Gibber: Abstractions for Creative Multimedia Programming

Charles Roberts, Matthew Wright, JoAnn Kuchera-Morin, Tobias Höllerer
Media Arts & Technology Program, University of California at Santa Barbara
charlie@charlie-roberts.com, {matt,jkm}@create.ucsb.edu, holl@cs.ucsb.edu

ABSTRACT

We describe design decisions informing the development of *Gibber*, an audiovisual programming environment for the browser. Our design comprises a consistent notation across modalities in addition to high-level abstractions affording intuitive declarations of multimodal mappings, unified timing constructs, and rapid, iterative reinvocations of constructors while preserving the state of audio and visual graphs.

We discuss the features of our environment and the abstractions that enable them. We close by describing use cases, including live audiovisual performances and computer science education.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Interactive Environments; J.5 [**Arts and Humanities**]: Performing Arts

General Terms

Design, Human Factors

Keywords

Multimodal Programming; Live coding; Audio; Graphics; Web; JavaScript; Creative Coding

1. INTRODUCTION

Creative coding environments have often favored one modality at the expense of others. Popular audio programming environments like ChucK [33] and SuperCollider [14] come with minimal or no graphics capabilities out of the box, while the example audio code for Processing [22], one of the most popular environments for teaching programming to visual artists, is buried in nested submenus. In some cases, the design of a domain-specific language (DSL) specific to one modality potentially affects its ability to be adapted to others. The syntax of the ChucK language, for example, contains abstractions affording terse graph formation, but in a way that is arguably specific to audio programming. Researchers have argued in favor of using DSLs for teaching creative coding [18, 33], as they enable abstractions affording creative expressivity to be encoded within language syntax. However, we argue that multimodal content creation often requires the use of general-purpose programming languages to accommodate the needs of differing representations. General-purpose languages for creative coding also provide an opportunity to use and teach syntax and other elements that, assuming a given language has wide adoption, transfer to other environments.

We seek to bring the expressivity often found in domain-specific languages for creative coding to a general-purpose programming language through the use of high-level abstractions. We address this challenge in the design of an audiovisual, creative coding environment for the browser, *Gibber* [25]. In this paper, we begin by discussing the motivation and philosophy behind Gibber and other creative coding environments that inform Gibber's design. We then provide an overview of Gibber's multimedia affordances and examine the notation used by Gibber end-users, as the notation is intricately tied to the abstractions that are a focus of our research. After discussing three abstractions for creative coding that we have found particularly rewarding, we conclude with a description of selected audiovisual performances that were realized with Gibber and discuss end-user feedback gathered about the environment.

2. MOTIVATION AND AUDIENCE

Gibber began its development as an environment for *live coding* performance, a practice discussed further in Section 3. As its development continued, the educational potential of a browser-based programming environment became appealing to us, and we began to target both performers and beginning programmers as our audience. In particular, Gibber is appropriate for people interested in learning JavaScript who also have interest in audiovisual programming. The youngest group of users we know of to program with Gibber started at age eleven; it has also been used to teach at the university level. With this audience in mind, there were four principles that motivated Gibber's development:

Capitalize on Browser Features and Ubiquity

Due to its ubiquity and recent additions to its feature set, the web browser provides an accessible vehicle for disseminating audiovisual works and the tools to create them. Gibber takes advantage of this ubiquity, providing a programming environment usable by anyone running a modern web browser.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MM'14, November 03 - 07 2014, Orlando, FL, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3063-3/14/11 $15.00.
http://dx.doi.org/10.1145/2647868.2654949.

Figure 1: Three screenshots of generative shaders running in Gibber. The first two images show graphical output while the final image also depicts overlaid JavaScript and GLSL editors, which can be hidden and revealed via keystroke macros.

No installation of software is required, removing an immediate barrier to entry: anyone with the URL of Gibber's homepage can immediately begin using it. The browser offers a variety of affordances for creative practice, including 3D graphics, realtime audio synthesis, realtime audio and video input, networked communications, and support for touch, mouse, and keyboard interactions; Gibber takes advantage of all of these. Gibber also encourages social aspects of content production in the browser by providing a centralized database for publishing and browsing audiovisual projects and a variety of tools for realtime collaboration.

Use JavaScript, not a Domain-Specific Language
JavaScript was selected as the end-user programming language in Gibber due to its first-class status in the browser as well as its use as a scripting language in commercial, multimedia applications such as Max/MSP[1], Apple's Logic Pro[2], various Adobe applications including Photoshop, Illustrator, and Flash[3], and game engines such as Unity[4] and DX Studio[5]. JavaScript's dynamic language features and its ability to meld functional and object-oriented programming techniques make it particularly well suited to exploratory creative practice. It has also been identified as an excellent language for introducing computer science students to programming [23].

One important constraint when designing Gibber was to avoid requiring pre-processing of end-user code to make it valid for execution by the JavaScript runtime. That is, all code written by Gibber end-users must be valid JavaScript. By avoiding application-specific language extensions, we increase the likelihood that syntax and other lessons learned while programming in Gibber will transfer to other environments with a JavaScript API. In contrast, some of the knowledge gained while using domain-specific languages (at minimum, a knowledge of syntax) is inevitably lost when users attempt to transfer it to domains employing different languages. Our concern regarding knowledge transfer also drove our decision to use a textual programming language rather than a visual one, as visual programming languages are often application specific.

