The Web Browser As Synthesizer And Interface

Charles Roberts
Media Arts and Technology Program
University of California at Santa Barbara
charlie@charlie-roberts.com

Graham Wakefield
Graduate School of Culture Technology
KAIST
Daejon, Republic of Korea
grrrwaaa@kaist.ac.kr

Matthew Wright
Media Arts and Technology Program
University of California at Santa Barbara
matt@create.ucsb.edu

ABSTRACT

Our research examines the use and potential of native web technologies for musical expression. We introduce two JavaScript libraries towards this end: Gibberish.js, a heavily optimized audio DSP library, and Interface.js, a GUI toolkit that works with mouse, touch and motion events. Together these libraries provide a complete system for defining musical instruments that can be used in both desktop and mobile web browsers. Interface.js also enables control of remote synthesis applications via a server application that translates the socket protocol used by web interfaces into both MIDI and OSC messages.

Keywords

mobile devices, javascript, browser-based NIMEs, web audio, websockets

1. INTRODUCTION

Web technologies provide an incredible opportunity to present new musical interfaces to new audiences. Applications written in JavaScript and designed to run in the browser offer remarkable performance, mobile/desktop portability, and longevity due to standardization. As the web browser has matured, the tools available within it to create dynamic musical content have also progressed, and in the last two years realtime and low-level audio programming in the browser has become a reality. Given the browser's ubiquity on both desktop and mobile devices it is arguably the most widely distributed run-time in history, and is rapidly becoming a write once, run anywhere solution for musical interfaces.

Additionally, web apps can now incorporate accelerometers, multitouch screens, gyroscopes, and fully integrated sound synthesis APIs, making web technologies suddenly very attractive for NIMEs. Our research stems from the desire to explore the affordances offered by NIMEs that are web apps and the belief that such instruments should be open and cross-platform. Ideally, a web-based musical interface should be able to run on desktop machines, laptops, and mobile devices regardless of the browser or operating system it is presented in. We believe the libraries described here are a significant step towards realizing this goal.

Our research is especially sympathetic to NIME developers wanting to take advantage of web technologies without having to learn the quirks and eccentricities of JavaScript, HTML and CSS simultaneously. We designed the syntax of Gibberish.js and Interface.js so that they can be utilized, both independently or in conjunction, almost entirely using JavaScript alone, requiring a bare minimum of HTML and absolutely no CSS (see Section 5). The library downloads include template projects that enable programmers to safely ignore the HTML components and begin coding interfaces and signal processing in JavaScript immediately.

2. PROBLEMS AND POSSIBILITIES OF THE WEB AUDIO API

In 2011 Google introduced the Web Audio API[14] and incorporated supporting libraries into its Chrome browser. The Web Audio API is an ambitious document; instead of merely defining a basic infrastructure for processing audio callbacks, the API defines how audio graphs should be created and lists a wide variety of unit generators ("ugens") that should be included in browsers as standard. These ugens are native pre-compiled objects that can be assembled into graphs and controlled by JavaScript. For better or for worse, the Mozilla Foundation had previously released a competing API ("Audio Data")[9] that they integrated into the Firefox browser. This API also provided a well-defined mechanism for writing audio callbacks in JavaScript, however it provided no specifications for lower level, native, pre-compiled ugens to be standardly included in browsers. Fortunately, the Mozilla Foundation recently declared their intention to adopt the Web Audio API[1], suggesting that it will become the single standard for browser-based audio synthesis. In October of 2012, Apple rolled out iOS 6.0 to their phone and tablet devices and integrated the Web Audio API into mobile Safari; it seems likely to be adopted in other mobile platforms in the future, with promise of longevity through standardization. Our research supports both these APIs and automatically selects the correct one to use.

Although the Web Audio API comes with highly efficient ugens for common audio tasks such as convolution and FFT analysis, there are tradeoffs that accompany its usage. Most relate to the architectural decision to process blocks of at least 256 samples ($\approx$ 6 ms) instead of performing single-sample processing. This places two notable restrictions on audio programming performed using the Web Audio API: lack of sample-accurate timing and inability to create feedback networks with short delays, which are necessary to important signal-processing applications including filter design and physical modeling.

