GIBBER: LIVE CODING AUDIO IN THE BROWSER

Charles Roberts, JoAnn Kuchera-Morin

University of California at Santa Barbara
Media Arts & Technology Program
charlie@charlie-roberts.com, jkm@create.ucsb.edu

ABSTRACT

We present *Gibber*: a live coding environment for web browsers. Gibber performances are written in pure JavaScript with no syntactical additions or modifications; this enables Gibber code to be executed in any web page viewed inside a browser implementing a realtime audio API. Gibber offers an array of synthesis options (FM, granular, subtractive, physical modeling), audio effects and sequencing objects to control them. The Gibber environment enables simple networked performances where multiple users simultaneously control a remote instance of Gibber. We strove to make the syntax of Gibber clear and concise; when coupled with the ability to run examples in any web page this gives Gibber interesting possibilities as an educational tool.

1. INTRODUCTION

The performance practice of live coding has grown dramatically over the course of the last decade and the increasing number of practitioners has been accompanied by a growing number of languages and environments. Into this context we introduce *Gibber*: a JavaScript library for perfoming high-level audio synthesis and sequencing in web browsers. In addition to its JavaScript implementation Gibber also includes a performance environment that can be run in browsers implementing a realtime audio API; Google's Chrome browser currently provides the best audio experience but other browsers are also supported. Gibber is built on top of many open-source JavaScript libraries, the most important of these being audioLib.js by Jussi Kalliokoski[1]. We layered an extremely terse syntax on top of audioLib.js and extended it to include a variety of sequencing options, FM synthesis, granular synthesis, Karplus-Strong physical modeling, flanging, waveshaping, buffer shuffling and a number of other oscillators and effects. Some of these have already been contributed back to audioLib.js; we hope to contribute more in the future.

1.1. Motivation

The creation of Gibber was motivated by the desire to use JavaScript as a language for browser-based live coding performances. When work on Gibber began, no musical live coding environments using JavaScript existed. Although there are many visual environments that let you change JavaScript code and quickly see the updated results, few are geared towards live performance [1]. JavaScript, although sometimes maligned[2], is an excellent language for live coding. Its prototypical nature makes it easy to extend and combine objects, it is dynamic, features object introspection and meta-programming and has first-class functions with closures. This flexibility enabled us to create a syntax in which the creation and playback of a sine wave as simple as:

```
Sine(440, .5);
```

In the above example, 440 is the frequency and .5 is the amplitude of the wave form. Throughout the creation of Gibber we strove for this level of simple, declarative syntax. The next example creates a triangle oscillator and adds distortion and reverb to it:

```
t = Tri(220, .5);
t.fx.add( Dist(), Reverb() );
```

By coupling this syntax with an accessible web-based environment, we hope to provide a vehicle for students to easily experiment with synthesis. Towards this end, we included a number of tutorials on additive, subtractive and FM synthesis with Gibber.

Gibber also supports networked live coding performances. To join a networked performance performers simply launch Gibber in a browser, choose a username and enter the IP address of a remote computer running the Gibber environment. Any code sent to this remote instance will be displayed for the audience and other performers to see (assuming video from the remote computer is projected as per the typical live coding performance) and executed in the remote context.

1.2. The Browser As A Realtime Synthesis Platform

Over the past twenty years digital audio practitioners have witnessed the gradual evolution of the the browser as a realtime music platform. From its humble beginnings as a player of (often aesthetically questionable) general MIDI files, to the use of Java and Flash plugins to synthesize audio, to the current trend of using native JavaScript for sample-level synthesis, the browser has, almost since its

[1]One exception to this is the excellent Livecodelab found at http://www.sketchpatch.net/livecodelab/

[2]http://wtfjs.org

inception, been an environment for realtime music creation and consumption.

Support for JavaScript based audio synthesis varies wildly across current web browsers. Google's Chrome and Chromium provide the best support; Firefox is a close second but appears to operate less efficiently and with inferior timing. Gibber's default script (featuring reverbs, delays, FM synthesis, physical modeling, drum samples and various other audio effects) uses an average of 30% of one core on a 2.8 GHz Intel Core i7 processor under Chrome while Firefox uses an average of 46%. JavaScript does receive access to audio output buffers in Internet Explorer and realtime audio is only available in a limited form in beta versions of Safari. No mobile web browsers currently support JavaScript audio synthesis. Apple's security restrictions against JIT compilation in mobile web browsers have stopped the WebGL standard from being implemented in iOS for many years and similarly block realtime audio implementations in the browser; there is little reason to hope that these restrictions will change in the near future.

