BRAID: A Web Audio Instrument Builder with Embedded Code Blocks

Ben Taylor and Jesse Allison
Louisiana State University
Center for Computation and Technology
Baton Rouge, Louisiana
{btayl61, jtallison}@lsu.edu

ABSTRACT

Braid (Browser Audio Interface and Database) is a web audio instrument-building environment developed with the NexusUI platform. To identify the requirements of such an environment, the utility of NexusUI as an audio interface engine for browser-based projects is reviewed. The addition of inline web audio within a drag-and-drop interface-building environment is discussed. A consideration of a modified Model-View-Controller architecture to integrate DSP code and interface is followed by an examination of the workflow of designing browser-based instruments within Braid. Finally, a database for saving and sharing web audio instruments for performance or audience distribution is described.

Categories and Subject Descriptors

H.5.2 [**User Interfaces**]: Evaluation/methodology; H.5.2 [**User Interfaces**]: Graphical User Interfaces (GUI); H.5.5 [**Sound and Music Computing**]: Methodologies and techniques; D.2.6 [**Programming Environments**]: Interactive environments

General Terms

Design, Performance, Management, Human Factors

Keywords

Web audio, audio interface, MVC, instrument design, digital audio, creative code, live-coding user interface

1. INTRODUCTION

The web browser has become a powerful tool for sonic interaction [5,6,11,23], instrument distribution [3], and databases of musical informatics [9]. Meanwhile, mobile instrument design has flourished through a mature community of designers and toolkits [4,10,18,20]. Web Audio [21] and browser-based instrument design have broad potential to enable the creation and sharing of cross-device desktop and mobile instruments. Technical barriers to building and sharing web audio instruments are many: designing creative and compatible interfaces, writing code in several languages (HTML, CSS, JavaScript), hosting your instrument online, and connecting your instrument to interested users.

Recent toolkits assist with the programming of web audio instruments, most notably Gibber's Interface.js [16]. Additional web environments like WebPD[1], Patchwork[2], True Grid[3], and WebModular[4] offer varying levels of customizability within Patcher-like or modular synth-like formats.

Several priorities come to the fore when attempting to reduce barriers to web audio instrument design:

- **Audio Flexibility:** To maintain user access to low-level web audio programming, allowing for full flexibility and customization of DSP and synthesis ideas. Digital audio programming environments have flourished in the past two decades because they enable rich creativity in sound design through the expressivity of coding. Users should be able to focus on Audio DSP code without having to worry about the fragmented workflow and technicalities of web development.

- **Interface Flexibility:** To give users broad control over their interface, meaning choices of standard and non-standard interface components, extensibility of interface components, and easy access to the interface's layout through a drag-and-drop system.

- **Distribution and Access:** To enable saving and sharing instruments as a way to spread ideas, web audio strategies, and encourage distribution of web audio instruments.

These concerns come together to inspire Braid [2], a web audio instrument-building environment in the browser, with drag-and-drop interface editing, inline code editors for integrating web audio, and connection to a cloud database for storage and distribution of instruments. The Braid name is indicative of weaving together code, interface, and distribution while maintaining the integrity of each of these tenants of instrument design. Braid is built upon several existing toolkits, including the Nexus User Interface platform [1] and Gibber.lib[5], and responds to instrument database ideas presented in [17].

[1]http://github.com/sebpiq/WebPd
[2]http://www.patchwork-synth.com/
[3]http://www.modulargrid.net/racks/synth
[4]http://www.g200kg.com/jp/docs/webmodular/
[5]http://charlie-roberts.com/gibber/gibber-lib-js/

1.1 Nexus User Interface

Nexus User Interface (NexusUI) is a library of multipurpose HTML5 interface widgets developed previously and described in ”Simplified Expressive Mobile Development with NexusUI, NexusUp, and NexusDrop” [19]. The NexusUI project includes integration with several server paradigms that send Open Sound Control [22] data to other audio applications, as well as providing editable JavaScript callback functions which can enact JavaScript and web audio. NexusUI is touch-compatible and designed especially for use with mobile devices. See [19] for a more in-depth description of NexusUI.