Support Audio, Visual and Interactive Programming
Gibber emphasizes equal treatment of audio, visual, and interactive modalities, and was designed so that users can easily connect inputs and outputs across modalities without requiring special code considerations. The API for each modality is designed with a low threshold for beginning programmers and a high ceiling for users with more experience. For example, the graphics subsystem affords placing a 3D geometry on the screen with a single function call (example: `Cube()`) but also enables advanced users to live code vertex and fragment shaders. Similarly, the audio subsystem affords the creation and playback of a drum loop in a single function call (example: `Drums('xoxo')`) but also enables advanced users to perform audio-rate modulation of sample-accurate scheduling.

Provide an Expressive Notation
Given the motivations and constraints listed above, one resulting challenge was to design a consistent, expressive notation that lends itself to creative practice; the result is discussed in Section 5 and Section 6.

3. RELATED WORK

We limit our discussion here to representative *creative coding environments* that, in contrast to visual programming environments, use text editing as the primary coding mechanism. We use the term *creative coding*, widely used colloquially, to discuss programming practice geared towards artistic content production; it has also been found effective in computer science education [9]. The software used for creative coding often includes an integrated development environment, libraries for generating audiovisual output, and, in many cases, a domain-specific language designed to ease creative exploration.

One of the first and most successful languages for creative coding was *Logo* [20]; at current count, over two-hundred and fifty Logo implementations have been authored [5]. Originally created as a Lisp dialect to control the movements of a drawing robot, Logo became famous in the 1980s after it was altered to create graphics on computer monitors and subsequently taught around the world. Research on Logo and creative coding flourished at MIT, where Seymour Papert, one of the original creators of Logo, was a professor. During the 1990s the creative coding environment *Design By Numbers* (DBN) [11] was developed by John Maeda and his Aesthetics and Computation Group at MIT. DBN moved away from the concept of the turtle to provide a new, domain-specific language for creating non-interactive visual artworks. Two of Maeda's students, Casey Reas and Ben Fry, drew inspiration from using DBN to teach and designed their own creative coding environment, *Processing*, which is now widely used and taught. Processing sketches are written in Java;

[1] http://cycling74.com/products/max/
[2] http://www.apple.com/logic-pro/
[3] https://www.adobe.com/creativecloud.html
[4] http://unity3d.com
[5] http://www.dxstudio.com/

as a result its output is fairly cross-platform and can be embedded in web pages as Java applets. Java applets do not, however, run in most mobile operating systems. Developer John Resig solved this problem by porting the main Processing classes to JavaScript in a version called *Processing.js* [24]. In a clever twist, he also wrote a pre-processor that takes Processing programs written in Java as input and compiles them to JavaScript. This enables many of the demos and examples available in the Processing ecosystem to be compiled and subsequently run in the browser without any modification to the original Java source code. However, many important library extensions for Processing have not been ported to Processing.js; as one example, the audio functionality provided by the *minim* [19] library is not available at the time of this writing. Work has begun on *p5.js*[6], a port of Processing that, unlike Processing.js, uses JavaScript as the end-user programming language.

The audio community has also been active in the development of creative coding environments; much of this activity stems from the performance practice known as *live coding*, where works of audiovisual art are programmed in front of audiences while the programmer-performer's code is projected for audience members to see and potentially follow [6, 13]. Live coding was one of the motivations for the first iteration of Gibber, and Gibber has since drawn inspiration from a number of environments and languages that enable such performances. *SuperCollider* [14] is one popular audio programming environment with a language based on ideas from Smalltalk and functional programming. The Supercollider language is coupled to a flexible sound synthesis server that serves as the basis for a number of live coding environments, most notably *ixi lang* [12] and *Overtone* [4]. The proxy system described in Section 6.3 was heavily inspired by techniques pioneered in SuperCollider. ChucK [33], another popular live coding system for audio, provides a domain-specific language catered towards audio graph construction, concurrency, and sample-accurate scheduling.

None of these live coding environments provide significant built-in graphics capabilities, though SuperCollider's basic 2D drawing API can be extended with OpenGL functionality via the SCGraph library [2] and Overtone is designed to interoperate with a number of graphics applications [3, 1]. LuaAV [32] and Extempore [30] are creative coding platforms that include both audio and visual affordances. LuaAV uses the Lua general-purpose language and Extempore a uses a combination of Scheme and the custom language *xtlang*. Although Gibber is similar in that it uses a general-purpose language (in this case JavaScript), we argue that our research enables end-users to create audiovisual works at a higher level of abstraction, easing the burdens of multimodal programming for beginning creative coders. The abstractions also afford a terseness to Gibber programming that makes it well-suited for live coding performances, where language verbosity is one factor that directly impacts the potential for virtuosity. Almost the entirety of the Gibber environment, from the end-user language to the server to the 2D and 3D graphics APIs, is programmed in JavaScript[7]; this means that users can modify and augment any part of the environment within the constraints of the browser.

[6]http://p5js.org/

[7]The one exception to this is fragment and vertex shaders, which are programmed in GLSL.

