Interactive Music with Tone.js

Yotam Mann
231 Bowery
2nd Floor
New York, NY
yotammann@gmail.com

ABSTRACT

This paper discusses the features, architecture and implementation of Tone.js, a Web Audio framework to facilitate the creation of interactive music specifically suited to the affordances of the browser.

Categories and Subject Descriptors

J.1.2 [**Human-centered computing**]: Web-based interaction; L.7 [**Applied computing**]: Sound and music computing

General Terms

Documentation, Performance, Design, Human Factors

Keywords

Signal Processing, Synthesis, Effects, Music, Web Audio API, Library

1. INTRODUCTION

Tone.js is a Web Audio framework for creating interactive music in the browser. There are two expressions in that sentence that warrant definition: a "framework" is distinguished from a "library" in that a framework, in addition to functionality, prescribes a structure or architecture to the code [10]. Tone.js provides library modules and abstractions, but also presents a way of using those modules in the context of arranging and producing music. "Interactive music," according to Todd Winkler, refers to a composition or improvisation in which a user/performer's actions affect music generated by a computer [19]. This broad definition encompasses creating interactive, adaptive, and generative music that responds to user input, which is precisely what Tone.js was built for. Tone.js source code is available under the MIT license on github [11].

Towards the goal of creating interactive music, Tone.js' development was guided by three tenets: **musicality** (such as the ability to define scores and synthesizers as JSON objects with note names and rhythmic notation), **modularity** (exemplified in the dozens of signal processing and synthesis building blocks) and **synchronization** (the ability to use those building blocks to coordinate sounds and events along a shared timeline).

`Tone.LFO`, a low frequency oscillator, illustrates these three ideas in a single class. The `start` method accepts musical time as an argument. `Tone.Time` is a tempo-relative encoding of time that can be used, for example, to describe a quarter note duration: `"4n"`. Every method that accepts a time also accepts `Tone.Time`. Secondly, using Tone.js' modular signal processing components, the output of the LFO can be scaled to any range linearly or exponentially. Whereas detune cents might be in the range of -50 to 50 and interpolated linearly, a filter cutoff frequency is perceived exponentially and takes on values anywhere between 20 and 20,000. Lastly, `Tone.LFO` is synchronizable to the global transport; its `start` method can be triggered by a call to the transport's `start` method making it easy to start all LFO instances at once. Additionally, the LFO's frequency can be harmonized to the transport's tempo and even change when the global tempo changes, so that a panner LFO with a speed of `"2t"` (a half-note triplet) would remain in that ratio even as the tempo curves from 120 bpm (beats per minute) to 60 bpm.

2. HIGH-LEVEL ARCHITECTURE

Tone.js' architecture aims to be familiar to audio engineers and musicians coming from DAWs (Digital Audio Workstations) by including features such as grid-relative timing, send and receive buses, a master output channel, and a global transport.

2.1 Musical Time

With `Tone.Time`, `AudioContext` time can be expressed in tempo-relative terms that are translated into seconds with the `toSeconds` method. Delay times, for example, can be expressed in terms of beats: `"4n"` would translate to 0.5 seconds at 120 bpm.

`Tone.Time` takes on a few forms. Numbers are taken literally as seconds. Notation-style strings, inspired by Max/MSP's metrical timing [8], such as `"8n"` for an eighth-note or `"4t"` for a quarter-note triplet, will be evaluated against the current tempo to deduce the time in seconds. When represented as a colon-separated list, `Tone.Time` is interpreted as "bars:beats:sixteenths". This notation is similar to Ableton Live's transport time representation [4]. This sort of timing is useful for referring to events on a longer time-scale such as scheduling a chorus to start at measure 32: "32:0:0". Strings ending in "hz" (Hertz) will also be converted into seconds with toSeconds. "4hz" is equivalent to 0.25 seconds.

It is often useful to schedules values relative to the current AudioContext time. Prefixing any of the above representations with a plus sign as a string ("+") will add the AudioContext's currentTime to the following value. Furthermore, toSeconds (and any parameter which accepts Tone.Time) can also evaluate mathematical expressions with any of the above representations. toSeconds("(7 / 4) * 8n") equals an 8th-note septuplet.

