
        ],

        "index.outputs.0": "deviation.inputs.1",

        "envelope.outputs.0": [
            "modulatorAmplitudeGain.inputs.0",
```

```
        "envelopeGain.inputs.0"
    ],

    "deviation.outputs.0": "modulatorAmplitudeGain.inputs.1",
    "modulator.outputs.0: "modulatorOutputGain.inputs.0",
    "modulatorAmplitudeGain.outputs.0: "modulatorOutputGain.gain",
    "modulatorOuptutGain.outputs.0: "carrier.frequency",
    "amplitude.outputs.0: "envelopeGain.gain",
    "envelopeGain.outputs.0: "ouptutGain.gain",
    "carrier.outputs.0: "outputGain.inputs.0",
    "outputGain.outputs.0": "output"

    }
}
```

(Mostly) Plain Web Audio API Version

```
/*
Here, we are assuming that:
    * Nodes such as ConstantOutputNode have been implemented or polyfilled
    * Crucial wrappers such as EnvGen have been implemented (though their APIs may be fuzzy)
    * Other "semantic" Nodes such as Value, Mul, Add, etc. are not present
*/

var ac = new (window.AudioContext || window.webkitAudioContext)();

var frequency = new ConstantOutputNode(ac, {
    offset: 440
});

var amplitude = new ConstantOutputNode(ac, {
    offset: 1.0
});

var ratio = new ConstantOutputNode(ac, {
    offset: 2
});

var index = new ConstantOutputNode(ac, {
    offset: 5
});

var noteGate = new ConstantOutputNode(ac, {
    offset: 0.0
});

// How will we implement this?
// Using the current Web Audio API with only native nodes,
// it will need to live on the "client" side, and thus can't be triggered
// by an audio signal. So either we inefficiently implement Flocking-style
// envelopes as ScriptProcessorNodes, or can't support "control voltage"-style
// triggering of envelopes. Either seems workable for the (hopefully short)
// interim period until AudioWorklets have been implemented.
var envelope = new EnvGen({
    levels: [0, 1, 0.75, 0.6, 0],
        times: [0.1, 0.1, 0.3, 0.1],
        sustainPoint: 2
});

var modulatorFrequency = new GainNode(ac);

var deviation = new GainNode(ac);

var modulatorAmplitudeGain = new GainNode(ac);

var modulator = new OscillatorNode(ac, {
    type: "sine"
});

var modulatorOutputGain = new GainNode(ac);

var carrier = new OscillatorNode(ac, {
    type: "sine"
});

var envelopeGain = new GainNode(ac);

var outputGain = new GainNode(ac);

noteGate.connect(envelope.gate, 0);
```

```
frequency.connect(modulatorFrequency, 0, 0);
ratio.connect(modulatorFrequency.gain, 0);

modulatorFrequency.connect(deviation, 0, 0);
index.connect(deviation.gain, 0);

envelope.connect(modulatorAmplitudeGain, 0, 0);
deviation.connect(modulatorAmplitudeGain.gain, 0);

modulatorFrequency.connect(modulator.frequency, 0);

modulator.connect(modulatorOutputGain, 0, 0);
modulatorAmplitudeGain.connect(modulatorOutputGain.gain, 0);

modulatorOutputGain.connect(carrier.frequency, 0);
frequency.connect(carrier.frequency, 0);

envelope.connect(envelopeGain, 0, 0);
amplitude.connect(envelopeGain.gain, 0);

envelopeGain.connect(outputGain.gain, 0);
carrier.connect(outputGain, 0, 0);
```

Other Examples We Need

- Multiple inputs/output and channel up/down mixing
- Audio buffers
- Implementations of the routing examples in the Web Audio API specification
- A cyclic node arrangement using a DelayNode (From the spec: "It is possible to connect an `AudioNode` to another `AudioNode` which creates a cycle: an `AudioNode` may connect to another `AudioNode`, which in turn connects back to the first `AudioNode`. This is allowed only if there is at least one `DelayNode` in the *cycle*")

Notes/Issues/To Do

- We at least need a way of abstracting the inputs and outputs (i.e. routing to the AudioContext's destination) such that "graph fragments" could be constructed and wired up to each other. This is essential for any kind of higher-level composition elsewhere (this will happen in layers 2/3)
- Elaborate on the the three-part architecture discussed with Antranig on Jan 20, 2017:
    - A regularized, low-level "compile target" document format that does not include connection references, automatic expansion, anonymous nodes, or other conveniences intended for hand-coders but which may risk introducing irregularities or additional processing requirements that reduce compileability or interoperability
    - An Infusion-supported component format that can be used to support authoring tools; this format can be compiled down to the low-level representation
    - A high-level, textual coding representation roughly of the form described above, but with the introduction of a better means for naming and composing "graph fragments"