Alex McLean's *Tidal* language for live coding pattern represents a middle road between general-purpose and domain-specific languages [17]. Tidal is a DSL that is used within Haskell to define patterns controlling sound synthesis. When Tidal syntax is parsed each token becomes a Haskell function. Although it is possible to instead create similar patterns solely using Haskell, McLean believes the extra verbosity required would not lend itself to live coding performance practice [16]. Inspired by the sequencing capabilities of Tidal, we are currently exploring strategies using JavaScript to define and manipulate patterns[8].

Fluxus [10] is a live coding environment by David Griffiths that uses Scheme to define and manipulate 3D scenes made with OpenGL. It also comes with an extension named Fluxa that provides basic audio synthesis capabilities. In comparison with Gibber, Fluxus has impressively superior 3D graphics functionality, but does not include 2D graphics or shader programming capabilities.

Other live coding environments run in the browser, however, most use DSLs instead of JavaScript for their end-user programming language. One such example is *Livecodelab* [7], which primarily provides capabilities for controlling and sequencing 3D graphics and shaders. The authors of Livecodelab began with JavaScript as the end-user language, moving to CoffeeScript (an expressive language that compiles to JavaScript) and finally to developing their own custom DSL. Thus, instead of seeking greater expressivity through abstraction, Livecodelab's developers focused on designing a language customized to their needs. Like Gibber, *Lich.js* [15] provides both audio and graphics programming in the browser, and is notable for its flexibility in creating and customizing musical patterns. Similar to the authors of Livecodelab, the author of Lich.js designed a Haskell-inspired DSL as its end-user programming language, creating a compiler that converts the end-user code to JavaScript for execution.

4. AN OVERVIEW OF GIBBER

Gibber is a creative coding environment that runs in most modern web browsers, including Chrome, Safari, and Firefox. Gibber programs (aka *giblets*) are created and edited at runtime. Users enter code and dynamically evaluate code fragments with simple keystroke macros. Execution of selected code can optionally be delayed until the beginning of the next musical measure to ensure that musical sequences created at different moments in time are rhythmically synchronized with sample-accurate precision.

The graphics engine provides 2D and 3D APIs and also a variety of pre-built generative and post-processing shaders. The post-processing shaders work in both 2D and 3D giblets; we accomplish this in 2D by drawing to a canvas which is subsequently used to texture a full-screen OpenGL quad. The ease of combining 2D drawing with post-processing shaders is relatively unique among creative coding environments; these effects can add visual complexity and interest to simple 2D forms that beginning programmers may produce. Gibber also provides the capability to live-code fragment and vertex shaders. Its editing window can be divided into multiple columns; in a live shader editing session there is typically one column of JavaScript and one or two columns

[8]For a demo of JavaScript patterns in Gibber see: http://gibber.mat.ucsb.edu/?p=charlie/pattern_demo

of GLSL. Shaders can be edited and recompiled with the same keystroke macros used to execute JavaScript; Gibber notes the column that is currently focused when keystroke events are triggered and executes the macro appropriate to the language being edited. Abstractions are also provided to easily pass information from the JavaScript runtime to the GLSL environment as uniforms; a new uniform can be defined and mapped in a single line of JavaScript. Fig. 1 shows multi-column shader programming.

Gibber's audio capabilities include low-level elements such as oscillators and filters, as well as higher-level, aggregate instruments designed to immediately provide an interesting sonic palette for musical performance and composition. These instruments include an emulation of a classic analog drum machine, a two-operator FM synthesizer, a three-oscillator monosynth, a sampler, and many other options. A variety of audio effects are also included along with flexible mechanisms for routing and defining audio graphs.

Graphics and audio can be synchronized with relative ease in Gibber. Scheduling of events is sample-accurate and can be performed using various measures of time including audio samples, milliseconds, seconds, or musical beats. Gibber's support for live coding dance music includes a visual metronome in the upper hand corner that defaults to 4/4 time but can be set to other meters or turned off entirely.

Gibber provides interactive control of audiovisual objects by generating control signals from the cursor position in the browser window or from keystroke events. A companion GUI library, *Interface.js*, is included in Gibber to support the creation of more elaborate user interfaces [26, 27].

Maximizing the social potential of content creation in the browser is an important consideration in Gibber's design. Users can publish giblets to a centralized database and disseminate associated URLs to friends and colleagues who might want to view their work. When users publish giblets they can tag them with metadata and provide a description; a browser is provided that allows users to easily search the database and view each giblet's associated metadata. Gibber's built-in chat system enables users to easily ask questions of one another and is extended with a simple mechanism for entering into collaborative editing sessions. Clicking on another user's name in the chat window and selecting a code fragment to share is all that is required to begin a joint programming session, or, potentially, a joint networked performance.

5. NOTATION DESIGN

We designed Gibber's notation to provide a consistent end-user programming experience across modalities; it is an integral part of the abstractions described in this paper. It contains some debatable design decisions; each was considered carefully in the context of creative coding and we believe the results well serve the needs of creative coding community. Important aspects of the notation include:

- *Abuse of Global Namespace* - The majority of constructors in Gibber are typically invoked in the global namespace. This makes calls to constructors extremely terse and simple, and avoids any required initial discussion of namespaces with beginning programmers. Gibber includes a simple module system for publishing and using JavaScript extensions to its environment. As users advance in skill and want to customize their programming experience they can learn about and employ namespaces in the design of their own extensions.