1.1.1 *Interface Widgets*

NexusUI includes many standard electronic music control interfaces including buttons, toggles, dials, sliders, and multisliders (see Figure 1). The UI library also includes more complex and mobile-specific control interfaces such as multitouch controls, tilt sensors, sequencers, and animated physics objects. Each widget is drawn upon its own HTML5 canvas which can be scaled and positioned with CSS. To coordinate and manage widgets, NexusUI instantiates a central object, defined as *nx*, which provides shared functions and which can create, monitor, and remove widgets dynamically.

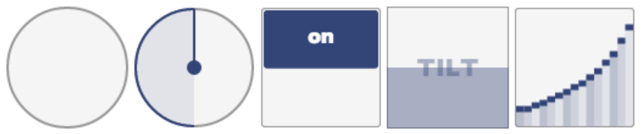


Figure 1: Five NexusUI widgets

1.1.2 *NexusDrop Interface Builder*

NexusDrop is a drag-and-drop editor for NexusUI which opens the platform to a wider userbase; no knowledge of web programming is necessary. Interfaces built with NexusDrop send OSC to Max/MSP or another audio environment when hosted on a local server. NexusDrop interfaces can be downloaded as HTML, but cannot be saved online for further editing.

1.1.3 *Web Audio Usability Concerns*

NexusUI has far-reaching applications as an interface library for web audio projects. Each widget’s programmable callback function allows custom JavaScript including web audio to be executed using the widget’s data output. However, a considerable drawback of the NexusDrop drag-and-drop interface builder is its lack of integration with web audio. NexusDrop is designed as a prototyping environment for mobile interfaces that send OSC or ajax to other audio applications. Since NexusDrop does not involve coding, using it to control Web Audio must be done in a circuitous route by downloading the HTML page and then writing web audio into it. Any updates to the UI at a later time require manually creating the new UI code or recreating, reexporting and importing the web-audio code into the updated interface. While NexusDrop is a powerful interface tool when used as a control for other audio applications, significant changes are required to make it useful for creating web-audio based instruments.

2. BRAID: EMBEDDING CODE EDITORS

Built atop NexusDrop, Braid enables embedding web audio code blocks within widgets in a drag-and-drop environment. Like other browser-based audio coding environments such as Gibber [7] and Lich.js [14], audio code is written directly into the browser. Unlike those environments, the code is predominantly integrated with a visual control interface. This is accomplished through a popup code editor, in which users can write callback functions for individual widgets.

2.1 Exploring the Model-View-Controller Relationship

To maintain the flexibility of both the interface and the instrument’s signal processing, web audio code is integrated into the interface using some aspects of a Model-View-Controller architecture.

Model-View-Controller (MVC) is a common configuration for software applications [15], while recent research by Jamoma describes its relevance to audio applications [12]. In an MVC design, the *view* (a visual representation of data), *controller* (interface), and *model* (the underlying data) are independent. Interaction with controller affects the model, which updates the view. (An in-depth explanation of the MVC relationship within audio applications can be found in [12]). The MVC paradigm has several advantages, including the ability to create several different views for the same model, as well as allowing the development of the interface (controller) and logic (model) independently and unencumbered by the other.

