Designing Musical Instruments for the Browser

Charles Roberts,* Graham Wakefield,† Matthew Wright,* and JoAnn Kuchera-Morin*

*AlloSphere Research Group,
California NanoSystems Institute
University of California, Santa Barbara
Elings Hall, MC 6105
Santa Barbara, California 93106, USA
charlie@charlie-roberts.com,
{matt, jkm}@create.ucsb.edu
†York University
Goldfarb Centre for Fine Arts
4700 Keele St, Unit 303C
North York, Ontario M3J 1P3, Canada
grrrwaaa@gmail.com

Abstract: Native Web technologies provide great potential for musical expression. We introduce two JavaScript libraries towards this end: Gibberish.js, providing heavily optimized audio DSP, and Interface.js, a GUI toolkit that works with mouse, touch, and motion events. Together they 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.

We have incorporated these libraries into the creative coding environment Gibber, where we provide mapping abstractions that enable users to create digital musical instruments in as little as a single line of code. They can then be published to a central database, enabling new instruments to be created, distributed, and run entirely in the browser.

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

Browser-based digital musical instruments (DMIs) can incorporate accelerometers, multitouch screens, gyroscopes, and fully integrated application programming interfaces (APIs) for sound synthesis, making Web technologies particularly attractive for instrument designers. Our research explores the affordances of DMIs that run in Web browsers, with the belief that such instruments should be open and cross-platform. The libraries described here constitute a significant step toward Web-based musical interfaces that run on desktop machines, laptops, and mobile devices, regardless of the browser or operating system in which they are presented.

Although modern Web development typically requires knowledge of multiple programming and markup languages (at a minimum JavaScript, Hypertext Markup Language [HTML], and Cascading Style Sheets [CSS]), our research empowers instrument designers to take advantage of Web browsers without having to learn all their various quirks and eccentricities. We designed the syntax of Gibberish.js and Interface.js so that they can be used, both independently or in tandem, almost entirely via JavaScript alone, requiring a bare minimum of HTML and absolutely no CSS. The library downloads include template projects containing boilerplate HTML that enables programmers to quickly begin creating interfaces and audio graphs in JavaScript.

Continuing in this vein, we have integrated our research into the browser-based, creative coding environment Gibber (Roberts and Kuchera-Morin 2012; Roberts et al. 2014a) which removes the need to use any HTML at all. Gibber's multirate mapping abstractions enable DMIs to be tersely defined; a single line of JavaScript can create a simple instrument with an audio graph, an interface,

Computer Music Journal, 39:1, pp. 27–40, Spring 2015
doi:10.1162/COMJ_a_00283

and mappings between the interactive and audio elements. Instruments designed in Gibber can be published to its central database, distributed, and subsequently accessed via URL.

Problems and Possibilities of the Web Audio API

In 2011 Google introduced the Web Audio API (available online at www.w3.org/TR/webaudio) 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, it defines how to create audio graphs and includes a wide variety of standard *ugens* (unit generators). These ugens are native precompiled objects that can be assembled into graphs and controlled by JavaScript. The API has since been adopted by Firefox and Safari in both their desktop and mobile incarnations; as of this writing, support is in development for Internet Explorer (see status.modern.ie/webaudioapi).

Although the Web Audio API includes highly efficient ugens for common audio tasks such as convolution and fast Fourier-transform (FFT) analysis, there are tradeoffs that accompany their use. Many relate to the architectural decision to process blocks of at least 256 samples (circa 6 msec) instead of allowing single-sample processing. This places two notable restrictions on audio programming: lack of sample-accurate scheduling for manipulating audio graphs (particularly node insertion or removal) and an inability to create feedback networks with short delays, which are necessary to important signal-processing applications, including filter design and physical modeling. Although these limitations are found in many other audio programming environments, they are exacerbated in the browser by an inability to add native extensions that provide such capabilities. Other environments enable the authoring of custom extensions that use feedback. These are usually written in C++, although Max/MSP also provides Gen (Wakefield 2012).

Fortunately, the designers of the Web Audio API took these concerns into account and provided a solution, which they continue to refine: namely, the ScriptProcessor node that calculates output using JavaScript callbacks defined at run time. These afford sample-accurate scheduling of graph manipulation and the creation of complex feedback networks, but at the expense of the efficiency that the native precompiled ugens provide. Efficiency issues with the ScriptProcessor node are compounded by the fact that their processing takes place in the main thread, which can be subject to blocking due to network traffic or user interaction (Wyse and Subramanian 2013; Lazzarini et al. 2014). The ScriptProcessor node also introduces latency equal to the duration of one callback. For example, if the main audio graph is being processed in blocks of 256 samples, the use of a ScriptProcessor node (or nodes) will introduce an additional 256 samples of latency.