These concerns aside, at some point in the future a standard for web audio will be agreed upon and provide browser developers with greater motivation to properly implement realtime audio capabilities. Various standards have already been suggested; they range from a relatively simple API providing access to input and output sample streams[2] to a much more complex API suggested by Google[8] that provides C++ implementations of convolution and other expensive DSP algorithms in order to obtain performance improvements over JIT-compiled JavaScript.

It has been a distinct pleasure using the same programming language to implement our coding environment, perform networking for distributed performances and synthesize realtime audio. Although there are efficiency limitations for JavaScript-based audio synthesis when compared to C/C++, these limitations are balanced by the benefits of an accessible platform with a unified development language.

1.3. Related Work

At the time of this writing, toplap.org (an umbrella website for performers interested in live coding) lists thirty-four active live coding environments; this is by no means a comprehensive list. In order to position Gibber within this extensive and growing community, we compare the following attributes and draw upon a number of live coding implementations that were influential in the creation of Gibber:

- Language - Is a new programming language implemented or is an existing language used?
- Synthesis Source - Are synthesis algorithms written in the same language used for performance? Is the live coding language being used to control an external application?
- High-Level / Low-Level - Does the environment manage an audio graph? Do you have access to low-level sample output?

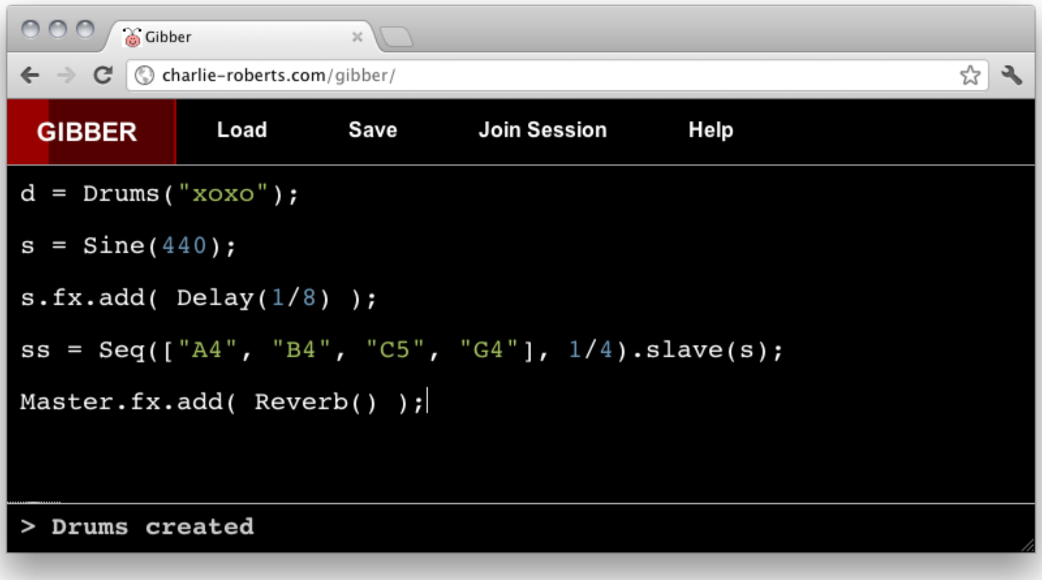
In terms of language, Gibber is implemented in pure JavaScript with no syntatic additions or modifications. This is similiar to live coding environments using languages descended from Lisp: *Impromptu*[10], which is written in Scheme, and *Overtone*[3], which uses Clojure. JavaScript itself borrows a great deal from these functional languages, including functions as first-class objects and closures. Other langauges such as *ixi lang*[6] and *ChucK*[12] implement new languages for live coding. Another JavaScript live coding environment was recently announced[4] focusing on graphics and low-level audio synthesis. This environment adds special keywords to JavaScript; code written with these synatactical additions cannot be run using a standard JavaScript interpreter without preprocessing.

Many live coding environments include synthesis algorithms written in the same language that is used for performance while others are used to control external audio sources. Impromptu, for example, controls Audio Unit plugins in OSX; this provides access to an extremely rich pallete of commercial and open-source synthesizers and effects. ixi lang and Overtone both control the audio server found in *SuperCollider*[7]. Low-level unit generators in are programmed in C/C++ in ChucK; however, these can be combined at a higher level using the ChucK language itself. *LuaAV*[11] takes a relatively unique approach to live coding synthesis where graphs of operator expressions are defined using the interpreted Lua language, and then JIT compiled to machine code to obtain the best performance possible. In Gibber, synthesis is performed internally using JavaScript. The audio quality of Gibber currently suffers when compared to platforms that rely on richly developed synthesis tools like the SuperCollider server or various commercial audio plugins.