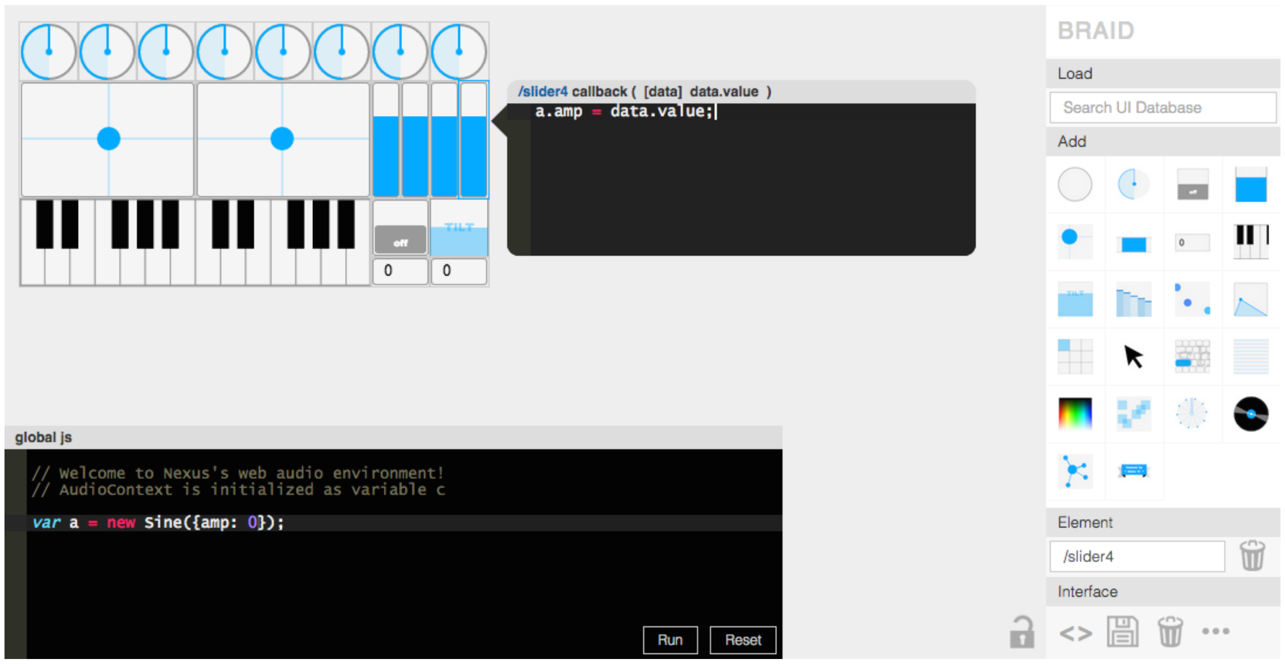
In addressing usability concerns, Braid separates the controller and the model of a Braid instrument, in order to build each in its most natural environment. In Braid, the *controller* is the HTML5 interface that is built by drag-and-drop. The *model* is the web audio code that is written by the user in a code editor. A Braid user then creates custom mappings between the controller and the model by writing callback functions which link each controller to aspects of the model. These mappings are written in an embedded code editor connected to each interface object. In this way, the controller and the model can each be edited independently in their logical environment: a drag-and-drop view and a line-by-line DSP code editor.

In contrast, an example of a less-separate MVC architecture in web audio is Patchwork, in which DSP functions are tied to specific user interfaces, and their arrangement may affect both the DSP model and the controller. To a Patchwork user, the DSP model and the interface controller are inseparable; when a different DSP module is inserted in the chain, the model, view, and controller aspects all change thereby conflating the interface and the underlying audio processing.

While an interface like Patchwork’s is useful for easy patching and prototyping, we designed Braid with more independence between controller design and model design in order to encourage more diverse relationships between controllers and models, to maintain the creativity of web audio coding, and to offer intuitive drag-and-drop creation of the controller interface.

2.1.1 *Field*

An interesting reference for Braid is the creative coding



```
/slider4 callback ( [data] data.value )
  a.amp = data.value;
```

```
// Welcome to Nexus's web audio environment!
// AudioContext is initialized as variable c

var a = new Sine({amp: 0});
```

Figure 2: Braid

environment Field[6], which uses a related system of attaching code to visual items within a drag-and-drop environment. An example of the utility of this approach within Field is the ability to draw a shape with the mouse and then algorithmically multiply it in a code editor. The Field environment keeps both the visual and programmatic editors *visible* and *continuously interactive*, creating a hybrid working space in which neither aspect is more focal.

2.2 Legibility

While full MVC separation has its aforementioned benefits, Braid uses a more visually-integrated system in order to increase readability during the design process. Braid helps users visualize the association between interface elements and callback code, as opposed to keeping all code in a single document that is visually divorced from the interface. For example, we take a different approach from Gibber's declarative interface building, which are organized in a single coding window. Braid's approach puts more visual emphasis on the interface and less on the UI coding process.

2.3 Workflow

Braid is developed partially in response to experiences teaching NexusUI and Gibberish in workshops to secondary school students. While Gibberish was found to be an engaging method of teaching coding principles, teaching a full web development workflow including HTML5, CSS, JavaScript, web audio and user interface construction was too broad. Streamlining the NexusUI-Gibberish development process into one language (JavaScript) and one window (Braid in a browser) removed several technical barriers while retaining the same flexible product: a web audio instrument playable on mobile devices.