Despite these limitations—and in contrast to researchers who feel they limit the potential of the ScriptProcessor node for real-time musical applications (Wyse and Subramanian 2013)—we have successfully employed the ScriptProcessor node in many performances. Using a 2010 Apple MacBook Pro running at 2.8 GHz, the first author has given audiovisual live-coding performances consisting of music programming and real-time graphics. These have used a full-screen fragment shader implemented in OpenGL Shading Language without experiencing glitches in audio playback. Considering that these performances required editing programs in two code editors composited on top of a running OpenGL scene, dynamically recompiling shaders, and performing all audio signal processing in the main thread, we feel this is a strong indicator that the ScriptProcessor node in its current state is a practicable approach for many real-time audiovisual applications, particularly when they are run on moderately fast desktops or laptops and when reasonable steps are taken to avoid extra processing on the main thread. For example, users should avoid resizing browser windows during performances, as the necessary redraws incur a high amount of overhead that can easily block the main thread.

In the near future, ScriptProcessor nodes will receive a significant upgrade. The Audio Group of the World-Wide Web Consortium is developing a specification called Audio Workers that will run ScriptProcessor nodes in their own threads, and also

remove the additional latency they currently impose (Wilson 2014). In effect, ScriptProcessor nodes will become equal citizens with the other native audio nodes included in the Web Audio API. This makes research on optimizing JavaScript audio (outlined in the next section) particularly important, as more developers will be motivated to take advantage of the ScriptProcessor node when Audio Workers become available.

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, paths, 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 today's browsers, it can trace a long and diverse history to to the beginnings of Lisp in the late 1950s (Aycock 2003). Now JIT compilers can approach, and occasionally even exceed, the performance of statically compiled C code, though getting the best performance from a JIT compiler may call for coding habits quite different from those used for a static compiler or interpreter.

We began our research by looking at the performance bottlenecks associated with the JavaScript runtime environment (JSRE) and analyzing what could be done to mitigate them. The most significant cost we found while working at audio rate in a dynamic run-time environment is the overhead of object lookup. In complex synthesis graphs, the simple task of resolving object addresses thousands of times per second (potentially many hundreds of thousands of times with per-sample processing) levies a substantial cost. During our experiences developing Gibber, we found that existing audio libraries built for the ScriptProcessor node were not efficient enough to realize the complex synthesis graphs we envisioned, and thus we began work on our own optimized library, Gibberish.

Gibberish minimizes the cost of resolving object addresses by ensuring that data and functions used are inlined to the master audio callback and available within its local scope, avoiding the need for expensive indexing into externally referenced objects (Herczeg et al. 2009).

Code Generation

Unfortunately, ensuring the locality of data and procedures for performance is not compatible with the flexibility to dynamically change the ugen graph, because 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 the second author's doctoral work (Wakefield 2012), uses run-time code generation.

To optimize the audio callback, the Gibberish *codegen* (code generation) 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 prior to 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. Because 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 into a string representing the master audio callback. This string, along with a list of all arguments the master callback accepts, is used to create an invokable function via JavaScript's `Function` constructor.

As a simple example of the codegen algorithm in action, consider the high-level syntax used in Figure 1 to create a low-frequency modulated sine wave feeding a reverb that, in turn, feeds the default output bus in Gibberish. Traversing from the master output down through the graph, the code generation will occur in the following order: `Master Output Bus` → `Reverb` → `Carrier` → `Add` → `Modulator`. The result is the callback function displayed in Figure 2, which will be called every sample.

The variables `sine_5`, `sine_8`, `reverb_11`, and `bus2_0` refer not to the ugens but to their

Figure 1. End-user code to create a modulated sine oscillator feeding a reverb ugen.

```
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(); // connect to master output bus
```

Figure 1

Figure 2. Callback function generated from code in Figure 1, with comments and spacing added for readability.

```
function ( input, sine_5, sine_8, reverb_11, bus2_0 ) {
  var v_16 = sine_5( 4, 50 );
  var v_15 = sine_8( 440 + v_16, 0.1 );
  // arguments for reverb are input, roomSize, damping
  var v_17 = reverb_11( v_15, 0.5, 0.55 );
  // arguments for bus are input, amplitude, and panning
  var v_4 = bus2_0( v_17, 1, 0 );
  return v_4;
}
```

Figure 2

signal-processing functions (also known as audio callbacks). These individual ugen callbacks are passed as arguments to the master callback function. The simplistic style of the generated code generally leads to improved JSRE performance. For example, in unit generators with modulated inputs, the modulation graph and the resulting object references have to be traversed every sample. Flattening the graph ahead of time, by using code generation, removes this performance penalty.

Note that parametric properties of the ugens are translated as numeric literals (i.e., constants) in the generated code. By storing properties as compile-time constants, rather than using dynamic state, expensive calls to continuously index the properties are entirely avoided. But 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 given example, if we programmatically changed the frequency of our modulator sine wave from 4 Hz to 3 Hz, code generation would be retriggered. To reduce the cost of frequent regeneration, Gibberish caches and reuses 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.

We explored the possibility of using codegen to include all signal-processing procedures directly in the main callback. We did this by passing variables containing state to the main callback instead of passing callback functions to be invoked. Here the main callback becomes one giant function (potentially thousands of lines of code in length) instead of a shorter set of subroutine invocations. Our experiments with this yielded significant improvements in Chrome (better than twofold performance gains in version 38, 64-bit, under Mac OS X), but poor results in desktop versions of Safari and Firefox under the same operating system. We believe the JIT optimizer is unable to process the callback function after the function exceeds a certain length in these browsers. The results are improved in a beta version of Safari (r172617) we recently tested on Mac OS X, which adds a low-level virtual machine compilation stage to its JavaScript optimization arsenal. Callback functions that include all processing, however,

incur more than twice the CPU overhead of similar callbacks run in a contemporaneous version of Chrome (40.0.2197.2) running on the same operating system.