Figure 2: 2D animation in Gibber. On each frame of video, a polygon is drawn with a random color while previous output is faded by ten percent.

- *Constructor Syntax* - Gibber provides three different methods of passing arguments to object constructors. The first is an ordered list of comma-separated arguments, in a notation similar to C++ or Java. The second is an unordered dictionary of key/value pairs to assign to the generated object as properties. In contrast to a comma-separated list this shows exactly which properties are being set on the new object, which is useful for programmers, students, and potentially audience members. The third method is to pass a string naming a constructor *preset* that contains a set of predefined property values, along with an optional dictionary to selectively override aspects of the preset.

- *Capitalization* - The name of any function that returns an object, whether or not it would traditionally be referred to as a constructor, is capitalized. This allows for disambiguation of names that might return different values. For example, `rndf()` returns a single random number, whereas `Rndf()` (useful for sequencing) returns a function that in turn returns a random number. Capitalizing property names of Gibber objects triggers the mapping abstractions discussed in Sec 6.2 by implicitly creating an object that handles the complexities of multi-rate signal mapping.

- *Cascading method calls* - Many calls to object methods in Gibber return the object itself. This allows commands to easily be chained together. In practice, this is often used after calling constructors to customize the resulting object with effects or temporal sequencing.

A short example of the resulting notation, in which the rotation of 3D object is tied the output envelope of a distorted synth line, is given below:

```
mySynth = Synth({ attack:ms(1), decay:ms(200) })
  .note.seq( ['c4','g4'], 1/4 )
  .fx.add( Distortion() )

myCube = Cube()
myCube.rotation.x = mySynth.Out
```

6. ABSTRACTIONS FOR MULTIMEDIA PROGRAMMING

Gibber provides several programming abstractions that simplify exploratory programming practice and multimodal authoring.

6.1 Time, Scheduling, and Rhythm

6.1.1 Rhythmic Notation

Initially, all time values in Gibber were measured in audio samples. To indicate a rhythmic duration, variables preceded by underscores were used that stored the number of samples for particular metric subdivisions. For example, `_4` equaled the number of samples in a quarter note, `_16` equaled the number of samples in a sixteenth note etc. This was reasonably terse except in situations where long durations needed to be specified; for example, ten measures would be notated as the expression `_1 * 10` .

We were subsequently inspired by considering more traditional rhythmic notations, such as using `1/4` to represent a quarter note, `1/8` to represent an eighth note and `1` to represent a whole note. Using this notation the value `10` represents the duration of ten measures of music in 4/4 time. However, problems arise when subsequently considering how to represent temporal values that are not expressed in terms of rhythmic meter. As one example, suppose a user wants to indicate the attack of an amplitude envelope in samples? A very short attack might have a duration of fifty samples, but to define this duration Gibber requires a mechanism to differentiate between fifty samples and fifty measures. Accordingly, we declare a variable, `Gibber.maxMeasures`, that denotes the maximum number of measures that can be indicated as a duration. The default value for this variable is forty-four; this means that any higher duration value is considered to be measured in samples while any lower value is measured in terms of meter. Users can also notate time using the `ms()`, `seconds()` or `measures()` functions, which translate their arguments into samples.

6.1.2 Scheduling

Most scheduling is performed Gibber's audio callback with one exception that will be discussed momentarily. This affords audio-rate modulation of scheduling, a rare feature even in dedicated music authoring environments. It also enables experimental generative techniques where unit generators can be created and added to the audio graph (and subsequently removed if desired) within the bounds of a single execution of the audio callback; another feature that is rare in musical programming environments. The price for this flexibility is the potential for blocking, a problem that is exacerbated by the (predominantly) single-threaded nature of the JavaScript runtime. We have mitigated this problem via the creation of a highly-optimized JavaScript audio library, *Gibberish.js*, discussed in [26].

There are three main ways that events are scheduled in Gibber. The first is through calls to the `future` function, which accepts an arbitrary function and a time duration (measured in samples) to wait before executing it. It is similar to the standard `setTimeout` method included in most JavaScript runtimes, but the `future` method has better temporal accuracy through its reliance on the audio clock. Providing a functional approach to sample-accurate scheduling also affords a live coding idiom known as *temporal recursion* [31], where a function generating audiovisual output is repeatedly executed at various moments of time and iterated to gain complexity.

The second scheduling method is the use of `Seq` (sequencer) objects in Gibber. `Seq` objects are flexible vehicles for changing properties and calling methods on objects at defined temporal intervals; they can also be used to execute anonymous functions. The `Seq` object has a `durations` property that determines when it triggers actions and can either loop its actions indefinitely or perform them a pre-defined number of times. One unique abstraction in Gibber is a shorthand wherein every property and method of Gibber's standard library has its own `seq()` method, allowing properties to be sequenced in an extremely terse manner, as seen when sequencing the `frequency` and `amp` properties in the code example at the end of this section.

The final method is tied to Gibber's graphics engine as opposed to the audio clock. Any graphical object can be assigned an `onupdate()` event handler that is called for every frame of video presented. This enables a more traditional model of graphics scheduling as found in creative coding environments like Processing, while also ensuring that elements such as shader uniforms are only updated at visually salient moments in time.

