Strategies for Per-Sample Processing of Audio Graphs in the Browser

Charles Roberts
School of Interactive Games and Media
Rochester Institute of Technology
charlie@charlie-roberts.com

ABSTRACT

Due to current browser limitations, most synthesis in the browser is currently performed using the block-rate nodes included in the WebAudio API. However, block-rate processing of audio graphs precludes many types of synthesis in addition to limiting both the accuracy and flexibility of scheduling. We describe alternative strategies for performing efficient, per-sample processing of audio graphs in the browser using the ScriptProcessor node, affording synthesis techniques that are not commonly found in existing JavaScript audio libraries. We introduce a new library, Genish.js, that provides unit generators for common low-level synthesis tasks and acts as a compiler for signal processing functions; this library is a loose port of the Gen framework for Max/MSP. We used Genish.js to update a higher-level library for audio programming, Gibberish.js, realizing improvements to both efficiency and audio quality. Preliminary benchmarks comparing the performance of Genish.js audio graphs to equivalent graphs made with the WebAudio API show promising results.

1. INTRODUCTION

Constraints in current browser implementations of the Web Audio API (WAAPI) have limited the adoption of *per-sample* processing techniques, which in turns constrains the synthesis algorithms available to musicians and sound designers in the browser. We use the term per-sample to describe audio graph processing algorithms where each node (aka *unit generator*) outputs one sample at a time, as opposed to blocks of multiple samples. Currently, the only option for authoring audio graphs using per-sample processing in the browser is to create the entire graph in a *ScriptProcessor* node, which affords authoring sample-level DSP in JavaScript. However, the ScriptProcessor node imposes many penalties in comparison to the other WAAPI nodes, including typical higher CPU usage, increased latency, and the potential for interrupts by other processes running in the main thread of the JavaScript runtime; we further discuss these constraints in Section 2.3. For these reasons, most existing JavaScript audio libraries avoid the ScriptProcessor node, and in doing so discard a wide variety of synthesis techniques that are specific to per-sample processing, as discussed in Section 2. Even libraries that make extensive use of the ScriptProcessor node, such as Flocking.js [1], typically use block-rate processing for efficiency reasons and are unable to run at a block size of `1`.

Our previous research in this area included creating a library, *Gibberish*, that compiled optimized callbacks calculating the output of an audio graph one sample at a time [9]. The library was designed for use in ScriptProcessor nodes and is a critical component of the live-coding environment Gibber[7], as well as a number of other projects [10, 2]. We felt that there was room for significant improvement in Gibberish, which did not effectively reuse code when authoring unit generators (ugens) or provide easy paths for optimization. In this paper we present a new library, *Genish.js*, that provides low-level DSP components that lend themselves well to unit testing, benchmarks, and optimization. Genish was in turn used to create a new version of Gibberish, providing it with a unified system for assembling high-level synthesis and filtering algorithms.

This paper will describe some of the benefits and drawbacks of per-sample processing in the browser and our reasons for continuing to use the ScriptProcessor node, describe Genish.js in detail, and then briefly describe changes to Gibberish and the performance gains achieved using Genish.

2. AFFORDANCES AND CONSTRAINTS OF SINGLE SAMPLE PROCESSING IN THE BROWSER

Per-sample processing affords possibilities for synthesis and scheduling that cannot be realized with block-based signal processing. However, these affordances come with significant drawbacks in efficiency, latency, and potential audio interruptions when working within the browser. Here we describe some of the benefits and constraints of per-sample techniques in the context of current browser technologies.

2.1 Synthesis Using Per-Sample Techniques

Although per-sample processing is often more expensive than block-rate processing, it expands the possibilities of both synthesis and scheduling. For example, one of the most successful digital synthesizers of all times, the Yamaha DX7, featured thirty-two synthesis "algorithms" (modulation paths); every one employed a feedback path that requires ugen output to be processed sample by sample[1]. The FM instrument

[1]For an excellent browser port of the DX7 (via

Web Audio Conference WAC-2017, August 21–23, 2017, London, UK.

in Gibberish features a feedback parameter that approximates this behavior.

Similarly, the difference equations of most filter models also require the ability use feedback loops, which means their creation in the browser depends on using ScriptProcessor nodes to write custom sample processing routines in JavaScript. Our update to Gibberish contains "virtual analog" filter models that depend on per-sample processing.