Gibberish Ugens

Gibberish includes a variety of ugens, listed in Table 1. In addition to standard low-level oscillators, there are also 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 models for programmers interested in writing their own ugen definitions.

Single-Sample Feedback

Enabling single-sample feedback is a benefit of processing on a per-sample basis instead of on 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. Figure 3 shows a simple feedback example with a filter feeding into itself.

In the generated callback shown in Figure 4, note that there are two functions for the SSD ugen. One (`ssd_8`) records an input sample, and the other (`ssd_3`) plays the last sample recorded.

Figure 3. End-user code to create single-sample feedback into a filter.

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

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

ssd.input = filter;
```

Figure 3

Scheduling and Sequencing

Although the JSRE includes its own methods for scheduling events, there is no guarantee about the accuracy of the time used by this scheduler. Chris Wilson, one of the primary architects of the Web Audio API, suggested one approach to ensuring accurate scheduling via temporally overlapping calls to a scheduler that runs in the JSRE's main thread (Wilson 2013). The overlap helps account for jitter that could otherwise occur due to other processing on the main thread. This method creates tradeoffs between temporal accuracy, efficiency, and the ability to change characteristics of scheduling (such as tempo) at will. Its primary advantage is resilience to various browser events that incur processing on the main thread (such as window resizing, network message processing, and garbage collection) and that could interfere with scheduling.

In contrast, per-sample processing in the ScriptProcessor node provides the potential for sample-accurate scheduling performed directly in its callback. In Gibberish the master clock is controlled by

Table 1. Gibberish Ugens by Type

Oscillators	Sine, Triangle, Square, Saw, Band-limited Saw, Band-limited PWM/Square, Noise, Wavetable
Drums	Kick, Snare, Tom, Clave, Hi-Hat, Conga, Cowbell
Effects	Waveshapers, Delay, Decimator, Ring Modulator, Flanger, Vibrato, Chorus, Tremolo, Reverb, Granulator, Buffer Shuffler/Stutterer
Filters	Biquad, State Variable, 24-dB Ladder, One-Pole
Synths	Synth (oscillator + envelope), Synth2 (oscillator + filter + envelope), FM (2-operator), Monosynth (3 oscillators + 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

Figure 4. Callback function for single-sample feedback generated from code in Figure 3. Comments and spacing added for readability.