[6]http://openendedgroup.com/field/

3. DESIGNING BRAID INSTRUMENTS

NexusUI widgets are added to an instrument through drag-and-drop. Widgets can be resized, styled, renamed, or deleted after being added to the page. The *nx.add(type)* function handles adding widgets to the page and the *Widget.destroy()* function erases them and automatically removes any event handlers for the object (including accelerometer or animation events). Widgets may be highlighted, moved and deleted in groups. When an interface layout is complete, an edit-mode, performance-mode paradigm similar to Max or Pure Data lets users toggle between an editable interface and a performable interface.

3.1 Global and Local Code Blocks

While much audio code in Braid is executed by interface widgets, some code may not be associated with a widget, or may be desired to be executed only once. A division of web audio code into two types of code arises from the following use case of a simple theramin-like sine oscillator.

```
var a = new Sine({frequency: 440})
a.frequency = newValue;
```

The first line of code—a definition of a sine oscillator using Gibber.Lib—should be executed once to initialize the instrument. The second line of code may be executed each time a frequency change is desired, and would most likely be responding to an interface widget in Braid. To accomplish the attachment of certain web audio code to interface widgets, while maintaining the ability to write global audio code which initializes an instrument, Braid separates web audio code into two types: global code and local callback code.

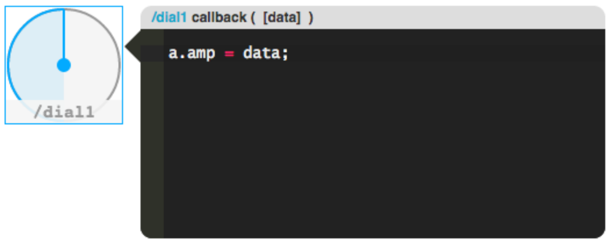
- **Local Callback** code is attached to a specific widget and executed each time that widget is interacted with. Local code might involve setting the frequency or amplitude of an existing oscillator.

- **Global** code is executed once when an instrument is loaded or when an instrument's audio is reset manually by the user. This may contain code such as initializing oscillators, samples, or audio effects.

We find these two genres of code to require different interfaces within Braid.

3.1.1 Local Callback Editor

Each widget provides access to a code window for audio callback code specific to the widget. This code editor appears when an object receives focus, and disappears when the widget is blurred (no longer selected). Code written into this editor executes each time the widget's values change, generally after touch, move, release, or animation events. Values of the widget's current state are accessible in the function via the *data* argument, which is an object with properties specific to the widget. For example, the *data* object of a 2D position slider has two properties: data.x and data.y. If a widget has only one value, such as the dial widget's value, *data* will be a number equal to that value. Figure 3 illustrates a dial's callback editor.



```
a.amp = data;
```

Figure 3: Pup-up Callback Editor

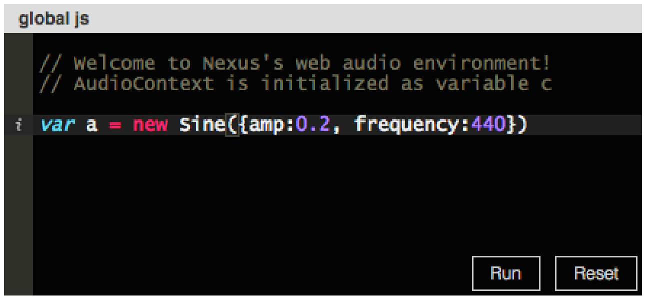
3.1.2 Global Audio Code Editor

A larger code window is always accessible and can contain global code. *Run* and *Reset* buttons let the user start and stop web audio code after making edits. Global code is also automatically run when loading a saved instrument in performance mode (see *Recalling Instruments*), so that instruments can be immediately performed with when launched. Braid's ability to restart audio code is made possibly with Gibber.Lib's *Gibber.clear* function, which erases all Gibber audio processes that have been added to its scriptprocessingnode. Currently, Braid does not erase or reset code written with the Web Audio API or JavaScript that does not envoke Gibber. Resetting and running non-Gibber global instrument code is therefore a significant barrier to a smooth instrument editing workflow, an issue discussed in the future directions of this paper.