Fortunately, there is a workaround for these restrictions. The Web Audio API contains a specification for a JavaScript audio node that calculates output using JavaScript callbacks that can be defined at runtime. These guarantee sample-accurate timing and enable complex feedback networks, but at the expense of the efficiency that the native pre-compiled ugens provide. During our experiences developing *Gibber*[13], a live coding environment for the browser, we found that existing audio libraries (described in Section 6) built for this runtime JavaScript node were not efficient enough to realize the complex synthesis graphs we envisioned and thus began work on our own optimized library, *Gibberish.*

3. GIBBERISH: AN OPTIMIZED JAVASCRIPT AUDIO LIBRARY

The principal reason for JavaScript's excellent performance in the browser is the use of just-in-time (JIT) compilation: the virtual machine detects the most heavily used functions, path and type-specializations of the code as it runs, and replaces them with translations to native machine code. Although JIT compilation is making waves in the browser today, it can trace a long and diverse history to the earliest days of LISP[5]. Now JIT compilers can approach and occasionally even beat the performance of statically compiled C code, though getting the best performance from a JIT compiler may call for very different coding habits than for a static compiler or interpreter.

We began our research by looking at the performance bottlenecks associated with the JavaScript runtime and analyzing what could be done to ameliorate them. The most significant cost we found while working at audio rate in a dynamic runtime is the overhead of object lookup. In complex synthesis graphs, the simple task of resolving object addresses thousands of times per second (and potentially hundreds of thousands of times) levies a substantial cost. Gibberish minimizes this cost by ensuring that all data and functions used within the audio callback are defined and bound within the outer local scope (as "upvalues"), avoiding the need for expensive indexing into externally referenced objects[8].

3.1 Code Generation

Unfortunately, ensuring the locality of data and procedures for performance is not compatible with the flexibility to dynamically change the ugen graph, since adding or removing ugens implies changing the set of data and procedures that should be considered local to the audio callback. Our solution to this problem, inspired by [16], utilizes run-time code generation.

To optimize the audio callback the Gibberish code generation ("codegen") engine translates the user-defined ugen graph created with object-oriented syntax into a single flat audio callback where all routing and modulation is resolved ahead of execution. This process is basically one of string manipulation. Each ugen is responsible for generating a fragment of code that invokes both its callback function and the callbacks of all ugens that feed into it. Since all inputs to ugens are resolved recursively, requesting the master output bus to perform code generation will effectively flatten the entire audio graph into a linear series of code fragments invoking ugen callbacks. These code fragments are then concatenated to form a string representing the master audio callback which is then dynamically evaluated into a callable function using JavaScript's `eval()` function.

As a simple example of the codegen algorithm in action, consider the following high-level programming syntax to create a frequency-modulated sine wave fed into a reverb that in turn feeds the default output bus:

```
modulator = new Gibberish.Sine( 4, 50 ); // freq, amp
carrier = new Gibberish.Sine({ amp : .1 });
carrier.frequency = add( 440, modulator );
reverb = new Gibberish.Reverb({ input:carrier });
reverb.connect();
```

Traversing from the master output down through the graph, codegen will occur in the following order: `Bus > Reverb > Carrier > Add > Modulator`. The result is the following callback function, which will be called every sample:

```
function () {
  var v_16 = sine_5( 4, 50 );
  var v_15 = sine_8( 440 + v_16, 0.1 );
  var v_17 = reverb_11( v_15, 0.5, 0.55 );
  var v_4 = bus2_0( v_17, 1, 0 );
  return v_4;
}
```

The `sine_5`, `sine_8`, `reverb_11` and `bus2_0` variables are all upvalues defined immediately outside the scope of the master audio callback and thus resolved inexpensively. These variables refer not to the ugens but to their signal processing functions (aka audio callbacks). The simplistic style of the generated code generally leads to improved runtime performance. For example, passing values by simple function arguments rather than using higher-level object properties and instance variables is ten percent faster in Google Chrome 26, according to [3].