2.2 Sample-Accurate Scheduling with Temporal Modulation

Per-sample processing also enables interesting possibilities for scheduling. By driving sequencers at audio rate we afford forms of temporal modulation similar to those found in analog modular synthesis systems. Individual sequencers can be driven by separate signals, or can all use a global rate signal as a base for generating other modulated temporal controls (this is the strategy used in Gibber). As one browser-based example, in a composition written for Gibber the band 65daysofstatic ends the piece by taking the global clock signal and modulating it with a sine oscillator, creating a cascade of rhythms gradually fluctuating in time.

Intra-block scheduling is also only possible using per-sample techniques. To clarify, we are referring to scheduling a new event within the current block being rendered (in this case, the block being generated by a ScriptProcessor node). This enables a sequencer firing on sample n to schedule a new event on the next sample processed (n+1), affording sample-accurate scheduling even when using stochastic functions; in the case of Gibberish these functions can modify a single node or the entire topology of the running audio graph from one sample to the next. In comparison, while many exemplary audio libraries for the browser can schedule events with sample-accurate precision using methods of the WAAPI such as `setValueAtTime` [5, 11], this requires some lookahead to be effective.

2.3 Browser Constraints on Per-sample Processing

While per-sample processing affords unique synthesis and temporal possibilities, the constraints of using the ScriptProcessor node, currently the only vehicle for per-sample processing in the browser, have been well documented [13]. Although implementations of the WAAPI provide block sizes as low as 256 samples, the ScriptProcessor node imposes an extra buffer of latency, resulting in a minimum latency of `512` sample between interaction events and hearing their effects in the output audio signal. This is more than $10ms$ of latency at a $44.1kHz$ sampling rate, although using an audio interface that provides higher sampling rates can mitigate this effect. For many applications a more significant problem is that the ScriptProcessorNode performs all signal processing in the JavaScript's main thread, which means that other events using this thread (such as network and user interaction events) have the potential to block audio processing and create interrupts. This is especially problematic for applications with rigorous graphical demands in addition to audio signal processing needs.

These concerns will potentially be removed with the introduction of the *AudioWorklet* node, an upcoming addition to the WAAPI whose specification has been standardized[2]. The AudioWorklet node will operate in its own thread and remove the latency penalties associated with the ScriptProcessor node. However, there are still efficiency gains from using the various C++ ugens included with the WAAPI instead of ugens created in a dynamic language like JavaScript, in addition to performance improvements from rendering the output of each ugen block by block instead of sample by sample. As devices become increasingly powerful developers must consider whether or not the performance penalties incurred by the ScriptProcessor and AudioWorklet nodes are too heavy a cost for the expanded synthesis and scheduling capabilities per-sample processing provides.

3. GENISH.JS

genish.js is a "low-level" collection of unit generators that can be combined to form audio graphs; these graphs are then compiled into optimized JavaScript functions outputting either a mono sample or a stereo pair. We describe the included unit generators as low-level because each is responsible for a single small task, for example, incrementing a number (`accum`) or determining the equality of two values (`eq`). For audio processing the compiled functions that Genish.js creates are typically used inside an instance of a ScriptProcessor node, but the functions can also be used to calculate the output of a wide variety of non-musical, physically informed systems where feedback is an important consideration. Genish.js is based on the Gen framework for Max/MSP [12], which provides a low-level DSP environment generating output one sample at a time; this runs alongside the block-based audio processing system that MSP provides. We drew inspiration from Gen for two reasons: first, to take advantage of a design that has been well-used and tested for over five years (and much longer if you consider that Gen was itself derived from MSP), and second, to potentially attract existing users of Gen to Genish.js.

The Genish.js website contains a variety of materials to help start using the library. First, a code playground provides a number of examples detailing how to write ugens, including using the jsdsp syntax discussed in Section 3.5. Many of the examples document signal processing techniques that are only possible in the browser with ugens that operate on a per-sample basis, like the design of dynamic IIR filters, FM feedback, and feedback delay networks. Second, an introductory long-form tutorial starts with the basics of how accumulators, phasors, and oscillators work and advances to creating a two-operator, enveloped, FM synthesis ugen. Finally, an API reference provides documentation for each individual ugen and examples of use. Examples are also provided showing how to use a Genish.js callback function within a ScriptProcessor node.