Most classes will defer evaluating the Tone.Time argument until just before execution in order to reflect the current meter, therefore accommodating tempo-curves and time signature changes. For example, any of the values of Tone.Envelope, an ADSR (Attack Decay Sustain Release) envelope, can be expressed as Tone.Time and will stay tempo-relative even as the tempo changes.

2.2 Buses

In addition to making it easy to share a single reverb effect across many audio nodes, buses also promote loose-coupling between audio modules. Sending 50% of a synthesizer signal to a reverb bus, and in a separate module, receiving all signals routed to "reverb" would be written like this:

```
synth.send("reverb", 0.5);
//...in a separate audio module
reverbEffect.receive("reverb");
```

There are a few advantages to this approach over directly connecting synth to reverbEffect. Firstly, dependency management is greatly simplified in that reverbEffect does not necessarily have to be loaded and evaluated at the time that synth is connected to the reverb bus. Additionally, in a modularized JavaScript application without any global objects, synth does not need to obtain a reference to reverbEffect, further aiding in modularization. This is particularly useful during development when synth may have been written before reverbEffect or visa versa.

2.3 Master Output

The master output is an abstraction on the native AudioDestinationNode. It provides a few conveniences over the native AudioNode: having a wrapper on the AudioDestinationNode makes adjusting the global volume or muting the entire application trivial; also, by placing a node before the final output, global effects, compressors, and limiters can be applied to the entire mix.

2.4 Transport

A single transport is central to many music production environments since it allows for tightly synchronized and coordinated events [3, 6, 5, 8]. While many Web Audio libraries offer a way to schedule audio events along a timeline [14, 13, 18], Tone.Transport differentiates itself in two ways: it schedules callback functions and it allows for tempo-curves. Similar to the browser's native setInterval method, Tone.Transport has a setInterval method that accepts a callback and an interval (in Tone.Time). The callback is invoked at the desired interval right before the time of the event with the sample-accurate AudioContext time passed in as the function's argument:

```
Transport.setInterval(function(time){
    //check application state
    //trigger an event using 'time'
}, "8n"); // invoked every 8th note
```

For interactive music applications, just-in-time execution of scheduled events is advantageous over scheduling events far in advance since it allows the events to reflect the most up-to-date application state, which may be constantly changing based on user input or other factors. Tone.Transport's API also includes a setTimeout method for scheduling single events in the future relative to the current clock time, and a setTimeline method which schedules methods along a loop-able and seek-able global timeline.

Tone.Transport is implemented using a ScriptProcessorNode and a square wave OscillatorNode. The OscillatorNode is set to a small subdivision of a quarter note. The ScriptProcessorNode listens for when the oscillator crosses above zero (a tick) at which point any callbacks that are scheduled for that tick are invoked with the tick's time. This approach is similar to WAAClock [12] but adds an OscillatorNode for tracking ticks instead of computing the tick time at each onaudioprocess event. Using the OscillatorNode makes it trivial to create tempo-curves where the bpm changes gradually over time by simply invoking exponentialRampToValueAtTime on the oscillator's frequency AudioParam. Using an OscillatorNode also allows for scheduling the start and stop methods with sample-level accuracy.

There are some issues with this approach. Invoking deferred callbacks from the audio thread introduces the potential for jitter and glitches as the audio thread is held up by the main thread or visa versa [17], though in practice, this has not been a significant issue. More research needs to be done to find the right buffer amount to handle these issues.

3. LOW-LEVEL BUILDING BLOCKS

Tone.js provides dozens of classes for performing low-level math and logic on signals. In the first example, a combination of Tone.Add and Tone.Multiply are used to scale the range of the oscillator in Tone.LFO to any output range. Tone.js' signal calculations are computed at audio-rate making them suitable to be used for controlling AudioParams. Furthermore, none of the signal classes use any ScriptProcessorNodes which makes them efficient and low-latency.

3.1 Tone.Signal

At the heart of the signal processing modules is Tone.Signal, which provides functionality similar to the native Web Audio AudioParam in that Tone.Signal can be connected to an AudioParam to give audio-rate control over parameters. It provides advantages over the native AudioParam in that mathematical and logical operations can be applied to the signal before being connected to an AudioParam. Additionally, a single Tone.Signal can be connected to multiple AudioParams, allowing for synchronized automations. Just like the AudioParam, Tone.Signal has methods such as setValueAtTime and linearRampToValueAtTime which allow for sample-accurate automation of the signal's output.