Note that parametric properties of the ugens are translated as numeric literals (constants) in the generated code. By storing properties as compile-time constants, rather than using dynamic state, expensive calls to continuously index them are entirely avoided. However the implication, which may appear unusual, is that code generation must be invoked whenever a ugen property changes, in addition to when ugens are added or removed. In the above example, if we changed the frequency of our modulator sine wave from four to three Hertz, code generation would be retriggered. To reduce the cost of frequent regeneration, Gibberish caches and re-uses code for all ugens that have not changed. Whenever a ugen property is changed, that ugen is placed inside of a "dirty" array. During codegen each ugen inside the dirty array regenerates the code fragment that invokes its callback function; the code fragments for the rest of the ugens remain unaltered. Even if the callback is regenerated dozens or hundreds of times a second it still winds up being fairly efficient as only ugens with property changes invoke their codegen method; all "clean" ugens simply provide the last fragment they generated to be concatenated into the master callback.

3.2 Gibberish Ugens

A variety of ugens come included with Gibberish, listed in Table 1. In addition to standard low-level oscillators such as sine waves and noise generators, Gibberish also provides a number of higher-level synthesis objects that combine oscillators with filters and envelopes. These high-level objects provide a quick way to make interesting sounds using Gibberish and their source code also serves as guidance for programmers interested in writing their own synth definitions.

It is fairly terse for users to add, redefine, or extend a Gibberish ugen at runtime. For example, here is the definition for the Noise ugen:

```
Gibberish.Noise = function() {
  this.name = 'noise';
  this.properties = { amp:1 };
  this.callback = function( amp ) {
    return ( Math.random() * 2 - 1  ) * amp;
  };
  this.init();
}
Gibberish.Noise.prototype = Gibberish.Ugen;
```

Table 1: Gibberish Ugens by Type

Oscillators	Sine, Triangle, Square, Saw, Bandlimited Saw, Bandlimited PWM / Square, Noise
Effects	Waveshapers, Delay, Decimator, Ring Modulation, Flanger, Vibrato, Chorus, Reverb, Granulator, Buffer Shuffler / Stutterer.
Filters	Biquad, State Variable, 24db Ladder, One Pole
Synths	Synth (oscillator + envelope), Synth2 (oscillator + filter + envelope), FM (2 operator), Monosynth (3 oscillator + envelope + filter)
Math	Add, Subtract, Multiply, Divide, Absolute Value, Square Root, Pow
Misc	Sampler (record & playback), Envelope Follower, Single Sample Delay, Attack/Decay Envelope, Line/Ramp envelope, ADSR Envelope, Karplus-Strong, Bus

The `init()` method is inherited from the Ugen prototype and sets up code generation routines for the ugen in addition to performing other initialization tasks. The *properties* dictionary in the ugen definition identifies properties that are used in the ugen's callback method signature, and specifies their default values. When code generation occurs, the order that individual properties are enumerated in this dictionary defines their order in the ugen's callback method signature.

3.3 (Single Sample) Feedback

Allowing single sample feedback is a benefit of processing on a per-sample basis instead of processing buffers. It can be achieved in Gibberish using the Single Sample Delay (SSD) ugen. This ugen samples and stores the output of an input ugen at the end of each callback execution and then makes that sample available for the next execution. Below is a simple feedback example with a filter feeding into itself:

```
pwm = new Gibberish.PWM();
ssd = new Gibberish.SingleSampleDelay();

filter = new Gibberish.Filter24({
  input: add( pwm, ssd ),
}).connect();
ssd.input = filter;
```

In the generated callback below note that there are two functions for the SSD ugen. One records an input sample while the other outputs the previously recorded sample.

```
function () {
  var v_7 = single_sample_delay_3();
  var v_5 = pwm_2(440, 0.15, 0.5);
  var v_6 = filter24_1(v_5 + v_7, 0.1, 3, true);
  var v_4 = bus2_0(v_6, 1, 0);
  single_sample_delay_8(v_6,1);
  return v_4;
}
```