```
function ( input, ssd_3, ssd_8, pwm_2, filter24_1, bus2_0 ) {
  var v_7 = ssd_3();
  // pwm arguments are frequency, amp, pulsewidth
  var v_5 = pwm_2( 440, 0.15, 0.5 );
  // filter arguments are input, cutoff, resonance, lowpass
  var v_6 = filter24_1( v_5 + v_7, 0.1, 3, true );
  // bus arguments are input, amp, panning
  var v_4 = bus2_0( v_6, 1, 0 );
  // single-sample-delay arguments are input, multiplier
  ssd_8( v_6,1 );
  return v_4;
}
```

an audio-rate signal. As a result, changes to tempo do not affect scheduling, because they are calculated on a per-sample basis. In addition to controlling the master clock via an audio-rate signal (which is freely definable by users), Gibberish provides a sequencer variant with a clock that can be modulated at audio rate in relation to the master clock signal. This enables experimental approaches to time, such as the creation of multiple sequencers that gradually drift in and out of phase with one another through audio-rate modulation.

Such scheduling is not limited to audio processing. As all audio processing occurs in the main thread, any instantiated object in the JSRE can be manipulated within the audio callback. Gibberish provides the ability to easily sequence anonymous function calls that can directly affect visual and interactive behaviors. It also offers a convenience method, `future()`, that provides users with a concise means of scheduling a single function invocation with sample accuracy.

Interface.js

Interface.js provides a solution for quickly creating graphical user interfaces (GUIs) using JavaScript that work equally well with mouse- and touch-interaction paradigms. The output of these GUI widgets can control other JavaScript objects (such as Gibberish ugens). The output can also be converted to MIDI or Open Sound Control (OSC, cf. Wright 2005) 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.

Table 2. Interface.js Widgets

Sliders	Vertical Slider, Horizontal Slider, Crossfader, Range Slider, Multislider
Buttons	Toggle, Momentary, Contact, Multibutton, Polygonal Button
Sensors	Accelerometer, Gyroscope,
Miscellaneous	Multitouch *xy* (with physics), Piano Keyboard, Menu, Label, Text Input, Patchbay, Knob

In addition to a wide variety of GUI elements, Interface.js also provides unified access to accelerometer readings and the orientation of mobile devices. The widgets provided by Interface.js are outlined in Table 2.

Event Handling in Interface.js

Interface.js binds widgets to object functions and values using a *target/key* syntax. A widget's target specifies an object the widget should control, 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, and object methods are invoked with the widget value as the first argument.

Widgets providing multiple dimensions of control (e.g., *x-y*, accelerometer) are configured with arrays of targets and keys containing one target and one key for each dimension. Users can also register custom

event handlers to create interactions that are more complex than direct mappings between widgets and property values.

Because most browsers only deal with either touch or mouse events, Interface.js provides another type of event handler, `touchmouse`, which responds to either form of input. This enables developers to create a single callback that will function regardless of the type of device running the interface.

Another event handler, `onvaluechange`, is called every time the value of a widget changes, regardless of whether the widget changes by touch, by the mouse, or by motion. Procedurally changing a widget's value (perhaps owing 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.

Interface Layout and Appearance

The main container unit of Interface.js is the *panel widget*. A panel widget wraps an instance of HTML's *canvas* element, which is a 2-D drawing surface with hardware graphics acceleration support in all current desktop browsers and most current mobile browsers. A panel widget's constructor allows users to specify an HTML element to contain the canvas tag, allowing multiple interfaces to be placed at different points in an HTML document. 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 provided either as absolute values measured in pixels, or as relative values defined in terms of the widget's enclosing panel. Using relative sizes enables users to define interfaces without having to worry much about varying window sizes or the screen sizes of different devices—for example, a slider with a width of 0.5 will always fill half the width of its enclosing panel, regardless of the panel size, which is often in turn determined by the size of the browser window. In certain usage scenarios this approach may not yield satisfactory results, however. For example, although a complex interface with a high number of widgets designed for a large screen will render with the correct proportions on a small screen, the widgets may become too small to be useful as interactive musical controls.

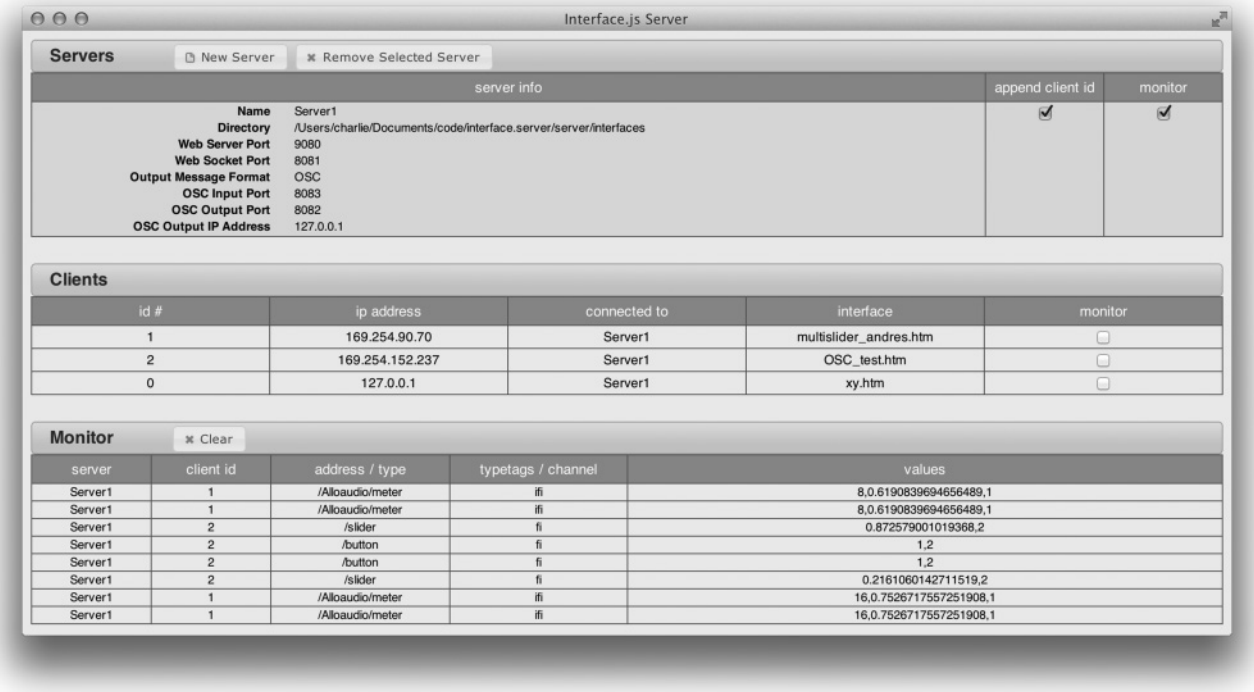
Networked Control

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

Although there have been previous applications that translate WebSocket data into OSC or MIDI messages, we believe our solution is highly efficient. Our application, Interface.Server, includes an HTTP server that serves Interface.js files to Web browsers. After launching Interface.Server, any user directing a browser to the IP address where it is running will receive an HTML page listing interfaces available for use. When a user selects an interface from this list, it is downloaded to the browser and a WebSocket connection is automatically created, linking the browser back to Interface.Server. In other words, the computer running Interface.Server is automatically the recipient of WebSocket messages generated by the interfaces it provides. WebSocket messages sent by interfaces are translated by Interface.Server into OSC or MIDI messages as described in the Interface.js files and forwarded to target applications.

A distinct advantage of this system is that programmers do not need to think about establishing socket connections when developing their interfaces, as no hard-coded IP addresses or autodiscovery protocols are needed. Once the IP address and port of the Interface.Server application is entered into the browser, the URL can be bookmarked

Figure 5. Screenshot of Interface.Server, monitoring the output of three connected interfaces running Web browsers.



for easy access. Individual interfaces can also be bookmarked; in this case a WebSocket connection is established as soon as the bookmarked interface is loaded. On mobile devices, these bookmarks can be represented by icons on users' home screens; simply tapping one of these icons fetches the interface and immediately opens the appropriate connection.

The connection between devices running interfaces and Interface.Server is bidirectional, enabling target applications receiving OSC and MIDI messages from Interface.Server to send messages to connected interfaces and change characteristics of individual widgets. Interface.Server also includes live-coding capabilities so that target applications can dynamically create and modify interface layouts. These capabilities are based on ideas first explored in the mobile application Control (Roberts, Wakefield, and Wright 2012) and are currently supported by libraries for SuperCollider and Gibber. Interface.Server provides a GUI enabling users to easily monitor existing browser connections and the messages they send to target applications, as shown in Figure 5. The GUI also enables users to define multiple target applications. These can reside locally with Interface.Server or on different computers, creating possibilities for distributed computation and audio rendering.

Instruments in the Browser

Our research has considered both low-level and high-level authoring techniques for browser-based DMIs. As a relatively low-level technique, developers can create their own HTML pages containing synthesis graphs and interfaces designed using JavaScript. Developers are also responsible for articulating the minutiae of musical mappings and for publishing their creations to a Web site, assuming they want to make their instrument available for others to use.

At a higher level, end users can create DMIs with Gibber. In Gibber, users deal only with JavaScript and are completely removed from all HTML and CSS markup. Mapping abstractions included in

Figure 6. HTML with embedded JavaScript code to create a two-slider interface controlling a sine oscillator.