The code example below illustrates the three methods of scheduling discussed in this section.

```
// use the sequencer included in each property to
// schedule changes in frequency and amplitude
a = Sine()
  .frequency.seq( [440,880], 1/4 )
  .amp.seq( [.5, .25, .1, 0], 1/8 )

// use future to schedule a function call
future( function() { a.fadeOut( 1 ) }, ms(2000) )

// use the onupdate method to increment a property
// in each frame of rendering
b = Cube()
b.onupdate = function() {
  b.rotation += .01
}
```

6.2 Multimodal Mappings

Creating mappings between modalities is complex enough to deter creative coders from regular experimentation. It involves the application of filters and envelope followers (or raw sample-rate conversion), as well as various affine transformations. The goal of Gibber's mapping abstractions is to enable people to think of audiovisual and interactive objects in Gibber the same way audio synthesists think of patching in a modular synthesis environment, where all signals are normalized to a standard control voltage; the result is that anything can freely be patched (mapped) from one object to any other.

Initially presented in the context of designing musical instruments [27], here we expand our discussion of Gibber's mapping abstractions to include mappings between audio and visual modalities. We have created a system and notation that enables, as one example, the rotation of a cube to be continuously controlled by the frequency envelope of a synthesizer using a single line of code:

```
cube.rotation = synth.Frequency
```

The capitalization of Frequency in `synth.Frequency` denotes that we should assign a mapping object providing a continuous translation of the synthesizer's frequency to the rotation of the cube. If `Frequency` were lowercase the instantaneous value of the synth's frequency property when the line of code was executed would be mapped to the rotation; the frequency would not be translated to a range of values appropriate to geometric rotations and would most likely yield non-interesting results. In contrast, when a capitalized property name is mapped to another property two actions occur:

- Gibber compares the timescales of the two properties; here the `synth` is an audio object (operating at 44.1 kHz) while the `cube` is a graphical object (ideally operating at 60 Hz). To accommodate these differing timescales, Gibber places an envelope follower on the frequency of the synth and modulates the cube's rotation with the envelope follower's output. If the mapping was reversed, Gibber would place a low-pass filter on the cube's rotation and assign the result to control the synth's frequency.
- Gibber recognizes that the two properties have different ranges of possible values, and attaches an affine transform to the envelope follower to ensure that it outputs values which make sense as rotations. In effect, it converts a value in the range of 50–3500 Hz to 0–2π radians.

Mapping objects can have different output curves associated with them so that properties such as amplitude can be mapped logarithmically (for perceptual accuracy) while rotation is mapped linearly.

In addition to meta-data on operational timescale and perceptual output curve, every object property in Gibber includes a default output range with the goal of providing intelligent default behaviors for cross-modal mappings. With certain properties we choose a narrower default range than what is capable of being perceived. For example, with oscillator frequency it does not make sense to always map between 20 Hz and the Nyquist limit, even though this is arguably the range of frequencies perceivable. Instead, we choose a narrower range of frequency values, 50Hz to 3500 Hz, which are more commonly used musically. These settings are freely customizable, as even mappings using a narrower frequency range will still generate little effect if the pitch of the unit generator consistently remains low, such as for a bass line. In such cases, users may override the default range specified by the mapping object as follows:

```
cube.rotation.y = synth.Frequency
synth.Frequency.min = 40
synth.Frequency.max = 400
```

Similarly, we can constrain the rotation of our cube to a narrower range than the default of (0,2π):

```
cube.rotation.Y.min = 0
cube.rotation.Y.max = Math.PI * .5
```

The properties of mapping objects (such as `min` and `max`) also employ Gibber's mapping abstractions. This enables the creation of complex feedback networks where properties of one object affect another and the amount of the effect is continuously controlled by a property on yet another object.

Figure 3: Two shaders post-process a scene consisting of a spinning icosahedron and a background quad textured with a live video feed, made in nine lines of JavaScript.

In the example below, the rotation of a geometry controls the pitch of a drum loop, while the output envelope of a bass line controls the effect of the geometry's rotation on the drum loop's pitch.

```
tetra = Tetrahedron().spin( .005 )
drums = Drums('xoxo')
drums.pitch = tetra.rotation.X
bass = FM('bass').note.seq( [0,7], 1/8 )
drums.Pitch.max = bass.Out
```

End-users can also sequence mapping properties using the sequencing abstraction discussed in Section 6.1.2. The example below randomly changes the `max` property of a mapping every 250ms and inverts it every two seconds.

```
sphere = Sphere({ scale:Mouse.X })
sphere.Scale.max.seq( Rndf(1,3), ms(250) )
sphere.Scale.invert.seq( null, ms(2000) )
```

The abstraction described in this section is similar to that found in UrMus [8], a multimedia programming toolkit for creating mobile applications. Although there are number of notable differences between the multi-rate mapping strategies adopted by Gibber and UrMus, we believe that the most important distinction is the notation itself. Gibber provides a mechanism for creating multimodal mappings that requires nothing more than the capitalization of a object property name, while UrMus (which provides greater flexibility in how multi-rate mappings are composed) requires multiple lines of code to achieve a similar effect. The goal of our abstraction was to remove the cognitive burdens associated with programming multimodal mappings and we believe our notation accomplishes this; all end users need to remember is to use a capital letter when they wish to make a continuous mapping between two properties.