3.1 Compilation

The `accum` ugen, which increments a number every time it is called, is one of the most frequently ugens in Genish.js. An example using it to create a sawtooth oscillator[3], along with the corresponding compiled callback function, is shown in Listing 1.

Output functions are generated by calling the `createCallback`

the ScriptProcessorNode) see http://mmontag.github.io/dx7-synth-js/

[2]http://tinyurl.com/audio-worklet

[3]An `export` function enables ugens to be easily placed in the global namespace for experimentation; all code examples in this paper will assume use of the global namespace.

function and passing it a graph. If an array is passed the graph is assumed to be multi-channel, with each array member representing one channel of the graph; channels can share nodes freely. Once `createCallback` has generated a function it can be used without any arguments to produce output data (`input` ugens enable passing arguments to the compiled function). Note that in the Genish.js callbacks all unit generators share a single block of memory that is dereferenced at the top of the callback; shared memory usage provides a large performance boost to the library, discussed further in Section 3.3. The `[0,1]` indices in the memory block are reserved for writing the final output of the function.

Although complex audio graphs can compile to functions that are hundreds of lines of code in length, many other algorithms can be expressed quite succintly. For example, the generated output code for a biquad filter, assuming that coefficients are calculated ahead of time, is provided in the appendix; the compiled code is quite terse.

```
// the following is Genish.js end-user code...
const graph = accum(
  440 / samplerate,
  0,
  { min:-1, max:1 }
)

// ...and compiles into the following function:
function gen(  ){
  "use strict"
  var memory = gen.memory

  var accum123_value = memory[2]
  memory[2] += 0.009977324263038548
  if( memory[2] >= 1 ) memory[2] -= 2

  memory[0]  = accum123_value

  return memory[0]
}
```

Listing 1: A sawtooth oscillator in Genish.js

3.2 Unit generators

Table 3.2 shows most of the unit generators that Genish.js provides. Although many ugens are shared between Gen and Genish.js, Genish.js does contain a few notable differences. These typically account for differences between the patching environment that Gen is embedded in and the textual environment that Genish.js code is written in. Genish.js also contains both two-stage and four-stage envelopes that are not found in Gen; we considered these useful enough to include despite being at a higher level of abstraction than typical library ugens.

3.3 Memory management

Genish.js allocates a single heap containing all memory used in a generated output function; the size of the heap is user-adjustable but defaults to a `Float32Array` of length 4096. Using a single heap substantially improves lookup speed of phase, wavetables, delay lines, and other stored information; it is also part of what makes asm.js (see Section 3.4) efficient.

When each ugen is instantiated it requests a block of memory to be allocated. After performing allocation, a memory manager returns an index into the heap that the ugen should use for all subsequent read and write operations. Memory management is performed using a library named *MemoryHelper*[4], specifically created to support genish.js. Unlike other similar JavaScript libraries, MemoryHelper performs no checks to see if particular indices can be read from or written to by a given object; this tradeoff increases the speed of memory access but also creates the possibility that, if improperly written, a ugen might overwrite memory used by another ugen. In our experience the compilation stage of Genish.js has been effective in preventing such problems.

Table 1: genish.js ugens by type

Integrators	accum, counter
Buffers	data, peek, poke
Waveforms	phasor, cycle (sine oscillator), noise, train (impulse train)
Math / Numeric	add, mul, div, sub, sin, asin, cos, acos, tan, tanh, atan, pow, abs, sqrt, round, floor, ceil, sign, fold, wrap, clamp
Logic & Comparison	eq, max, min, gt, lt, gtp, ltp, gte, lte, not, bool, and, ifelse
Feedback & Filters	delay, history, dcblock, slide
Enveloping	ad, adsr, t60
Miscellaneous	switch, selector, gate, pan, in, param, bang

3.4 Using asm.js as a compilation target

Genish.js can also target asm.js, a subset of JavaScript which runs more efficiently in most browsers. The biggest benefits accompanying asm.js are type information provided to the compiler along with the use of a single memory heap. As Genish.js already used a single heap (see 3.3) the primary implementation task in getting Genish to compile to asm.js was adding the necessary typing information to each variable in the compiled code.