Gibber's primary focus is on pattern-based music generation; it features high-level methods that afford sample-accurate sequencing and synthesis capabilities. In this way it is very similar to (and directly inspired by) ixi lang, the aforementioned language layered on top of SuperCollider. Although performance practice using Gibber primarily consists of executing high-level commands to control sequence and synthesis parameters, Gibber does provide the ability to override the default audio callback so that users can directly fill output buffers with samples using Gibber unit generators.

2. IMPLEMENTATION

Gibber is a JavaScript library for the declarative creation and sequencing of audio synthesis graphs. It also contributes a performance environment that can be run in web browsers and affords simple networked music performances. We will discuss the JavaScript syntax and performance environment separately.



```
d = Drums("xoxo");

s = Sine(440);

s.fx.add( Delay(1/8) );

ss = Seq(["A4", "B4", "C5", "G4"], 1/4).slave(s);

Master.fx.add( Reverb() );
```

```
> Drums created
```

Figure 1. Gibber running inside of Google Chrome. The red bar in the upper left is a metronome indicating the current beat in 4/4 time; beat one is currently highlighted.

2.1. Performance Environment

Gibber runs in web browsers that implement an API for realtime audio synthesis. Users can either download the performance environment from GitHub to run on their own computer[3] or they can simply visit the Gibber website[4]. The peformance environment features a code editor built using the open-source project CodeMirror[5]. The editor provided by the CodeMirror library affords syntax highlighting, line numbering, customized GUI themes and keystroke shortcuts amongst many other features.

One important addition to the code editor in Gibber is sample-accurate delay of code execution. Although selected code can be executed immediately via a keystroke combination, other keystroke combinations allows code execution to be delayed until the first sample of the next musical measure or beat. This ensures that the various sequences that comprise a Gibber performance will be started and stopped in sample-accurate synchronization if desired.

The setup for a Gibber networked performance is very simple. A remote computer runs an instance of Gibber that multiple clients connect to. Performers can edit code on their personal computer, monitor the results via headphones, and then send it to the remote computer to be projected and executed if they like the results. In order to host a networked Gibber session, the remote computer needs to run a Node.js[6] script included in the Gibber download; there is also a single line of code to execute within the Gibber code editor to force it to accept network connections.

Performers only need to enter the IP address of the remote Gibber instance and a username to begin transmitting code. The ability to delay code until the first sample of a musical measure comes in handy for networked performances; clients are assured that their code will be executed precisely on the downbeat of the next measure if desired, regardless of network jitter and latency. At worst, under extremely high latency, the code will be executed a measure later than intended but will still generate results that are in phase with the rest of the performance. In addition to displaying code, the remote computer also displays a realtime chatroom that allows performers to communicate during performances; this practice of displaying dialogue amongst performers can also be found in the current network setup used by the Hub and in the recently introduced live coding environment LOLC[5].

In addition to the code editor, the performance environment also features a menubar containing quickstart instructions, GUIs for loading/saving files and joining distributed performances, a list of key commands and general information about the Gibber project. A sample Gibber session is presented in Figure 1.

2.2. Features

2.2.1. *Syntax*

As mentioned in the introduction, the audio synthesis in Gibber is layered on top of the audioLib.js library. This library comes with a number of unit generators and also handles browser discrepancies in providing access to output sample buffers. Although excellent for general use,

[3]https://github.com/charlieroberts/Gibber
[4]http://www.charlie-roberts.com/gibber
[5]http://codemirror.net
[6]http://nodejs.org

the audioLib.js syntax is unneccesarily verbose for live coding. For example, compare the process of defining a 440 Hz square wave in audioLib.js and Gibber:

```
//audioLib.js
s = new audioLib.Oscillator(44100, 440);
s.waveShape = "square";

// Gibber
s = Square(440);
```

Note that the above audioLib.js code does not generate any audio output; a programmer would need to manually append the oscillator output to a buffer feeding the DAC, presumably inside a sample loop. In Gibber, the square wave oscillator is automatically added to a signal processing graph and begins outputting audio immediately.