3.4 Scheduling and Sequencing

Although the JavaScript runtime includes its own methods for scheduling events, there is no guarantee about the accuracy of the time used by this scheduler. Accordingly, the only way to ensure accurate timing is to perform event scheduling from within the audio sample loop. This in turn means that any JavaScript DSP library that performs automatic graph management and callback creation (as Gibberish.js does) must also provide a mechanism for event timing that is synchronized to the audio sample loop. Gibberish.js contains a `Sequencer` object that can be used to sequence changes to property values, calls to methods, or execution of named or anonymous functions at user-defined rates with sample-accurate timing.

The syntax for a sequencer uses a simple target/key mechanism. A sequencer's *target* specifies the ugen (or any other JavaScript object—scheduling is not limited to controlling ugens) that the sequencer will control. The *key* specifies the name of a property that should be set or a method that should be called by the sequencer. The *values* array defines the values that are passed to object methods or assigned to object properties, while the *durations* array controls how often the sequencer fires off events. By default time is specified in samples, but convenience methods are provided for converting milliseconds, seconds, and beats to samples. Thus, the following sequences a `FMSynth` looping through three pitches and alternating between half a second and a quarter second duration:

```
a = new Gibberish.FMSynth().connect();

b = new Gibberish.Sequencer({
  target: a, key: 'note',
  values: [ 440, 880, 660 ],
  durations: [ seconds(.5), seconds(.25) ],
}).start();
```

Musicians may also want to attach more than one property change or method call to each firing of a sequencer object, for example, changing an oscillator's amplitude whenever its frequency changes. To accommodate this the Sequencer object can accept a *keysAndValues* dictionary that can feature multiple key/value pairs to be sequenced.

```
a = new Gibberish.Sine().connect();

b = new Gibberish.Sequencer({
  target: a,
  durations: [ beats(2), beats(1), beats(1) ],
  keysAndValues:{
    frequency: [ 440, 880, 660 ],
    amp: [ .5, .25 ],
}).start();
```

This keysAndValues syntax has similarities to SuperCollider's *Pbind* object[11], which also enables the sequencing of multiple parameters using the same timing.

When a values array contains a function, the Sequencer will simply invoke it (with no arguments) to determine the value it should use; this enables the sequencer to schedule arbitrary behavior such as randomly picking elements from an array. Functions within a sequencer's `values` array will still be called even if the Sequencer has no specified target, supporting sample-accurate sequencing of any arbitrary JavaScript function (for example, to synchronize state changes of multiple ugens).

4. INTERFACE.JS

Interface.js provides a solution for quickly creating GUIs using JavaScript that work equally well with both mouse and touch interaction paradigms. The output of these GUI widgets can control other JavaScript objects (such as Gibberish ugens) and/or be converted to MIDI or OSC[18] messages via a small server program that is included in the Interface.js download. This means Interface.js can be used both to control programs running in the browser and also to control external programs such as digital audio workstations.

Interface.js includes a useful array of visual widgets for audiovisual control, including `Button`, `MultiButton`, `Slider`, `MultiSlider`, `RangeSlider`, `Knob`, `XY` (multitouch, with physics), `Crossfader`, `Menu`, `Label`, and `TextField`.

Interface.js also provides unified access to accelerometer readings and the orientation of the device. These motion-based sensors work on most mobile devices as well as the desktop version of Google Chrome.

4.1 Event Handling in Interface.js

Interface.js binds widgets to object functions and values using a target / key syntax similar to the Gibberish sequencer (see Section 3.4). A widget's *target* specifies an object and the *key* specifies a property or method to be manipulated. What happens when a widget value changes depends on what the key refers to: object properties are *set* to the new widget value, while object methods are *invoked* with the widget value as the first argument. Widgets providing multiple dimensions of control (e.g., XY, accelerometer...) are configured with arrays of targets and keys with one target and one key for each dimension of control.