As mentioned in Section 4, shader uniforms also utilize Gibber's mapping abstractions enabling users to easily incorporate audio signal analysis or geometric object properties into GPU programming.

6.3 Proxies and Maintenance of State

Consider the results of executing the following two lines of code, one after another:

```
a = Sine( 440, .5 )
a = Sine( 880, .5 )
```

An intuitive expectation is that the second sine oscillator, being assigned to the same variable `a` as the first one, would simply replace it in the audio graph; you would be left hearing a single sine oscillator at 880 Hz. This is, in fact, the result of executing these two lines in Gibber thanks to the implementation of *proxy* objects for nodes in the audio and visual graphs. In the above example, the variable `a` represents a proxy object. Whenever an object is assigned to it, the proxy looks at the node it currently holds and removes it from whatever graph it is attached to. The new object then replaces the old object's position in the graph.

In practice, this has a very important effect: the same line or block of code that calls a constructor can be called repeatedly with different argument variations without creating duplicate nodes in the audio or visual graphs. This is much more immediate than manipulating individual properties of objects after a constructor has been called; instead of subsequently typing names and targeted properties one can simply change the arguments to the original constructor call and re-execute it.

In a more complex relationship, sequencing objects that are not themselves stored in proxies are still bound to side effects of their behaviors. Whenever a sequencer is assigned to `target` a particular audiovisual object, that object retains a reference to the sequencer as well. When a proxy is assigned a new object, it checks to see if the object it contains holds any sequencer references. If it does, the proxy object re-routes the sequencers to target the new object stored in the proxy instead of the old one that is removed from the audio or visual graph. This allows users to easily experiment with different sound sources that will all be controlled by the same sequencer. The following block of code shows a sequencer targeting a synth and a variety of commented out potential replacements for the synth.

```
a = Sine( 440, .5 )
// a = Synth({ attack:1/4, decay: 1/4 })
// a = FM({ index:19, cmRatio: 3.011 })
// a = Mono( cutoff:.1, resonance:4 })

b = Seq({
  note:[ 440,660,880,1100 ].random(),
  durations:[ 1/4,1/8,1/2 ].random(),
  target:a
})
```

Thanks to proxies, executing any of the commented-out redefinitions of the variable `a` will change the timbre of the sequenced notes with no interruption in the sequence itself. Not having to create a new sequencer for each iteration or re-execute the constructor of the existing sequencer to pass a new reference makes the process of iteration more immediate and, in our opinion, conceptually simpler. The target of the sequencer is always whatever object is currently held in the global variable `a`, not just the object that was the value of `a` when the sequencer was instantiated.

Proxies are identified in Gibber through the use of variable names that are single, lower-case letters. This convention exists because JavaScript does not possess a language feature known as *catchalls*; that is, there is no way to metaprogram behaviors for any arbitrarily named variable added to an object. The alternative used here is to create a collection of properties on the global object and perform the meta-programming in question only when these properties are assigned to. This requires some initial discussion when introducing Gibber, as the following two lines have completely different effects if they are are executed repeatedly:

```
a = Sine( rndi(400, 800) )
aa = Sine( rndi(400, 800) )
```

In the first line of code the Sine constructor is generating an object assigned to a proxy; therefore any previous objects stored in the proxy will be removed from the audio graph. It can be executed repeatedly, always yielding a single sine oscillator generating sound at any given moment. In contrast, the second line of code does not assign to a proxy object. Every time this line is executed a new node will be placed in the audio graph and the old node that was formerly held in the variable will continue to generate audio but will no longer be referenced in the global namespace. Executing the second line of code ten times will result in ten different sine waves running simultaneously at randomized frequencies.

The mapping abstractions discussed in Section 6.2 are also subject to proxy behaviors. This means that if a proxy object A is replaced, any object B that contained mappings referring to object A will now use the new replacement object for mapping.

The idea of using proxies in audio graphs was initially explored in SuperCollider [28]; here we extend its use to graphics and interactive subsystems while also providing compatibility with Gibber's multi-rate mapping abstractions.

7. USE AND EVALUATION

Our evaluation begins with a brief comparison illustrating how the notation for the various abstractions in this paper compares with the notations of other environments. We then look at use of Gibber in live performance and in the classroom, and end with the results of a feedback survey completed by twenty-eight Gibber users.

7.1 Comparison of Notation

It is easy to cherry-pick comparisons where Gibber's terseness makes another environment look excessively verbose. In one comparison we performed it took seventeen lines of Processing code to mirror a two-line giblet which mapped cursor position to the frequency of a sine oscillator; the frequency of the oscillator then controlled the rotation of a cube. Such comparisons can easily be discredited: for example, although Processing might be substantially more verbose for the above task, it provides functionalities that are not available to Gibber. In order avoid such comparisons, we chose two code examples directly from the documentation of other creative coding environments and attempted to replicate their functionality in Gibber. For more information about differences between Fluxus, Tidal and Gibber, please refer to Section 3.