```
<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:[ 0,0,1,.5 ], // x,y,width,height
  });

  sliderAmp = new Interface.Slider({
    target:sine, key:'amp',
    label:'amp', bounds:[ 0,.5,1,.5 ],
  });

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

Gibber enable users to map interface elements to synthesis elements in a single line of code, without having to worry about expected value ranges or perceptual output curves. Finally, users can easily publish their instruments to a central database in Gibber, removing the need for a Web hosting service to distribute DMIs.

Integrating Interface.js and Gibberish.js

There are a few simple steps required to use Interface.js and Gibberish.js together to make a musical interface without Gibber. Starting with an HTML document containing the bare minimum necessary tags (`<html>`, `<head>`, and `<body>`), add two `<script>` tags to import the Interface and Gibberish Javascript files. A third `<script>` tag inside the body element contains all user code to create the interface and the audio graph. Figure 6 shows a complete sample interface file consisting of a sine oscillator with frequency and amplitude

Figure 7. A simple instrument created in 22 lines of code and markup using Gibberish.js and Interface.js, as shown in Figure 6, or a single line of JavaScript in Gibber.

controlled by two sliders, and Figure 7 shows a simple instrument created with the code from Figure 6.

Instrument Design in Gibber

Gibber provides numerous abstractions to assist in designing DMIs and a server for storing instruments and compositions. In this section we discuss the server and summarize two abstractions that enable us to condense the 22-line code example shown in Figure 6 into a single line of code:

```
Sine(Slider( ), Slider( ))
```

Mapping and Interface Abstractions

Creating mappings between audio, visual, and interactive elements is a nontrivial task. Many

factors come into play, including signal sampling rates, usable ranges, and whether changes to signals are perceived linearly or logarithmically. Thus, even simple mappings, such as mapping the movement of a slider to the frequency of an oscillator, requires a variety of processing steps to achieve a musically satisfying result.

Using our slider-to-frequency example, first we must set the output range of our slider to a range of frequencies that we want to control. Next, we need to set the output scale of the slider to be logarithmic, to match human perception of pitch. Finally, we need to place a one-pole filter on the slider output, to remove the quantization effects that result from mapping a discrete value (on-screen slider values are resolution-limited to individual pixels and sampled much more slowly than audio rate) to a continuous one. Even the most naive mapping would, at minimum, require the first transformation in order to have a musically meaningful effect.

Georg Essl addressed this mapping problem in his UrMus system (Essl 2010), by creating an abstraction called *FlowBoxes* that enables developers to easily make connections between interactive and audio modalities. Once FlowBoxes wrapping ugens or interactive elements are initialized, they can be connected in UrMus as follows:

```
mySinOsc = FlowBox(FBSinOsc)

FBAccel.X:SetPush(mySinOsc.Freq)
```

In this example, the acceleration on the *x*-axis controls the frequency of an oscillator. UrMus has flexible semantics for defining push and pull relationships between inputs and outputs, and for performing various types of intermediate signal conditioning. In Gibber, we build on the ideas found in UrMus, but abstract them further to make a simpler mental model for programmers. This example could be accomplished with the following code in Gibber:

```
sine = Sine(Accelerometer.X)
```

Gibber forgoes the flexibility of defining push or pull connections that UrMus offers, and instead affords the construction of continuous, multirate mappings via simple assignment. To later change our sine oscillator's frequency to be controlled by the output amplitude envelope of an instantiated drum loop, only a single line of code is required:

```
sine.frequency = drums.Out
```