Users can also register custom event handlers to create more complex interactions than direct mappings between widgets and property values. Although most browsers only deal with either touch or mouse events, Interface.js provides another type of event handler, *touchmouse events*, that respond to both forms of input. The specific events are:

- `ontouchmousedown` - when a mouse press begins or a finger first touches the screen
- `ontouchmousemove` - when a mouse or finger moves after a start event
- `ontouchmouseup` - when a mouse press ends or a finger is removed from the screen

There is also a fourth event handler, `onvaluechange`, that is called every time the value of a widget changes, regardless of whether the widget changes by touch, the mouse, or by motion. Procedurally changing a widget's value (perhaps due to the output of another widget) will also trigger this event. Although the touchmouse and onvaluechange event handlers provide a unified interface for dealing with all events, users are free to register for dedicated mouse or touch events if they want to change how interactivity functions across different modalities.

4.2 Interface Layout And Appearance

The main container unit of Interface.js is the *Panel* widget. Panel wraps an instance of the HTML *canvas* element, a 2D drawing surface with hardware graphics acceleration support in all current desktop browsers and most current mobile browsers. Panel's constructor allows users to specify a HTML element for the canvas tag to be placed inside, allowing multiple panels to be placed at different points in a HTML page. If no container HTML element is provided, Interface.js will automatically create a canvas element that fills the entire window and attach it to the HTML page; this minimizes the HTML needed to create a GUI.

Widget sizes can be integer pixel values or float values giving the size relative to the widget's enclosing Panel. For example, consider placing a Slider inside a 500 × 500 pixel Panel. If the Slider's `x` value is .5, then its leftmost edge will be inset 250 pixels at the horizontal center of the panel. If the Slider's width is also .5, it will extend 250 pixels from the horizontal center to the rightmost edge of the Panel. Using relative layouts enables users to define interfaces without having to worry about varying window sizes or the screen sizes of different devices. An exception to this would be creating an interface for a large screen with a large number of widgets and then trying to view the same interface on a smartphone; in this case the widgets, although positioned and sized correctly, would be too small to use for accurate interaction.

In lieu of using CSS to specify colors, Interface.js instead allows them to be defined programatically using JavaScript. By default, a widget uses the background, fill and stroke properties assigned its container Panel object. Thus, by default, all widgets in the same Panel use the same set of colors. Changing the color properties of a Panel object immediately changes the colors of its child widgets. If color properties for an individual widget are explicitly set their values will override the Panel's default values.

4.3 Preset Management

Interface.js allows programmers to easily store the values of widgets in an interface for instant recall. A call to the method `Interface.Preset.save('preset name')` will serialize the values of all widgets in the current interface and convert the resulting object to a string that can be stored using the HTML `localStorage` object.

`Interface.Preset.load('preset name')` loads a preset. When a preset is loaded every widget sends out its target / key message (if so defined) and executes its `onvaluechange` event handler if the new value differs from the widget's current value.

4.4 Networked Control

As an alternative to controlling Web Audio synthesis graphs or other JavaScript objects, Interface.js can also transmit output over a network via the WebSocket API in order to control remote applications. The WebSocket API sends messages over TCP but features a handshake mechanism relying on the HTTP protocol. Since most musical applications do not understand the WebSocket API, it becomes necessary to translate messages received from a WebSocket to a more appropriate musical messaging protocol, such as OSC or MIDI.

Although there have been previous efforts creating servers that translate WebSocket data into OSC or MIDI messages, we believe our solution is uniquely efficient. Instead of simply creating a server that waits for incoming WebSocket messages and then forwards them using OSC or MIDI, we have included a HTTP server that serves interface files to client devices. After launching our server application, any user directing a client browser to the server's IP address (on port 8080) will receive an HTML page listing available interfaces stored in the server's default storage directory. When a user selects an interface, it is downloaded to their browser and a WebSocket connection is automatically created linking their client device to the server; in other words, the computer that serves Interface.js pages is automatically also the recipient of the WebSocket messages generated by such interfaces. WebSocket messages sent from the interface are translated into OSC or MIDI messages based on the contents of individual interface files.