7.1.1 Audiovisual Mapping in Fluxus and Gibber

Though few environments enable easy mappings between audio and visual modalities, the very first code example in the Fluxus documentation renders a cube whose 3D scaling is modified by the frequency content of an input audio signal, as shown here:

```
(define (render)
  (scale (vector (gh 1) (gh 5) (gh 9) ))
  (draw-cube))
(every-frame (render))
```

In the above example, the `gh` function (short for getHarmonic) retrieves the magnitude of the given bin of a FFT analysis performed on the audio input. Three calls to this function are used to scale the rendering context, after which a cube is drawn to the screen. In Gibber, the same effect is achieved with the following code:

```
cube = Cube()
input = Input()
fft = FFT() // defaults to measuring master output
cube.scale = Vec3( fft.Bin1, fft.Bin5, fft.Bin9 )
```

Here we see an object-oriented approach, where instead of explicitly defining a render function we add an object to the visual graph (this occurs in the constructor), which handles rendering for us. We use identifiers such as `bin1`, `bin2` etc. to identify FFT bin magnitudes; these are capitalized in the example above to use the continuous mapping abstraction discussed in Section 6.2. Although Gibber can mimic the introductory Fluxus example using a code fragment of similar length, it is important to note Fluxus does not generally have the flexibility in mapping that Gibber possesses. For example, there is no capability in Fluxus to tersely map the current frequency of an oscillator to the cube's scale or rotation. Instead, Fluxus users would have to manually track the appropriate variable and explicitly perform the required transformations. In Gibber, making the cube, the oscillator, and creating the continuous mapping can be done in two lines of code:

```
sine = Sine()
cube = Cube({ rotation:sine.Frequency })
```

7.1.2 Sequencing in Tidal and in Gibber

As sequencing and the creation of musical pattern is Tidal's primary motivation, we compare Gibber's sequencing abstraction to Tidal's. For simple cases the two notations closely mirror. For example, the following Tidal code (slightly modified from Tidal's documentation) sequences a simple drum pattern with changes to pan and speed:

```
d1 $ sound "bd sn sn"
    |+| speed ".25 .5 1"
    |+| pan "0 0.5 1"
```

In the example above all code in quotes above uses the Tidal DSL, while the rest is Haskell. The equivalent code in Gibber is the following:

```
Drums()
  .note.seq( [0,1,1], 1/3 )
  .pitch.seq( [.25,.5,1] )
  .pan.seq( [-1,0,1] )
```

As mentioned in Section 3, Tidal offers many sophisticated options for pattern creation and customization that Gibber currently lacks. But for simple cases like the one above, Gibber's notation is of comparable verbosity and, in our opinion, comparable clarity to that of Tidal without requiring the use of a DSL.

7.2 Performances

Networked live coding performances using Gibber have been given by the CREATE Ensemble, an electroacoustic group at UC Santa Barbara that places experimental interaction techniques at the center of its improvisatory practice. In the first such performance, Gibber's server enabled members of the ensemble to send code to a central computer for execution, while Gibber's built-in chatroom was used for ensemble members to communicate throughout the performance. The central computer was connected to a stereo audio feed as well as a video projector; the projected video showed both the code that members were sending to the central computer as well as the accompanying chat dialogue. An interesting side effect of the decision to send code to be executed on a central computer was that individual performers could audition the code on their laptops using personal headphones before they sent it over the network for public consumption.

The resulting piece, *G.meta* was performed on April 12th, 2012. The performance went smoothly but with some glitches in the audio due to high CPU usage. This problem with CPU inefficiency was an influential factor in deciding to begin work on a dedicated audio library for Gibber, Gibberish.js. An iteration of G.meta was also performed as part of the Santa Barbara New Music Series in August of 2012. In this version there were only three performers and the optimized audio library had been integrated into Gibber. Fewer performers and a better audio engine led to a performance free of audio stutters due to CPU performance.

In a subsequent networked live coding performance titled *Passages*, Gibber was augmented so that ensemble members could easily pass code fragments over a local area network. Building off the concept of the *Exquisite Corpse*, as code was passed around the ensemble each member receiving the fragment would modify it before it was passed to the next member and executed. The performance was conducted with ensemble members scattered throughout the audience using no amplification other than their built-in laptop speakers, a concept originally explored by the live coding group *powerbooks unplugged* [29].

Gibber has also been featured in many solo live coding performances in the United States, Europe, and Asia. One recent performance took place on July 3rd, 2014 at an Algorave[9] in London, where the majority of performers were live coders using a variety of environments and tools. The first author gave a twenty minute performance, creating both music and a generative, full-screen fragment shader from scratch using Gibber. Towards the end of the performance many of Gibber's pre-defined shaders were also employed, using the output of the live-coded fragment shader as their initial input. We look forward to future duo performances where each performer is responsible for the content of a single modality while using Gibber's abstractions to easily create mappings between them.

Audience feedback after a May 2013 solo performance by the first author was particularly interesting. During the performance, a critical error was made approximately three minutes into the performance that caused the audio processing in Gibber to begin a digital stutter, much in the fashion of a CD skipping. With no ability to resolve the problem, the performer simply copied all the code (minus the error which had been identified), refreshed the website, pasted in

[9] http://algorave.com

the copied code and executed it. This basically restarted the performance from the point immediately prior to the error. At a later point, another error again caused stuttering; however, this time the performer was able to resolve the problem after a brief moment of tension. After the performance, multiple audience members commented on how the errors lent the performance a sense of danger, and emphasized the improvisatory nature of the performance. Some went as far as to say the errors were their favorite part; we remain unclear what this says about the quality of the error-free segments of the performance.