The audio programming langauges Faust and Csound also have the ability to compile to asm.js [4, 3]. Both are richer audio frameworks than Genish.js, and include a wide variety of tools for analysis in addition to synthesis. A debatable advantage of genish.js in comparison is that it is JavaScript end-to-end, making it potentially easier for developers who are not experienced with the Faust and Csound languages. Additionally, using asm.js with Faust requires either that DSP is compiled ahead of time into JavaScript files that are then statically included on a page, or that the entire Emscripten compiler (which is responsible for converting Faust to asm.js) is also embedded in a website if dynamic compilation of end-user code is required. Genish.js imposes neither of these requirements.

3.5 Operator Overloading with JSDSP

Although JavaScript is now efficient enough to execute low-level DSP functions at audio rate, the task of writing these functions can often be somehwat arduous. In particular, the lack of operator overloading in JavaScript makes DSP functions both more time consuming to write and more difficult to read. For example, consider the two functions created

[4]https://github.com/charlieroberts/MemoryHelper

using genish.js as shown in Listing 2. Both functions generate the same output graph consisting of tremolo applied to a sine oscillator. The first example uses arithmetic ugens while the second takes advantage of operator overloading.

```
// using arithmetic ugens
const createGraph = ( freq, gain ) => {
  const lfo = mul( cycle(4), 10 )
  return mul( cycle( add( freq, lfo ) ), gain )
}

// with operator overloading
const createGraph = ( freq, gain ) => {
  "use jsdsp"
  const lfo = cycle( 4 ) * 10
  return cycle( freq + lfo ) * gain
}
```

Listing 2: Comparing arithmetic ugens and operator overloading in genish.js

We argue that the second example, using operator overloading, is easier to both read and write. For more complex graphs of ugen nodes, in particular the differential equations used in most filters, avoiding complex nesting of mathematical operators brings benefits in terseness, expressiveness, and readability. In Genish.js we've provided many of the benefits of operator overloading as a plugin for Babel[5], a widely used JavaScript compiler and transpiler. This plugin, named *jsdsp*, can either be used as part of a build script (for JavaScript production tools such as gulp or grunt) or, in the case of in-browser code editors, added as a compilation stage that is run whenever code is dyanmically executed by users. In the Genish.js playground we've added a button that toggles use of jsdsp on or off, so that end-users can judge for themselves if the use of the jsdsp plugin is merited for their particular application. The `'use jsdsp'` directive can be placed at the top of any functional scope (similar to the `'use strict'` directive) or placed at the top of any block to enable the extra compilation stage.

4. IMPROVING GIBBERISH

The goal of Genish.js is to standardize low-level DSP operations in unit generators that are efficient, reusable, and testable. Previous versions of Gibberish lacked this type of low-level library to work with, leading to a library that was difficult to optimize as optimizations were often specific to a particular synthesis object, rather than shared across the library. For example, any optimization applied to the `accum` ugen will automatically improve every DSP algorithm involving phase in Genish.js, whereas in older versions Gibberish each synthesis ugen handled phase in its own idiosyncratic way.

While many affordances of Gibberish remain the same (per-sample processing, sample-accurate scheduling, audio-rate modulation of scheduling, single-sample feedback loops) in the newest version of Gibberish almost all DSP was written using Genish.js. Each synthesis object in Gibberish consists of a compiled Genish.js function along with methods for connecting these functions together and performing other miscellaneous initialization tasks. Gibberish then uses these individual callbacks to generate a master callback function that returns the output of processing the entire audio graph, one sample (or stereo pair) at a time. These samples are in then used to populate the audio buffers provided by a ScriptProcessor node.

Incorporating Genish.js led to a variety of new features in Gibberish that improve its overall sonic character and ease of use.

4.1 Filters in Gibberish

Two new "virtual analog" filters have been added to Gibberish, using zero-delay algorithms described by Pirkle [6]. The first filter models the 24dB per octave Moog ladder filter, while the second models the diode filter design found in the Roland TB-303, a well-known analog monosynth. Combined with the previous ladder filter alogrithm as well as biquad and state variable filters, there are now five different filters that users can apply to instruments (in addition to utility comb and allpass filters).