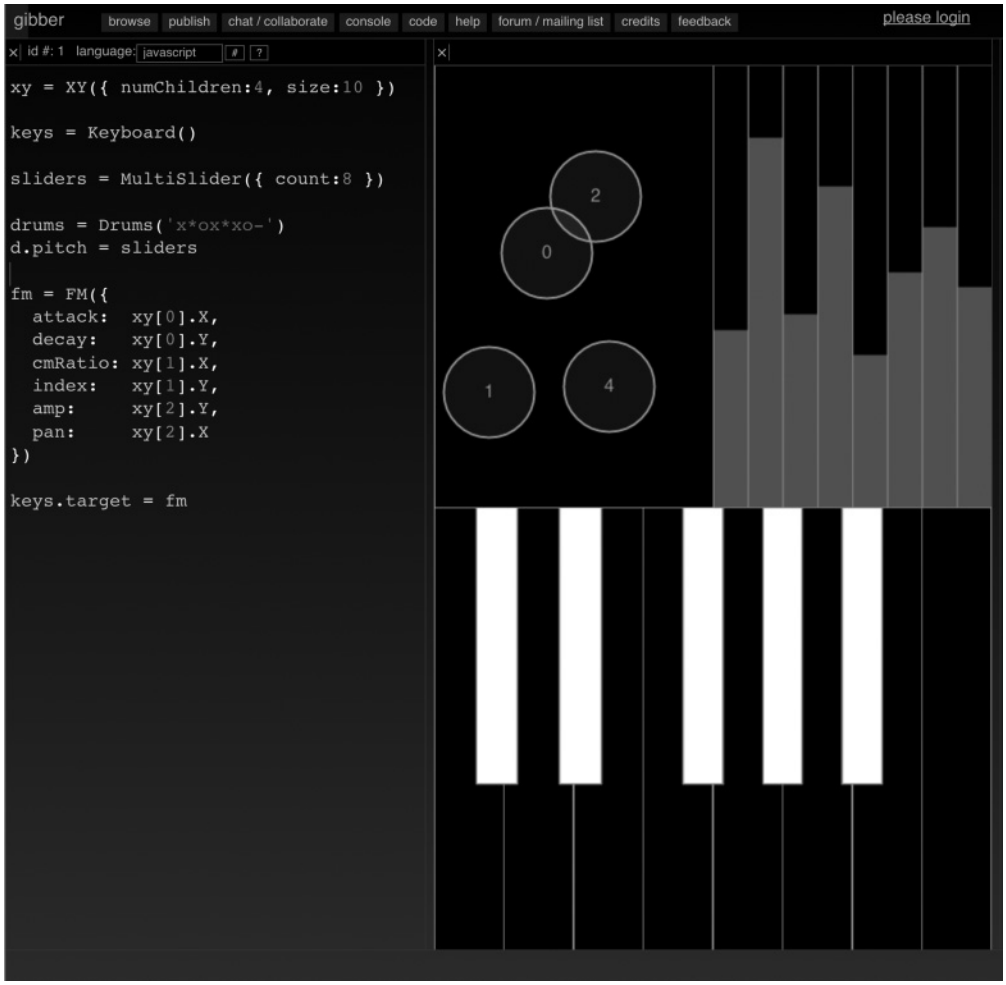
The key in the notation is to capitalize the property name on the right of the assignment operator (in this case `Out`). This notation tells Gibber to make a continuous mapping instead of making a one-time assignment using the instantaneous value. The contribution is removing barriers to the creation of continuous mappings that are potentially multirate and multimodal; all that is required is the capitalization of a single letter in normal assignment. We argue that this encourages experimentation, by significantly lowering the "viscosity" of the notation. The mapping abstractions also extend to mappings between the visual capabilities of Gibber (including 2-D drawing, 3-D scene creation, and live shader programming) and the audio and interactive affordances. As one example, the rotation of a cube can be mapped to continuously control the frequency of an oscillator in a single line of code, with the same ease as mapping interactive elements to control unit generators (Roberts et al. 2014a).

For GUI creation, Gibber takes the elements provided by Interface.js and automatically generates layouts using a subdivision algorithm we first explored in the mobile application Control (Roberts 2011). Although this algorithm performs well for prototyping simple interfaces quickly, Gibber also provides the ability to specify custom boundary boxes for each widget in a fashion equivalent to typical Interface.js programming. Figure 8 shows three widgets in an automatic layout.

Publication, Browsing, and Distribution of DMIs

Gibber provides a central database for publishing, browsing, and distributing programs written by end users; we refer to such programs as *giblets*. The server also enables real-time chat and collaborative editing between Gibber users. When users first publish a giblet, they are only required to give it a name. They can optionally provide a variety of metadata, however, that is helpful for searching the database and browsing the results of queries (shown

Figure 8. A more complex interface in Gibberish, with an xy *widget controlling a variety of parameters on an FM synthesizer, banks of sliders controlling pitch in a drum sequence, and a keyboard that targets the FM synthesizer.*



in Figure 9), including arbitrary user-defined tags and notes on how to run the giblet.