To further improve terseness and decrease the potential for typos during performance, we abandon some JavaScript best practices and take advantage of questionable language features. The Gibber environment makes extensive use of the global namespace; in doing so we avoid the need to prefix constructors with a namespace identifier and also mitigate the need to use the `var` keyword when declaring variables. We also promote the use of single letter variable names; audio and sequencing objects held in single letter variables are automatically replaced in the audio / control graphs when reassigned. Consider the following lines of code:

```
t = Tri();
s = Seq([440, 880], 1/4).slave(t);
```

These two lines of code create a triangle oscillator and sequence it to alternate between pitches of 440 and 880 Hz every quarter note (sequencing will be discussed in greater depth shortly). By using single letter variable names, we can easily substitute a square wave in place of the triangle wave by replacing `Tri()` with `Square()` and re-executing the first line of code. This causes the triangle oscillator to be removed from the audio graph; Gibber also will notice that the triangle oscillator was slaved to our sequencer and assign our new square oscillator to be slaved. Single letter variables in Gibber can be thought to function similarly to *proxies* found in the JITLib quark for SuperCollider[9].

2.2.2. *Modulation and Effects*

Modulation and effects are also managed automatically by the Gibber audio graph. To apply modulation, a programmer specifies the parameter to modulate, a source for the modulation and how the modulation output should be applied. Below is a simple example for vibrato.

```
s = Sine(440);

// Vibrato : +/- 8 Hz at a rate of 4 Hz
s.mod("frequency", Sine(4, 8), "+");
```

The "+" shows that we add the ouput of the modulating sine wave to the frequency component; other operators include assignment, multiplication, division etc. As another example, here is an unenveloped FM bell:

```
carFreq = 200;
c = Sine(carFreq, .15);
cmRatio = 1.4;
index = 190;

c.mod("freq",
      Sine(carFreq * cmRatio, index),
      "+");
```

Each oscillator has an effects chain that we can add to and remove from:

```
s = Sine(440);
s.fx.add( Flanger(), Delay(), Reverb() );

// remove flanger
s.fx.remove("Flanger");
// remove first effect in chain
s.fx.remove(0);
// remove all effects
s.fx.remove();
```

Any effect parameter and any modulation parameter can be modulated; modulations are applied recursively within the audio signal graph. There is also a global *Master* object that allows us to apply effects to the summed audio output of all oscillators.

2.2.3. *Rhythm and Timing*

Choosing a system for notating time and rhythm in Gibber has been an interesting challenge. Early on we decided that there would be an initial emphasis on using 4/4 time. With this empahsis it makes sense that there would be an easy way to indicate both divisions and multiples of a musical measure, however, we also wanted to retain the freedom to specify precise durations in samples. The initial solution to this was to create a set of variables holding the number of samples for subdvisions of a measure ranging from a whole note to a sixty-fourth note. These variables were preceded by an underscore character; thus, `_4` was a quarter note, `_16` was a sixteenth note etc. Durations lasting longer than a measure could also be easily notated, for example, `_1 * 2.25` represents the number of samples in two measures and a quarter note.

Although this system worked well for a number of both live performances and screencasts, it always felt contrived and was visually unappealing. When considering possible alternatives the potential visual representations of durations hinted at a better solution. Instead of using variables to represent subdivisions of a measure, we now assume that a musical measure is the standard unit of duration and simply pass a value of `1` to functions accepting a duration as a parameter in order to indicate a measure in length. In this system `1/8` represents an eighth note, `1/16` represents a sixteenth note and `16` represents sixteen measures in length. This is visually much more elegant than our previous solution and much terser for declaring lengths greater than one measure in length, for example: `_1 * 4.5` simply becomes `4.5`.

There are of course problems that arise when using integer values to represent both durations measured in samples and durations measured as multiples of a musical measure. In Gibber we assume that any value representing time less than sixty-four represents a duration measured as a multiple of a measure, while higher numbers represent durations measured in samples. Sixty-four is a default value that can be changed for types of music where longer rests or durations are needed (e.g. the gong of a gamelan can wait up to 128 measures between strikes).

2.2.4. *Sequencing*

Although there are a number of objects used to sequence parameters in Gibber, each is derived from the fundamental *Seq* (sequencer) object. The Seq object can be used to sequence any type of data. As examples, one Seq object might pass chord identifiers as strings to an Arpeggiator while another might call a series of different functions, perhaps alternating every measure between telling another Seq to randomize its values and then resetting them to their original contents. Virtually any aspect of Gibber can be controlled by a Seq object. In the code sample beloew, the first Seq controls the pitch of an oscillator while the second randomizes the order of values in the first and resets them to their original positions every four measures.

```
s = Sine(440);

// pass an array, a duration for each step
// slave the previously created sine ugen
ss = Seq([ 440,660,880 ], 1/4).slave(s);

// pass two methods to alternate calling
// every four measures
sss = Seq([ss.shuffle, ss.reset], 4);
```

The Seq object makes a distinction between the *sequence* of objects or functions it holds and the *durations* that define when these functions are executed. An array of durations can be passed to the Seq constructor instead of a single duration enabling arbitrary rhythms to be sequenced. This separation also allows for greater control of algorithmic processes as sequence and duration values can be accessed programmatically using different techniques.