A distinct advantage of this system is that programmers do not need to think about establishing socket connections when developing their interfaces; there are no hardcoded IP addresses and no auto-discovery protocols needed. The IP address and port is entered once when first accessing the server; after that the location can be bookmarked for easy access. Individual interfaces can also be bookmarked; in this case the WebSocket connection is established as soon as the interface bookmark is selected. On mobile devices these bookmarks can be represented by icons on user's home screens; simply tapping such icons fetches the interface and immediately opens the appropriate connection.

In order to indicate that a widget's output should be sent over the network, a target of Interface.OSC or Interface.MIDI is assigned. For widgets that target Interface.OSC, the key property then represents the address the

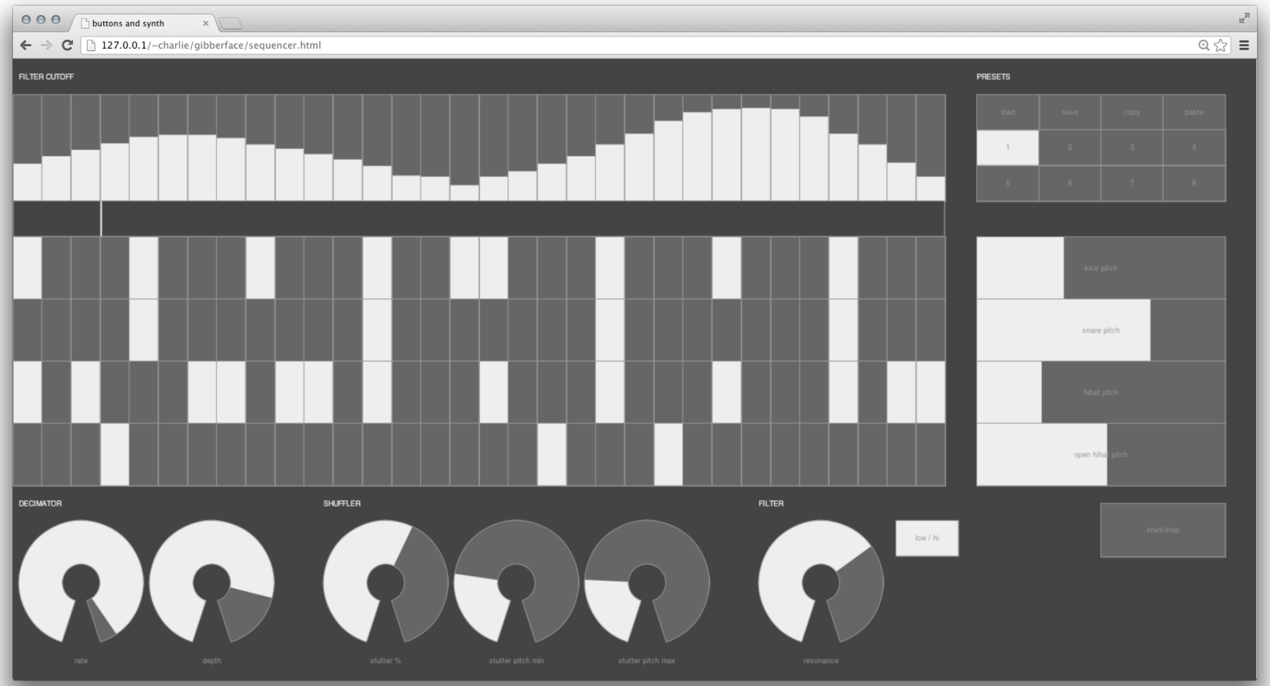

Figure 1: A drum sequencer with four tracks of 32nd notes and controls for various effects

message will be sent to. For widgets targeting Interface.MIDI, the key is an array containing the message type, channel, and number (if it is a three-byte MIDI message) for the output MIDI message. Multiple devices (and thus users) can load interfaces from the server and send messages for translation and forwarding simultaneously.