After publishing a giblet, the user is immediately provided with two URLs. The first displays the code associated with the giblet. The second immediately runs the giblet filling the entire browser window; the code and Gibber's interface are not displayed. Because typing URLs on mobile devices can be time-consuming and frustrating, Gibber provides every user account with a URL, enabling users to see all their published giblets in a scrollable list, so that they can easily tap any of their publications to launch it. Giblets launched from this list are immediately executed and scaled to fill the browser window. In addition to launching giblets through this list, users can also view their files and associated metadata in Gibber's file browser, as shown in Figure 9.

Related Work

There are a growing number of options for writing synthesis algorithms using JavaScript. Audiolib.js (see audiolibjs.org) was one of the first significant JavaScript audio libraries written, and it 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

Figure 9. Gibber's browser, showing the search interface, as well as recently added user files and their associated metadata.

leads to more efficient performance for per-sample processing involving dynamic audio graphs. Audiolib.js performs no graph management, leaving programmers to implement their own audio graphs as needed.

Other libraries that use the ScriptProcessor node take markedly different approaches in terms of the APIs they offer. For example, Flocking (Colin and Tindale 2014) enables users to define new ugens declaratively using JavaScript Object Notation (JSON), and Timbre.js (available online at mohayonao.github.io/timbre.js) is an impressive library that enables users to compose ugens in a functional syntax. In addition to its extensive use of code generation and emphasis on per-sample processing, Gibberish.js differentiates itself from these other libraries by providing complex synthesis ugens. One example is a polyphonic, enveloped, two-operator FM synthesis ugen. Another is the Gibberish Monosynth, a three-oscillator band-limited synthesizer with an envelope, a 24-dB resonant filter, and independent tuning and waveshape controls for each oscillator. By including complex ugens, we allow programmers to immediately begin creating music with rich sound sources. Tone.js (online at github.com/TONEnoTONE/Tone.js) is a relatively new JavaScript audio library that also includes complex synthesis ugens.

Although libraries using the ScriptProcessor node have perhaps seen the most development, a variety of other options exist for audio synthesis in the browser environment. One solution for synthesis is JSyn (Burk 1998), a comprehensive Java synthesis library originating over a decade ago. Unfortunately, many browsers do not support Java by default, and others do not support it at all (including Safari on iOS). Work with plug-ins has also explored optimizing Csound for use in the browser using various techniques (Lazzarini et al. 2014). A portable native client (PNaCl) executable that runs Csound was written for desktop Chrome, while Emscripten, a compiler that converts low-level virtual machine bytecode to JavaScript, was used to created an optimized version for Firefox that runs in the ScriptProcessor node. The authors of these ports hope that the Emscripten version of Csound will gain performance improvements with the upcoming Audio Worker implementation. Instead of using the ScriptProcessor node, the Web Audio API eXtension (WAAX) (Choi and Berger 2013) seeks to extend the native nodes of the Web Audio API and make them easier to use; Tone.js also uses native nodes wherever possible. Soliton.js, the JavaScript library that forms the audio backbone of the live-coding environment Lich.js (McKinney 2014), attempts to combine the best of both worlds, using native nodes when available and ScriptProcessor nodes whenever a ugen requires features not provided natively.

There are many other interface libraries for HTML and JavaScript, but few handle both touch and mouse modalities and almost none cater for the needs of musicians and live performers. A notable exception to this is the NexusUI project (Taylor et al. 2014), which provides similar functionality to Interface.js. Where Interface.js only requires use of JavaScript and boilerplate HTML to create interfaces, NexusUI instead uses a more traditional approach of HTML,

CSS, and JavaScript in tandem. The library is very easy to use, and has been used in conjunction with Gibberish.js and other audio libraries to teach teenagers to create DMIs. NexusUI also provides a novel integration with Max/MSP called NexusUp that allows interfaces made with Max widgets to be easily pushed to remote browsers and mirrored using NexusUI widgets. Unlike Interface.js, however, NexusUI does not currently provide bidirectional communication between remote interfaces and target applications. The NexusUI authors intend to provide full bidirectional communication in the future.

Conclusions and Future Work

Gibberish.js, Interface.js, and Gibber are all open source under the MIT license and available for download on GitHub (available online at github.com/charlieroberts/Gibberish, github.com/charlieroberts/interface.js, and github.com/charlieroberts/Gibber, respectively). They have been used in a variety of educational settings, from work with middle-school students to university courses. Universities using elements of our research in their teaching include the University of California at Santa Barbara; Griffith University; the University of Florida; Louisiana State University; Goldsmiths, University of London; and Istanbul Technical University. Going forward, we are particularly interested in expanding the educational potential of Gibber via its social affordances. For example, we plan to let users "star" giblets they like and to enable Gibber's file browser to support ordering search query results by the number of stars each giblet has. Such practices are already common in other online creative coding environments such as Scratch (Resnick et al. 2009).