Tone.Signal is implemented using a constant signal generator connected to a GainNode. The signal generator outputs an value of 1 at audio-rate that can be scaled to any value using the GainNode's gain AudioParam. Setting the gain to 20, for example, would make the output 20. The

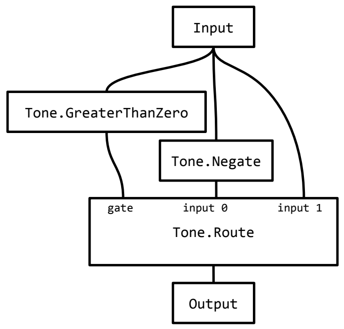

Figure 1: Block diagram of Tone.Abs

constant signal generator of `Tone.Signal` is created by running an `OscillatorNode` through a `WaveShaperNode`; the `WaveShaperNode`'s curve is set to output 1 for all inputs. This generator could also be written with a short, looped `AudioBuffer`. Further research is needed to see which performs better across various platforms. As an optimization, the `Tone.Signal` class shares a single, static generator and therefore each instance of the class is only composed of one `GainNode`.

3.2 Math

Math operators in Tone.js make use of the `GainNode`'s native ability to sum and multiply signals. In the Web Audio API, signal is summed when two `AudioNode`s are connected to the same node [2]; `Tone.Add` uses a `Tone.Signal` connected to a `GainNode` to add a value to any signal connected to the same `GainNode`. `Tone.Multiply`, is similarly simple. Again, using the `GainNode`'s `gain AudioParam`, the incoming signal can be multiplied by a number or signal-rate value.

3.3 Logic

Tone.js includes a-rate boolean operators such as `Tone.Equal`, `Tone.GreaterThan`, and `Tone.LessThan`. These operators rely heavily on the `WaveShaperNode`. The `WaveShaperNode`'s curve in `Tone.EqualZero` outputs 1 when the input is 0 and outputs 0 for all other inputs. A `Tone.Multiply` with a large value is connected before the `WaveShaperNode` to ensure that small input values are scaled far from zero to avoid interpolation by the `WaveShaperNode`. `Tone.Equal` is implemented using `Tone.EqualZero` with `Tone.Subtract` to subtract a value from the input before comparing it to zero.

3.4 Routing

It can be useful to conditionally route audio to or from another `AudioNode`; `Tone.Route` and `Tone.Select` do just that. `Tone.Route` has any number of inputs and only one output. By setting its `gate` to an input number, `Tone.Route` selectively routes that input to the output, stopping all others. `Tone.Select` routes a single input to one of the outputs depending on the value of the `Select`'s `gate`.

Logic and routing operations can be combined in order to perform operations like absolute value. `Tone.Abs` tests if the incoming signal is less than zero, in which case it routes the negated signal to the output, otherwise the output is equal to the input (see Figure 1). `Tone.Min` and `Tone.Max` work in similar ways.

3.5 Native vs ScriptProcessorNode

Web Audio's `ScriptProcessorNode` is one of the most powerful features of the API, but it is not ideal for interactive music. Firstly, the `ScriptProcessorNode` has more latency than the native nodes. In Google Chrome's implementation, the input and output of the `ScriptProcessorNode` are double buffered [17]. This can lead to a quite noticeable delay when the buffer size is large which can detract from the immediacy of interaction and degrade performance [15]. While a single `ScriptProcessorNode` on its own does not contribute a noticeable amount of latency, if Tone.js' modular architecture was implemented entirely with `ScriptProcessorNode`s, the latency would add up to a significant amount. For example, if each of the half-dozen signal processing stages of `Tone.StereoWidener` was composed of a `ScriptProcessorNode` instead of native nodes, the latency for that effect alone would be around a quarter-second (assuming a frame-size of 1024 and a sample rate of 44100kHz). Also, while the non-`ScriptProcessorNode`s are implemented in low-level languages like C/C++, the `ScriptProcessorNode` executes JavaScript which can be much slower than a native language. For these reasons, Tone.js goes to great lengths to find native workarounds for using the `ScriptProcessorNode`.