3.1.3 Interface Example

Returning to the original scenario of a simple theramin-like instrument, the code written into the omnipresent global code editor would be:

```
var a = new Sine({frequency: 440})
```



```
// Welcome to Nexus's web audio environment!
// AudioContext is initialized as variable c

var a = new Sine({amp:0.2, frequency:440})
```

Figure 4: Global Audio Editor

This would initialize a sine oscillator at 440 Hz when the global code Run button is pressed. A callback function tied to a dial, which would control this sine tone, might be:

```
a.frequency = data * 1000;
```

The slider could then control the frequency of the sine tone between 0 Hz and 1000 Hz.

3.2 Design on Desktop, Perform on Mobile

Braid encourages a desktop-to-mobile workflow in which users design instruments on desktop computers with adequate screen size, then load those instruments onto mobile devices for performances that take advantage of a mobile device's accelerometer, touchscreen, and mobility. Instruments can be loaded onto a mobile device using a custom URL or QR code (see *Recalling Instruments*). To assist with the development of instruments for mobile devices, dashed lines in Braid outline the pixel dimensions of a few average mobile phone and tablet sizes. These outlines can be toggled on or off.

3.3 Additional Performance Opportunities

Beyond designing static audio interfaces consisting of buttons and sliders, Braid offers a few additional interface opportunities.

Automation Tools.

As a computer music performer, controlling generative music processes is sometimes preferable to controlling individual widgets by hand. One drawback of bundling audio code within individual touch-widgets is that a performer is limited by how many fingers they can fit on the touchscreen. To make algorithmic composition in Braid more available, we have created tools for automating and animating individual widgets. The **remix** widget records other NexusUI widget values into a wavetable, and allows for looping, stretching, and granular playback of the wavetable, letting users sample and reuse widget gestures. In addition, several widgets, including dial, position, and metro can use animation for generative music.

Live Coding Interfaces.

Braid raises new strategies for live coding audio interfaces through a combination of drag-and-drop and Gibber code editing. For example, it is possible to add a new widget and write its callback function mid-performance. However, Braid is intended primarily as a web audio instrument design platform, focusing on building multitouch instruments for mobile devices which can be saved and recalled for later performance. Therefore, Braid does not currently take advantage of Gibber's live coding features and does not provide methods for initializing new global code without restarting all audio.

Graphics.

A final, unexplored territory of Braid is its capacity to control HTML5 <canvas> graphics using widget callback functions. Each NexusUI widget is drawn on a canvas, so any widget's canvas may be used as a drawing surface. Braid also includes a special **surface** widget which is blank canvas for drawing. The following code could be added into a **position** widget's callback code editor in order to draw onto a surface.

```
with( surface1.context ) {
   fillRect( data.x, data.y, 5, 5);
}
```

4. DISTRIBUTING BRAID INSTRUMENTS

In "Rapid Creation and Publication of Digital Musical Instruments" [17] Roberts et al describe the archiving and distribution of web audio instruments via a central instrument database. We find this to be a useful model for sharing web audio instruments, and have implemented a database for storing and recalling Braid instruments.

4.1 Encapsulating Interfaces as JSON

As all of the NexusUI widgets within Braid are managed by JavaScript object instances, and each widget's callback code is stored as a property of that instance, the most natural export format for a Braid user interface is Javascript Object Notation (JSON). Essential details of each widget such as name, type, layout data, callback function, and widget settings are stored as properties of a descriptor object, and those widget descriptors are pushed into an array of widget descriptors. This array of widget elements is stored along with global audio code and global settings parameters for the interface, and includes all details needed to reconstruct the instrument.

```
{
   "globalAudio":"//Global js\n\ntoggle1.val = 1;",
   "elements":[
      {
         "type":"toggle",
         "canvasID":"toggle1",
         "width":50,
         "height":50,
         "top":45,
         "left":95,
         "audioString":"console.log(data);",
         "settings":""
      }
   ],
   "settings":{
      "colors":{
         "accent":"#0af",
         "fill":"#f5f5f5",
         "border":"#999",
         "black":"#000",
         "white":"#FFF"
      },
      "version": {
         "nexusui": "1.0.0"
         "gibber": "1.1.0"
      }
   }
}
```

4.2 Sharing Instruments by Database

Braid users share their interfaces to a public repository of instruments. Clicking on the disk icon in Braid opens a dialog to name the interface. Once named, the interface is stored on a public server. Any subsequent saving of the interface requires the acceptance of a warning dialog before overwriting the interface.