In prior versions of Gibberish, there were separate classes of synths for those that included a filter as opposed to those that only included an oscillator and an envelope. In the current version, filters can be enabled at any point in time and end-users can easily experiment with using different filter models. There are a number of possible design patterns that we could have use to enable this, including swapping nodes in the audio graph at run time or large branching control structures to accommodate all filter use cases. However, in keeping with the theme of code generation found throughout Gibberish and genish, we instead simply recompile the callback function for the synthesizer (generated by genish.js) whenever a filter is enabled or a different model is selected. A similar strategy is used in select other cases where providing the ability to change a property would result in extremely large branching structures. For example, oscillators can easily be switched between algorithmic approaches, wavetables, and anti-aliased versions using FM feedback. When such a change occurs, the oscillators compiled function is simply re-compiled.

4.2 Authoring Unit Generators

In addition to improving efficiency, the incorporation of Genish.js into Gibberish has also provided a standardized method of authoring unit generators. Gibberish has a `factory()` method that accepts a Genish.js graph and a list of public properties that are tied to inputs in the graph; this method returns a Gibberish unit generator that can easily be connected to a Gibberish audio graph. In Listing 3 we provide a complete example createing a mono gain filter in Gibberish.

```
MonoGain = function( input=0, gain=1 ) {
  const ugen = Object.create(
    Gibberish.prototypes.ugen
  )
  const __input = genish.in( "input" )
  const __gain  = genish.in( "gain"  )
  const graph   = genish.mul( __input, __gain )

  // factory method accepts ugen, a genish.js
  // graph, a name, and a property dictionary
  Gibberish.factory(
    ugen,
    graph,
    "monogain",
    { "input":input, "gain":gain }
  )

  return ugen
}
```

[5]http://babeljs.io

```
// synth -> monogain -> output
syn = Synth()
gain = MonoGain( syn, .5 ).connect()
```

Listing 3: Defining and using a Gibberish unit generator

The Gibberish playground contains a detailed tutorial on authoring unit generators using genish.js.

4.3 Miscellaneous Changes

Other improvements include a new chorus model based on the circuitry found in the ARP Solina string model (ported from a Csound opcode by Steven Yi), selectable waveforms and a feedback parameter for the two-operator FM synthesizer (which now features an optional filter as well), and a implementation of the Datorro plate reverb algorithm in addition to Freeverb. Many of these new features incur increased computational costs, bringing new importance to the performance improvements gained by using Genish.js.

5. EVALUTION

We briefly consider three criteria when evaluating these libraries. The first concerns the reliability of the libraries themselves. Genish.js features extensive unit tests and also includes performance benchmarks justifying various compiler decisions; these tests extend to its custom memory manager.

Another evaluation criteria is the ease of using these libraries. Although we have not conducted any formal user studies, we believe having a standard library of ugens to draw upon (genish.js) simplifies the process of authoring higher-level synthesis and effects objects in Gibberish. And in our personal experience, DSP routines authored using the operator overloading provided by the jsdsp Babel plugin are much more appealing to both read and write compared to explictly instantiating unit generators for basic mathematical and logical operations.

Efficiency is another important criteria; however, it is difficult to evaluate these libraries in comparison to other JavaScript audio/music programming systems as each has different affordances and constraints. For example, while genish.js affords synthesis techniques not possible in other libraries, it imposes costs through both its reliance on the ScriptProcessor node and its use of per-sample processing. The results of preliminary benchmarks comparing genish.js to the native WAAPI nodes in three common synthesis tasks are displayed in Table 2. Considering that the built-in WAAPI nodes are written in C++ instead of JavaScript and perform block-based processing of the graph instead of processing one sample at a time, we feel the benchmarks for genish.js are strong — and in some cases are even better than the benchmarks of the native WAAPI nodes. While these three preliminary synthesis benchmarks are not enough to make any type of definitive statements about performance, the results are promising. All tests were performed on a late 2016 Macbook Pro (2.6 GHz Core i7) running macOS 10.12.3[6].

Comparing prior versions of Gibberish created without using genish.js is a more important test for us, is the increase in improvement from prior versions of Gibberish to the most recent one that employs genish.js, and we are pleased to note a $\approx 3x$ performance improvement in many applications. This enables the use of more sophisticated filters and effects that would have been prohibitively expensive in prior versions of the library.

[6]http://tinyurl.com/genish-benchmarks

Table 2: Performance comparison (mean and standard deviation, $n = 100$) between genish.js and the WAAPI.