5. INTEGRATING INTERFACE.JS AND GIBBERISH.JS

There a few simple steps in order to use Interface.js and Gibberish.js together to make a musical interface. Starting with an HTML document containing the bare minimum necessary tags (`<html>`, `<head>` & `<body>`), add two `<script>` tags to import the Gibberish and Interface Javascript files. A third `<script>` tag inside the body element contains all user code to create the interface and/or audio graph. The following complete sample interface file creates a sine tone controlled by two sliders:

```
<html>
<head>
  <script src="interface.js"></script>
  <script src="gibberish_2.0.min.js"></script>
</head>

<body>
  <script>
  Gibberish.init();
  sine  = new Gibberish.Sine().connect();
  var panel = new Interface.Panel();

  sliderFrequency = new Interface.Slider({
    target:sine, key:'frequency',
    min:150, max:1000,
    label:'freq', bounds:[ 0,0,.3,1 ],
  });

  sliderAmp = new Interface.Slider({
    target:sine, key:'amp',
    label:'amp', bounds:[ .3,0,.3,1 ],
  });

  panel.add( sliderFrequency, sliderAmp );
  </script>
</body>
</html>
```

6. RELATED WORK

There are a growing number of options for writing synthesis algorithms in JavaScript in the browser. Audiolib.js[10] was one of the first significant JavaScript audio libraries written and still provides an excellent array of synthesis options. It was our original choice for Gibber; we abandoned it only after discovering that code generation often leads to more efficient performance. Audiolib.js performs no graph management leaving programmers to implement their own audio graphs.

Other libraries take vastly different approaches in terms of the APIs they offer programmers. For example, Flocking[7] enables users to define new ugens declaratively using JavaScript Object Notation (JSON), while Timbre.js[4] is a very impressive library that enables users to compose ugens in a functional syntax inspired by jQuery, a JavaScript library for HTML manipulation. In addition to its extensive use of code generation techniques, Gibberish.js differentiates itself from these other libraries by containing several more complex pre-composed synthesis ugens. For example, Gibberish offers a polyphonic, enveloped two-op FM synthesis ugen; to our knowledge no other JavaScript libraries offer FM synthesis engines with easy control over carrier to modulation ratios and index properties. Another example of a complex pre-composed ugen is the Gibberish Monosynth, a three-oscillator bandlimited synthesizer with an envelope, 24db resonant filter, and independent tuning and waveshape controls for each oscillator. By including complex ugens we allow programmers to begin creating music with rich sound sources immediately.

Another solution for web based synthesis is JSyn[6], a Java synthesis library originating over a decade ago. Unfortunately many browsers do not support Java by default, and many do not support it at all (including Safari on iOS).

Although there are many other HTML / JavaScript interface libraries, very few understand both touch and mouse modalities and almost none are catered towards the needs of musicians and live performers. A extension to jQuery named Kontrol[15] provides well-engineered virtual knobs and slider banks that respond to both touch and mouse events. However, there is no support for control areas tracking multiple touches across a single rectangle; in our opinion this is one of the most useful widgets for performance on touchscreen devices. The Interface.js XY widget tracks up to 11 touches simultaneously and also features a built-in physics engine, inspired by the JazzMutant Lemur[2]. In addition to providing only three widgets, the Kontrol library also requires knowledge of HTML and of the jQuery library. In general the library is targeted towards existing web developers while Interface.js attempts to shield programmers from HTML and CSS as much as possible.

The massMobile project[17] allows audience participation in performances through web interfaces on mobile devices that output to a database that can be read by Max/MSP externals. It is interesting to note that this project started with dedicated client applications for iOS and Android before switching to use web technologies in order to become usable on a wider variety of devices. A notable precursor to the massMobile project was the composition Telemusic #1, by Randall Bradley, Steve Young and John Young[19]. In this piece, users accessed a website with an embedded Flash animation that communicated with a Java applet running in the same page; this Java applet then forwarded information to a remote instance of Max/MSP.