Currently, all Braid interfaces are open to the public. Users may load, edit, and resave any interface in the database. The openness of this approach leads to a dynamic mashup-style folk tradition of web audio instruments. However, we recognize the instability of interfaces within this type of system and are considering methods to set certain interfaces as final or non-editable before performances.

4.3 Recalling Instruments

Instruments saved to Braid may be recalled by several methods.

By Search.

A basic search bar allows users to easily search and load instruments previously built in Braid. The search bar displays a list of instruments through an autocompletion algorithm, encouraging users to explore other instruments. A special search page for mobile devices lets mobile users search and load instruments without navigating the main Braid desktop page.

By URL.

For quick access on mobile devices, in performance, or to share your instrument through social media, each interface can be loaded by URL, appending #*uiname* where *uiname* is replaced by the name of the instrument. If loaded by URL, an instrument automatically opens in performance mode and executes its global audio.

By QR Code.

An additional tool for mass distribution of Braid instruments is QR code, which can be generated at any time by a hyperlink within the Braid editor.

5. CONCLUSIONS

By offering a new approach to web audio instrument creation and editing within a MVC paradigm, and by encouraging easy access and distribution of these instruments, Braid hopes to contribute to the dialogue surrounding modular instrument development in web browsers. Braid's approach has benefits; it offers intuitive UI building, code editing, and instrument management, and streamlines the workflow of building web audio instruments with the NexusUI toolkit. However, Braid also has drawbacks, including difficulties re-initializing global code and restarting instruments, dividing web audio code between two different code editors, and currently supporting only one web audio library, Gibber.Lib.

This approach for building and managing web audio instruments incites further work in this field which can build upon Braid's ideas.

5.1 Future Directions

Braid is a young application and offers many further possibilities that could broaden its usability for creating and distributing web instruments. Braid's aforementioned drawbacks may be addressed in the future through strategies including compartmentally managing global audio code blocks, running instruments in new browser windows, and supporting other web audio libraries. Future possibilities with Braid also include sharing modular instrument components, and collaborative instrument building through websockets.

Sharing Modular Audio Components.

In addition to publishing whole instruments, one future goal is to allow users to publish interface modules that can be included in other instruments, rather than only publishing entire instruments. This might allow a user to publish a button that makes a specific sound, which can then be added by other users to their own instruments. Sophisticated processing units, synthesizers, and fully functional instruments could be published for recombination. A mashup culture or folk tradition would be further enabled with such a usage paradigm.

A significant difficulty in modularization is compartmentalizing global audio code. The global code of the module must be integrated with the global code of any instrument that it is added to. This will cause problems if variable names overlap with other instruments or when adding two or more of the same module to an instrument. Also, depending on how the modules interact, the order that they are instantiated may cause conflicts. This direction may require a significantly more involved management of code blocks by Braid.

Launching Instruments in Multiple Browser Windows.

Braid affords the possibility of launching multiple independent instruments in unique browser windows when performing on desktop machines. This method would have advantages in keeping the global code of each instrument separate. However, the performer would need to focus a browser window before interacting with it, leading to a two-click process for the first interaction on each window. In addition, these windows would need to be made aware of each other in order to use mouse or key event listeners in multiple windows at once. However, interfaces that use heavy automation, rather than heavy interaction, could act in multiple browsers at once without causing frequent interaction issues.

Database and Distribution.

The database storage and distribution model has much room to grow. Although the simple naming and JSON storage paradigm has been useful for prototyping, a number of other fields could be used to assist in making the database more functional. Possibilities include:

- recording the popularity of instrument through ratings or usage
- associating a specific user with an interface, including permissions or password protection which would be a useful addition for keeping performance instruments in a stable state
- sorting by the platform intended, specific objects used, or audio processes used
- storage and recollection of presets within each Braid UI

Web Audio Frameworks.

As a number of promising web audio frameworks and processing units are in development, integrating support for Tone.js [13], Flocking.js [8], and others could be pursued. Braid could also explore the ability to load JavaScript files dynamically in order to load libraries. Integration of other audio APIs like Soundcloud or Freesound would open doors for further work with recorded audio.

Collaborative Interface Building.