4. INSTRUMENTS AND EFFECTS

Tone.js combines low-level components into synths and effects. At the time of this writing, Tone.js includes eight instruments including an FM (Frequency Modulation) synthesizer and a Karplus-Strong plucked string modeling synthesizer. Tone.js also has a myriad of audio effects including `Tone.PingPongDelay`, `Tone.Freeverb`, and `Tone.BitCrusher`. Like the signal processing components, all instruments and effects adhere to Tone.js' philosophy of avoiding `ScriptProcessorNode`s for the best latency, performance and scaleability.

4.1 Presets, States and Scores in JSON

The states of instruments and effects in Tone.js can be set through JSON (JavaScript Object Notation) descriptions in the constructor and `set` method.

```
var fastPanner = new Tone.AutoPanner({
    "frequency" : "16n",
    "type" : "square"
});
```

The ability describe a sound generator/processor in an intermediary format like JSON makes it simple to create and share presets. Additionally, it can aid in decoupling the process of building sound generators and composing with those sounds. Using JSON Objects to hold state also opens up the possibility of applying transformations to the state before setting it, such as interpolating between presets or transposing all the values of the states. The JSON descriptions in Tone.js are inspired by the unit generator/score file distinction introduced in MUSIC by Max Mathews which has been a feature in many subsequent computer music languages [1]. An example "orchestra" (to use Csound terminology) with Tone.js might be a `Tone.PluckSynth` connected through a `Tone.Chorus` to the master output. The "score" part of the code for these components might look something like:

```
pluckSynth.set({
    "attackNoise" : 0.8,
    "resonance" : 0.6
});
chorus.set({
    "rate" : 0.75,
    "delayTime" : 3.5,
    "depth" : 0.7,
});
```

Scores are another JSON-based description used in Tone.js. A score can be parsed by `Tone.Note.parse` which schedules note events along the transport's timeline. Scores are represented in JSON with the name of the instrument or channel as the object's key and an array of events as the value. Instruments and other components can then listen for these events using `Tone.Note.route`. `Tone.Note.route` is invoked with a score's keys and an event callback function. The note description in scores is flexible; the only requirement is that it has the time of the event. Additional event data such as note values, durations, etc will be passed to the callback.

```
var score = {
    "drums" : [["0:0","kick"],["0:1","snare"],...
};
Tone.Note.parse(score);
Tone.Note.route("drums",function(time,sample){
    //play drum sample at time
});
```

5. COMPARISON WITH OTHER LIBRARIES

Comparisons can be drawn between frameworks that provide similar functionality such as WAAX [7], Gibberish [13], and Lissajous [14]. These three libraries include a suite of synthesizers and effects as well as their own event schedulers. Tone.js distinguishes itself in three ways. Firstly, Tone.js makes extensive use of `Tone.Signal` which allows for simple, sample-accurate synchronization of multiple `AudioParam`s. For example, when setting the frequency in WAAX's FM Synthesizer (FMKey7), the frequency value needs to be applied to the modulator, carrier, and modulator gain (which controls modulation index), but in Tone.js' implementation only a single frequency signal needs to be changed; this is achieved by routing the frequency signal through the modulation index and harmonicity multipliers before connecting them to the carrier and modulators' frequency `AudioParam`s. This signal processing approach is most similar to Audiolet by Joe Turner [16]. Similar to Tone.js, Audiolet also provides operators like Modulo and Multiply but implements them with the `ScriptProcessorNode` as opposed to Tone.js' use of native components exclusively. This signal-centric approach enables LFOs and other control signals to be easily applied across the library. Tuna.js [9] has a similar focus on enabling LFO control over as many parameters as possible, but, Tuna.js relies on individual `ScriptProcessorNode`s for each LFO which can degrade performance and latency in large applications [15].

Secondly, Tone.js can be easily intermixed with outside libraries and modules. Compared to Gibberish, in which all audio processing is done in a single `ScriptProcessorNode` making it difficult route its audio through other Web Audio components, Tone.js allows users to set the `AudioContext` to make it easy to interconnect Tone.js' modules with other libraries and `AudioNode`s. Lastly, Tone.js' event scheduler uses JavaScript callbacks to schedule events. Lissajous, for example, abstracts away the sequencer callbacks and only gives users the ability to schedule specific events like note triggering. Tone.js' more flexible approach to event scheduling allows anything to be scheduled from the callback. While Lissajous most easily accommodates loop-based events, Tone.js can handle single event, repeated events, and events along a timeline. Also, of the frameworks mentioned, Tone.js' oscillator-based scheduler is the only one that is capable of smooth tempo-curves. These features give Tone.js the flexibility to create a wide range of music.