7. CONCLUSIONS AND FUTURE WORK

Both Gibberish.js and Interface.js are open-source and available for download on GitHub.[1] A separate code repository, *Gibberface*[2], contains examples integrating both libraries, ranging from the simple (as in Section 5 above) to more complex interfaces and audio graphs such as the drum sequencer shown in Figure 1.

Part of the inspiration for Interface.js came from our work on the mobile application *Control*[12]. Control allows users to create interfaces for controlling remote applications with web technologies but also provides access to device features that are not exposed in browsers, such as the ability to send and receive OSC and MIDI. Advances in JavaScript APIs have exposed more and more functionality to mobile browsers' JavaScript runtime; gyro and compass sensors are two examples of sensors that were not accessible in the browser two years ago that now have established APIs. There are still a number of sensors commonly available on mobile devices that are inaccessible to JavaScript or impractical to process efficiently. As one example, detecting the surface diameter of individual touches can be used to create a crude yet surprisingly musical "aftertouch" signal that is not possible to read using browser APIs alone. As more APIs providing access to sensors become available in mobile browsers we will update Interface.js to take advantage of them.

In Gibberish there is a particular need for more analysis ugens beyond the simple envelope follower that currently exists. Gibberish has already been integrated into the live coding environment Gibber and numerous group and solo performances have been conducted using it as the synthesis engine. The first author has recently performed in concert using an instrument created with Gibberish.js and Interface.js on a tablet device; a modified version of this interface, which uses multitouch tracking to control bandlimited pulsewidth modulation oscillators, is included in the Gibberface repository.

[1]`https://github.com/charlieroberts/Gibberish` and `https://github.com/charlieroberts/interface.js`
[2]`https://github.com/charlieroberts/gibberface`

8. REFERENCES

[1] Web Audio API - MozillaWiki . `https://wiki.mozilla.org/Web_Audio_API`.

[2] JazzMutant Lemur. `http://www.jazzmutant.com/lemur_overview.php`.

[3] Parameter lookup vs. instance variables - jsPerf. `http://jsperf.com/parameter-lookup-vs-instance-variables`.

[4] T("timbre.js"). `http://mohayonao.github.com/timbre.js/`.

[5] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[6] P. Burk. JSyn–a real-time synthesis API for Java. In *Proceedings of the 1998 International Computer Music Conference*, pages 252–255. International Computer Music Association San Francisco, 1998.

[7] C. Clark. colinbdclark/Flocking - Github. `http://github.com/colinbdclark/Flocking`.

[8] Z. Herczeg, G. Lóki, T. Szirbucz, and Á. Kiss. Guidelines for JavaScript Programs: Are They Still Necessary? In *Proceedings of the 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*. Tampere University of Technology, 2009.

[9] D. Humphrey, C. Brook, A. MacDonald, Y. Delendik, R. Marxer, and C. Cliffe. Audio Data API - MozillaWiki. `https://wiki.mozilla.org/Audio_Data_API`.

[10] J. Kalliokoski. audiolib.js. `http://audiolibjs.org`.

[11] R. Kuivila. Events and patterns. In *The SuperCollider Book*, chapter 6, pages 179–205. MIT Press, 2011.

[12] C. Roberts. Control: Software for End-User Interface Programming and Interactive Performance. *Proceedings of the International Computer Music Conference*, 2011.

[13] C. Roberts and J. Kuchera-Morin. Gibber: Live coding audio in the browser. *Proceedings of the International Computer Music Conference*, 2011.

[14] C. Rogers. Web audio API. `http://www.w3.org/TR/webaudio/`.

[15] A. Terrien. jQuery Kontrol demo. `http://anthonyterrien.com/kontrol/`.

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

[17] N. Weitzner, J. Freeman, S. Garrett, and Y. Chen. massMobile–an audience participation framework. *Proceedings of the New Interfaces For Musical Expression Conference*, 2012.

[18] M. Wright. Open Sound Control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005.

[19] J. Young. Using the web for live interactive music. In *Proceedings of the 2001 International Computer Music Conference*, pages 302–305. Citeseer, 2001.