Finally, collaborative interface building could open a vast territory of live interface coding possibilities for education and ensemble work. If Braid is moved to a websocket-enabled node.js server, groups could create and code interfaces as an ensemble. In this event, each user would see the entire NexusUI interface, but would open their own callback editors so that different users could edit unique callbacks. This type of collaborative, live interaction with browser-based intstrument design can and should be explored, as software design moves to the browser and takes advantage of what the network has to offer us.

6. ACKNOWLEDGMENTS

The authors would like to acknowledge the support of the Louisiana State University Center for Computation & Technology, Cultural Computing focus area, and the School of Music. We would also like to acknowledge the support of the Board of Regents for Mobile Initiatives.

7. REFERENCES

[1] Nexus User Interface - http://nexusosc.com.

[2] Braid, http://nexus.cct.lsu.edu/braid, October 2014.

[3] J. Allison, Y. Oh, and B. Taylor. Nexus: Collaborative performance for the masses, handling instrument interface distribution through the web. In *Proceedings of the New Interfaces for Musical Expression conference*, 2013.

[4] N. J. Bryan, J. Herrera, J. Oh, and G. Wang. Momu: A mobile music toolkit. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME), Sydney, Australia*, 2010.

[5] P. L. Burk. Jammin' on the web - a new client/server architecture for multi-user musical performance. In *ICMC 2000 Conference Proceedings*. International Computer Music Conference, 2000.

[6] R. Canning. Realtime web technologies in the networked performance environment. In *Proceedings of the 2012 International Computer Music Conference*, 2012.

[7] J. K.-M. Charlie Roberts. Gibber: Live coding audio in the browser. In *Proceedings of the 2012 International Computer Music Conference*, 2012.

[8] C. Clark and A. Tindale. Flocking: a framework for declarative music-making on the web. In *Proceedings of the 2014 International Computer Music Conference*, 2014.

[9] A. D. Diana Young. Bowstroke database: A web-accessible archive of violin bowing data. In *Proceedings of the 7th Annual Conference on New Interfaces for Musical Expression*, 2007.

[10] G. Essl and A. Müller. Designing mobile musical instruments and environments with urmus. In *New Interfaces for Musical Expression*, pages 76–81, 2010.

[11] M. Gurevich. Jamspace: a networked real-time collaborative music environment. In *CHI '06 extended abstracts on Human factors in computing systems*, CHI EA '06, pages 821–826, New York, NY, USA, 2006. ACM.

[12] T. Lossius, T. de la Hogue, P. Baltazar, T. Place, N. Wolek, and J. Rabin. Model-view-controller separation in max using jamoma. In *Proceedings of the International Computer Music Conference (ICMC-SMC)*, 2014.

[13] Y. Mann. Tone.js https://github.com/tonenotone/tone.js/.

[14] C. McKinney. Quick live coding collaboration in the web browser. In *Proceedings of the 14th Annual Conference in New Interfaces for Musical Expression (NIME)*, 2014.

[15] T. Reenskaug. Models - views - controllers. Technical report, Xerox Parc, 1979.

[16] C. Roberts, G. Wakefield, and M. Wright. The web browser as synthesizer and interface. In *Proceedings of the New Interfaces for Musical Expression conference*, 2013.

[17] C. Roberts, M. Wright, J. Kuchera-Morin, and T. Höllerer. Rapid creation and publication of digital musical instruments. In *Proceedings of the 14th Annual Conference in New Interfaces for Musical Expression (NIME)*, 2014.

[18] A. Tanaka. Mobile music making. In *Proceedings of the 2004 conference on New interfaces for musical expression*, NIME '04, pages 154–156, Singapore, Singapore, 2004. National University of Singapore.

[19] B. Taylor, J. Allison, Y. Oh, D. Holmes, and W. Conlin. Simplified expressive mobile development with nexusui, nexusup, and nexusdrop. In *Proceedings of the New Interfaces for Musical Expression conference*, 2014.

[20] F. J.-G. S. Weitzner, N. and Y. Chen. massmobile – an audience participation framework. In *Proceedings of the New Interfaces for Musical Expression Conference*, 2012.

[21] C. Wilson, P. Adenot, and C. Rogers. Web audio api, http://webaudio.github.io/web-audio-api/.

[22] M. Wright. Open sound control-a new protocol for communicationg with sound synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, pages 101–104, 1997.

[23] J. Young. Using the web for live interactive music. In *Proceedings of the 2001 International Computer Music Conference*, 2001.