The authors of Flocking reported very strong performance advantages in comparison with Gibberish (Colin and Tindale 2014) after running an offline rendering test of a sine oscillator with modulated vibrato. Repeating their test with the most recent versions of Flocking and Gibberish yielded a 50 percent performance advantage for Flocking in Chrome Canary under Mac OS X on a 2.6 GHz i7 Macbook Pro, significant but much less than originally reported. Using the testing framework created by the authors of Flocking, we designed another test, creating a graph of 50 audio-rate sine oscillators. The tests are available online at github.com/charlieroberts/webaudio-performance-benchmarks. In this test, Gibberish performance was found to be roughly equivalent to that of Flocking in Chrome. These results, in conjunction with those reported previously by the authors of Flocking, show the difficulty in making generalizations about performance between JavaScript audio libraries. Comparisons must also take into account the architectural difference of Gibberish's per-sample processing versus Flocking's processing of audio in blocks of 64 samples by default; a block size of 64 samples was used by Flocking in all tests described here. Per-sample processing typically comes with a performance penalty but also enables various important features of Gibberish that are unavailable in Flocking, including sample-accurate scheduling, single-sample delay feedback networks, and intra-block graph reconfiguration. Correspondence with the authors of Flocking to examine these testing issues has also included discussion of collaboration to create a testing suite for libraries that use the ScriptProcessor node. We look forward to comprehensively analyzing the performance of Gibberish in comparison with other libraries with these future tests, while bearing in mind that performance results are best analyzed alongside architectural decisions that determine functionality.

We are planning the development of a JavaScript audio analysis library that could function either on its own, in conjunction with Gibberish, or inside of Gibber. At the time of writing, Gibberish only provides an envelope follower for analysis, whereas Gibber adds a FFT analyzing its master output that can be used to help generate audio-reactive visuals. Ideally, this library would take advantage of the ScriptProcessor node as well as the native Web Audio analysis nodes for efficiency and would perform both real-time and offline analysis. We look forward to exploring the Audio Worker specification once the standard has cross-browser support.

Acknowledgments

This article combines and extends research first presented by the authors at recent New Interfaces for Musical Expression (NIME) conferences (Roberts, Wakefield, and Wright 2013; Roberts et al. 2014b). The development of Gibberish.js, Interface.js and Gibber has been generously supported by the Robert W. Deutsch Foundation.

References

Aycock, J. 2003. "A Brief History of Just-in-Time." *ACM Computing Surveys* 35(2):97–113.

Burk, P. 1998. "JSyn: A Real-Time Synthesis API for Java." In *Proceedings of the International Computer Music Conference*, pp. 252–255.

Choi, H., and J. Berger. 2013. "WAAX: Web Audio API Extension." In *Proceedings of International Conference on New Interfaces for Musical Expression*, pp. 499–502.

Colin, C., and A. Tindale. 2014. "Flocking: A Framework for Declarative Music-Making on the Web." In *Proceedings of the International Computer Music Conference*, pp. 1550–1557.

Essl, G. 2010. "UrSound: Live Patching of Audio and Multimedia Using a Multi-Rate Normed Single-Stream Data-Flow Engine." In *Proceedings of the International Computer Music Conference*, pp. 534–537.

Herczeg, Z., et al. 2009. "Guidelines for JavaScript Programs: Are They Still Necessary?" In *Proceedings of the Eleventh Symposium on Programming Languages and Software Tools and Seventh Nordic Workshop on Model Driven Software Engineering*, pp. 59–71.

Lazzarini, V., et al. 2014. "Csound on the Web." In *Proceedings of the Linux Audio Conference*, pp. 77–84.

McKinney, C. 2014. "Quick Live Coding Collaboration in the Web Browser." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 379–382.

Resnick, M., et al. 2009. "Scratch: Programming for All." *Communications of the ACM* 52(11):60–67.

Roberts, C. 2011. "Control: Software for End-User Interface Programming and Interactive Performance." In *Proceedings of the International Computer Music Conference*, pp. 425–428.

Roberts, C., and J. Kuchera-Morin. 2012. "Gibber: Live Coding Audio in the Browser." In *Proceedings of the International Computer Music Conference*, pp. 64–69.

Roberts, C., G. Wakefield, and M. Wright. 2012. "Mobile Controls on-the-Fly: An Abstraction for Distributed NIMEs." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 474–478.

Roberts, C., G. Wakefield, and M. Wright. 2013. "The Web Browser as Synthesizer and Interface." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 313–318.

Roberts, C., et al. 2014a. "Gibber: Abstractions for Creative Multimodal Programming." In *Proceedings of the ACM Multimedia Conference*, ACM, pp. 67–76.

Roberts, C., et al. 2014b. "Rapid Creation and Publication of Digital Musical Instruments." In *Proceedings of the Interational Conference on New Interfaces for Musical Expression*, pp. 239–242.

Taylor, B., et al. 2014. "Simplified Expressive Mobile Development with NexusUI, NexusUp, and NexusDrop." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 257–262.

Wakefield, G. 2012. "Real-Time Meta-Programming for Interactive Computational Arts." PhD dissertation, University of California Santa Barbara.

Wilson, C. 2013. "A Tale of Two Clocks: Scheduling Web Audio with Precision." Available online at www.html5rocks.com/en/tutorials/audio/scheduling/. Accessed 18 December 2014.

Wilson, C. 2014. "Worker-Based ScriptProcessorNode." Available online at github.com/WebAudio/web-audio-api/issues/113. Accessed 18 December 2014.

Wright, M. 2005. "Open Sound Control: An Enabling Technology for Musical Networking." *Organised Sound* 10(3):193–200.

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