By default, progression through the sequence and durations arrays occurs linearly. However, users can assign an arbitrary `pick` function to either the sequence or durations array that can instead determine which array position to use when the sequencer advances. Gibber includes a `surpriseMe()` function that randomly picks an item from an array, and a `weight()` function that allows users to weight the likelihood of each item being chosen. In the example below, we see a Seq object where frequencies are randomly selected while durations are heavily weighted towards eighth notes as opposed to quarter and half notes.

```
// pass arrays of frequencies and durations
s = Seq( [440,660,880], [1/8,1/4,1/2] );

// tell Seq to randomly pick frequencies
s.sequence.pick = surpriseMe();

// use a weighted random function to pick
// durations
s.durations.pick = weight(.8, .1, .1);
```

Gibber defines other objects extending Seq that provide additional functionality:

- Arp - allows you to easily arpeggiate chords identified by nomenclature such as “G4m7” and “C4Maj7b9”.
- Drums - in addition to providing drum samples this object can be quickly sequenced using a string such as “xoxo” to represent kick and snare hits.
- ScaleSeq - Allows you to sequence a series of notes in a particular mode that are offsets of a root value. For example in A aeolian 0 = A, 1 = B, 2 = C etc.

All Seq objects register to receive messages from the Gibber master clock so that they update their step durations whenever the master tempo changes.

3. CONCLUSIONS AND FUTURE WORK

Gibber has been used in a number of performances since its introduction. The first formal concert performance was by the CREATE Ensemble at UC Santa Barbara, in which six performers submitted code to a remote computer for execution. It is worth noting that one of the ensemble members had no programming experience but was still able to participate by copy and pasting code and modifying variable values. The first author has also performed solo with Gibber both in formal and informal settings.

We are excited about the educational potential of a syntatically clear live coding environment presented on the web. We imagine classroom scenarios where a lecturer instructs students to explore a synthesis topic; after providing some time for experimentation the lecturer could then ask for volunteers to submit code to a remote instance of Gibber where it could be projected, executed and critiqued by the class. To further this idea we have included tutorials on various synthesis techniques in Gibber and plan to incorporate more.

In terms of future work the audio synthesis capbabilities of Gibber need to be expanded as the palette of available sounds is currently somewhat limited. There is also a great deal of optimization to be done. As one example, all modulation sources currently run at audio rate; control-rate modulation would greatly improve efficiency. We look forward to adding generative graphics capabilities to Gibber and exploring strategies for controlling both audio and visuals concurrently. Finally, we hope to further expand Gibber’s networked performance capabilities with the addition of audio graph visualizations that inform users about the activities of their fellow performers.

4. ACKNOWLEDGMENTS

Special thanks to all the open-source projects that contribute to Gibber. This research was supported by NSF grants IIS-0855279 / IIS-1047678 and by a graduate fellowship from the Robert W. Deutsch Foundation.

5. REFERENCES

[1] http://github.com/jussi-kalliokoski/audiolib.js.

[2] http://blog.aventine.se/post/18627646284/simple-audio.

[3] http://github.com/overtone.

[4] http://livecoder.net/.

[5] J. Freeman and A. Troyer, “Collaborative textual improvisation in a laptop ensemble,” *Computer Music Journal*, vol. 35, no. 2, pp. 8–21, 2011.

[6] T. Magnusson, “ixi lang: a supercollider parasite for live coding,” in *Proceedings of the International Computer Music Conference*. University of Huddersfield, 2011.

[7] J. McCartney, “Rethinking the computer music language: Supercollider,” *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[8] C. Rogers, “The web audio api,” W3C, W3C Editor’s Draft, 2012, https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html.

[9] J. Rohrhuber and A. de Campo, “The supercollider book,” S. Wilson, D. Cottle, and N. Collins, Eds. The MIT Press, 2011, ch. 7, pp. 207–236.

[10] A. Sorensen, “Impromptu: An interactive programming environment for composition and performance,” in *Proceedings of the Australasian Computer Music Conference 2009*, 2005.

[11] G. Wakefield, W. Smith, and C. Roberts, “LuaAV: Extensibility and Heterogeneity for Audiovisual Computing,” *Proceedings of Linux Audio Conference*, 2010.

[12] G. Wang, P. Cook *et al.*, “Chuck: A concurrent, on-the-fly audio programming language,” in *Proceedings of International Computer Music Conference*, 2003, pp. 219–226.