6. FUTURE WORK AND CONCLUSION

Aside from producing additional effects, instruments and signal processing classes, future work on Tone.js will continue to focus on aiding musicians and composers to make music in the browser. One hurdle in developing music in the browser with Tone.js is the lack of GUIs for exploring and refining musical and sonic parameters. Such tools could export their parameters as JSON which could be saved and incorporated into the composition during development. Another area to explore is using Tone.js as tool for collaboration and jamming. This might take the form of streaming events over WebSockets or sending audio using WebRTC. Challenges in this area include synchronizing transports across clients and network latency.

The introduction of the Web Audio API makes the browser a unique musical medium in that it is the means of production and distribution in one. Tone.js aims to facilitates music which takes advantage of the affordances of the browser to create interactive, collaborative, and generative music in real-time with a user.

7. ACKNOWLEDGMENTS

The author would like to thank Sarah Rothberg, Adrian Freed, Luisa Pereira, Kevin Siwoff, Jason Sigal, Chris Deaner, and Allison Wood for their contributions to this paper and Tone.js.

8. REFERENCES

[1] MUSIC-N. `http://en.wikipedia.org/wiki/MUSIC-N`. Accessed: 2014-10-21.

[2] Web Audio API: W3C Editor's Draft 22 October 2014. `http://webaudio.github.io/web-audio-api/#summing-inputs`. Accessed: 2014-10-25.

[3] Ableton. Ableton Live 9. `https://www.ableton.com/en/live/`. Accessed: 2014-10-08.

[4] Ableton. Ableton Live Manual: Arrangement View. `https://www.ableton.com/en/manual/arrangement-view/#transport`. Accessed: 2014-10-09.

[5] Apple. Logic. `https://www.apple.com/logic-pro/`. Accessed: 2014-10-08.

[6] Avid. Pro Tools. `http://www.avid.com/us/products/family/pro-tools/`. Accessed: 2014-10-08.

[7] H. Choi. WAAX Github Repository. `https://github.com/hoch/WAAX`. Accessed: 2014-10-26.

[8] Cycling74. Tutorial 19: Timing. `http://www.cycling74.com/docs/max5/tutorials/max-tut/basicchapter19.html`. Accessed: 2014-10-08.

[9] Dinahmoe. Tuna.js. `https://github.com/Dinahmoe/tuna`. Accessed: 2014-10-09.

[10] jAndy. What is the difference between a Javascript framework and a library? `http://stackoverflow.com/a/11576088`. Accessed: 2014-10-07.

[11] Y. Mann. Tone.js Github Repository. `https://github.com/TONEnoTONE/Tone.js`. Accessed: 2014-10-27.

[12] S. Piquemal. WAAClock. `https://github.com/sebpiq/WAAClock`. Accessed: 2014-10-09.

[13] C. Roberts. Gibberish.js. `https://github.com/charlieroberts/Gibberish`. Accessed: 2014-10-09.

[14] K. Stetz. Lissajous.js. `https://github.com/kylestetz/lissajous`. Accessed: 2014-10-09.

[15] N. Thompson. You Don't Need That ScriptProcessor. `https://medium.com/web-audio/you-dont-need-that-scriptprocessor-61a836e28b42`. Accessed: 2014-10-21.

[16] J. Turner. Audiolet Github Repository. `https://github.com/oampo/Audiolet`. Accessed: 2014-10-26.

[17] C. Wilson. web-audio-api Github issue: Worker-based ScriptProcessorNode. `https://github.com/WebAudio/web-audio-api/issues/113#issuecomment-54642502`. Accessed: 2014-9-05.

[18] C. Wilson. Web Audio Metronome. `https://github.com/cwilso/metronome`. Accessed: 2014-10-09.

[19] T. Winkler. *Composing Interactive Music: Techniques and Ideas Using Max*. The MIT Press, 2001.