100 sine oscillators for 1 minute	
genish.js (Chrome)	$2385ms$, $\sigma = 76ms$
WAAPI (Chrome)	$2987ms, \sigma = 71ms$
genish.js (FF)	$3515ms$, $\sigma = 181ms$
WAAPI (FF)	$2680ms, \sigma = 171ms$
50 sine oscillators w/ vibrato for 1 minute	
genish.js (Chrome)	$2568ms, \sigma = 104ms$
WAAPI (Chrome)	$1670ms$, $\sigma = 102ms$
genish.js (FF)	$4171ms, \sigma = 218ms$
WAAPI (FF)	$5248ms$, $\sigma = 275ms$
50 saw oscillators w/ envelopes for 1 minute	
genish.js (Chrome)	$2804ms, \sigma = 104ms$
WAAPI (Chrome)	$2029ms$, $\sigma = 102ms$
genish.js (FF)	$11281ms$, $\sigma = 218ms$
WAAPI (FF)	$2955ms, \sigma = 275ms$

6. CONCLUSION

There are serious limitations with the current implementation of the ScriptProcessor node, and web audio developers need to carefully weigh the benefits of writing sample-processing routines in JavaScript against its drawbacks. However, the upcoming AudioWorklet specification should remove many of the problems associated with per-sample processing in the the browser, and we believe that performing processing off the main thread and removing the additional buffer of latency that the ScriptProcessor currently imposes should make writing DSP in JavaScript much more appealing. The AudioWorklet will make libraries using per-sample techniques increasingly relevant, and we hope Genish.js will fill this role for many developers. It comes with an extensive testing suite, documentation, and encouraging performance results in preliminary tests comparing it to the native WAAPI nodes.

Notably lacking from Genish.js and Gibberish are features for analysis, re-synthesis, and convolution. When nodes authored with Genish.js are used in conjunction with the native WAAPI nodes in a graph this is not an issue. But inside of Gibberish, where the entire graph is processed in a single ScriptProcessor node, these are significant constraints. We hope to integrate these features into Genish.js in the near future, and subsequently continue to expand the synthesis options available in Gibberish.

In addition to its incorporation in Gibberish, Genish.js is also used in *gibberwocky.midi* [8], a live-coding environment for the browser that uses Genish.js to create modulation signals transmitted as MIDI Control Change messages. We looked forward to adopting it in other future projects.

7. ACKNOWLEDGMENTS

We would like to thank Graham Wakefield for answering general questions regarding the Gen framework, and to ac-

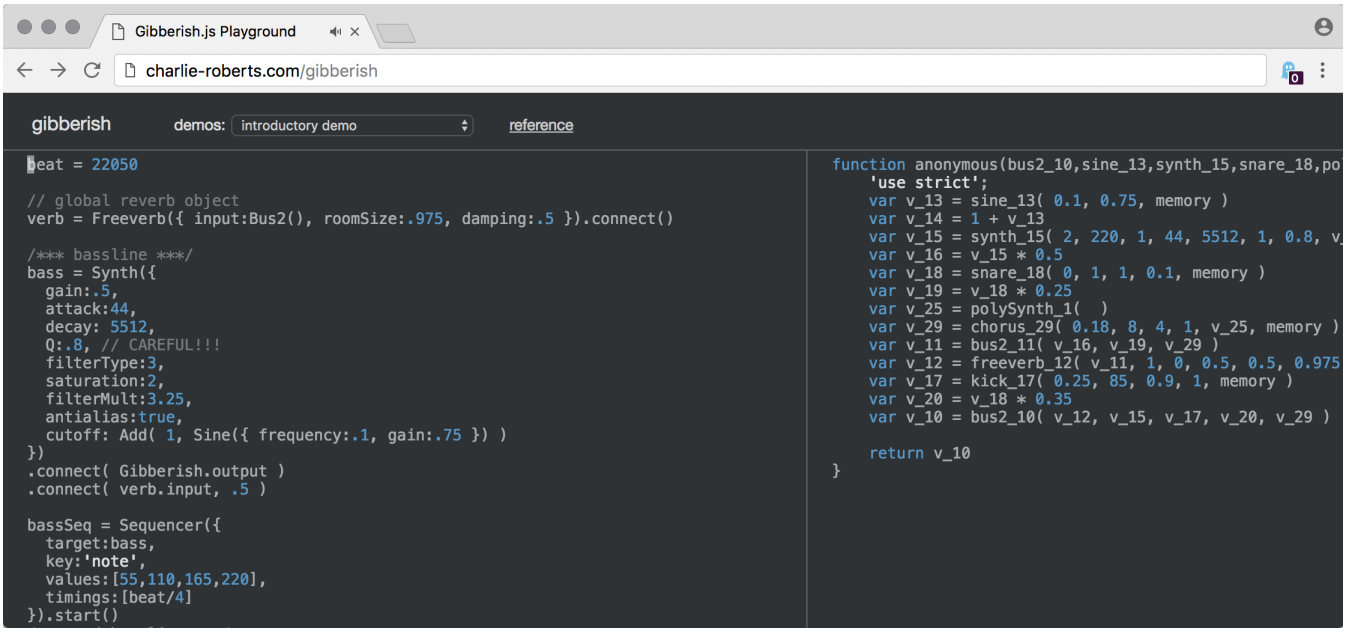

Figure 1: The Gibberish playground, with end-user code on the left and compilation output on the right.

knowledge Cycling '74 more broadly for the inspiration their product had on the research presented here. We would also like to thank Paul Adenot for his (most excellent!) suggestion to use a single block of memory in Genish.js callbacks.

8. REFERENCES

[1] C. Clark and A. R. Tindale. Flocking: A Framework for Declarative Music-Making on the Web. In *International Computer Music Conference*, 2014.

[2] K. L. Kim and W. S. Yeo. Griddy: a drawing based music composition system with multi-layered structure. In *Proceedings of the International Computer Music Conference*, 2014.

[3] V. Lazzarini, E. Costello, S. Yi, et al. Csound on the Web. In *Proceedings of the 2014 Linux Audio Conference*, pages 77–84. University of Bath, 2014.

[4] S. Letz, S. Denoux, Y. Orlarey, and D. Fober. Faust audio dsp language in the web. In *Proceedings of the Linux Audio Conference (LAC-15), Mainz, Germany*, 2015.

[5] Y. Mann. Interactive Music with Tone.js. In *Proceedings of the 1st annual Web Audio Conference*, 2015.

[6] W. Pirkle. Virtual analog (va) filter implementation and comparisons. 2013.

[7] C. Roberts and J. Kuchera-Morin. Gibber: Live Coding Audio In The Browser. *Proceedings of the International Computer Music Conference*, 2012.

[8] C. Roberts and G. Wakefield. gibberwocky: New Live-Coding Instruments for Musical Performance. In *Proceedings of the New interfaces for Musical Expression Conference*, 2017.

[9] C. Roberts, G. Wakefield, M. Wright, and J. Kuchera-Morin. Designing musical instruments for the browser. *Computer Music Journal*, 39(1):27–40, 2015.

[10] B. Taylor and J. Allison. BRAID: A web audio instrument builder with embedded code blocks. In *Proceedings of the 1st international Web Audio Conference*, 2015.

[11] T. Tsuchiya, J. Freeman, and L. W. Lerner. Data-Driven Live Coding with DataToMusic API. In *Proceedings of the 2nd annual Web Audio Conference*. Georgia Institute of Technology, 2016.

[12] G. Wakefield. *Real-Time Meta-Programming for Interactive Computational Arts*. PhD thesis, University of California Santa Barbara, 2012.

[13] L. Wyse and S. Subramanian. The Viability of the Web Browser as a Computer Music Platform. *Computer Music Journal*, 37(4):10–23, 2013.

APPENDIX

Below is a compiled Genish.js function for a the sample processing routine of a biquad filter, as described in Section 3.1:

```
function gen( input ){
  "use strict"
  var memory = gen.memory

  var mul7 = input * memory[3]
  var mul9 = memory[ 2 ]  * memory[5]
  var mul10 = memory[ 4 ]  * memory[6]
  var add11 = mul7 + mul9 + mul10
  var mul13 = memory[ 7 ]  * memory[9]
  var mul14 = memory[ 8 ]  * memory[10]
  var add15 = mul13 + mul14
  var sub16 = add11 - add15
  memory[0]  = sub16

  memory[ 2 ] = input
  memory[ 4 ] = memory[ 2 ]
  memory[ 8 ] = memory[ 7 ]
  memory[ 7 ] = sub16

  return memory[0]
}
```