7.3 CS Education

An interview was conducted with Carl Plant [21], a partner at *bitjam*, a company that (among other initiatives) has led a series of hacking workshops in England for teenagers. These workshops featured Arduino programming, working with Raspberry Pis, and programming music in Gibber. Each workshop hosted approximately ten students; after conducing six of them Plant estimated that he has taught Gibber to roughly sixty children between the ages of eleven and sixteen. In Plant's words the best aspect of Gibber was the "immediate gratification" it provided:

> "Kids are able to setup a beat straight away and modify it... add synth bass, wobble it."

Plant also mentioned that Gibber was preferred over other options as it is "not just long winded python coding, it can be exciting". His stated goal with the workshops was not explicitly to teach programming but rather to get kids excited about exploring it, and he believed Gibber was a much better fit for this goal than other musical (or non-musical) programming environments.

In addition to teaching teenagers, Gibber has been taught in a variety of university settings, including the University of Florida, Istanbul Technical University, Louisiana State University, Goldsmiths University of London, and the University of California at Santa Barbara.

7.4 Feedback Survey

Since December 4th, 2013, a survey link has been prominently displayed inside of Gibber. Twenty-eight users completed the survey as of July 24nd, 2014, providing qualitative and Likert scale feedback on which aspects of Gibber they enjoy and which they dislike. Of these twenty-eight users, 60% used Gibber for less than an hour before completing the survey. 62% of users had three or more years of programming experience and 85% had at least some experience programming JavaScript. 72% of users agreed or strongly agreed that they would consider using Gibber to teach or partially teach a class on programming.

88% of users either agreed or strongly agreed with the statement that they enjoyed programming in Gibber. User 20, the sole user who disagreed or strongly disagreed with the statement, expressed a dislike for web-based text editors, preferring to use vim or emacs for editing. In a similar vein, a number of users noted that they wished there was an offline version of the software:

> User 11: "I also dislike that it can't be used stand-alone without a back-end. I'd skip the social features and try to allow the app to stand-alone without the need for a remote web server."

> User 12: "(I don't like) the fact that i cant download it, and if you're offline, its a massive massive disappointment"

User 26 also expressed a desire to use Gibber within his/her bash shell. However, the same user also noted how easy it was to use since no installation is required:

> "It is very easy to setup. I already tried SuperCollider, Overtone and all this other stuff, it's nice that it's all browser based and there aren't any dependencies, build issues, etc."

In addition to allowing users with no internet connection to use Gibber, an offline version could also potentially offer improved end-user filesystem integration, easier communication with interactive devices, and more flexible options for defining the GUI used by the IDE. However, this would come with the expense of dealing with compatibility issues on different platforms.

When asked to name aspects of Gibber users found particularly enjoyable, many comments mentioned the audiovisual capabilities:

> User 13: "Simple and ultra lisible language. Audio and graphics programming in the same interface. Web based."

> User 8: "How simple and fun it is. Within a day I had visuals synced to music through my input on a projector for a new years party with 50+ people. I can spend hours playing with the software, I've never quite seen anything like it!"

> User 4: "Incredible sound and incredible graphics."

Although confident in our rationale for using a text-based environment, we were curious to learn if users thought this decision impacted Gibber's ease of use and asked users to respond to the statement "Gibber is easier to use than many visual programming environments." 50% of users agreed or strongly agreed with this statement. 17% of users had no experience with visual programming environments, and 25% of users were neutral on the subject. Only two users disagreed.

8. CONCLUSION

By integrating high-level abstractions for JavaScript into Gibber, we provide new possibilities for creative output and enable users to transfer knowledge acquired through programming in JavaScript to other domains. Three such abstractions we have integral to creative, multimodal coding are the use of proxies to manage removal/insertion of nodes in audiovisual graphs, multi-rate mapping strategies to afford mappings across modalities, and a unified system for scheduling and sequencing.

Our implementation of these abstractions provides an accessible, cross-platform, browser-based programming environment with audio, visual, and interactive affordances[10]. The system described here has been successfully used both in education and in live performance. A running instance of

[10] Gibber is open-source under the MIT license and available on GitHub at http://github.com/charlieroberts/Gibber

Gibber can be accessed on a central server hosted by the Media Arts and Technology program at UC Santa Barbara[11].

Moving forward, we will focus on the social aspects of Gibber and on improving its educational potential. For example, fine-grained sharing permissions would enable teachers and students to selectively share course-related giblets with members of their class. We plan to increase the ways giblets are consumed by enabling users to download giblets (along with the necessary supporting libraries) to embed in their own personal websites. Based on end-user feedback, we will explore creating a version of Gibber that runs without requiring internet access.

9. ACKNOWLEDGMENTS

The Robert W. Deutsch Foundation generously supported this research. This work was partially supported by ONR grant N00014-14-1-0133 and ARL/ARO MURI grant W911NF-09-1-0553.

10. REFERENCES

[1] Quil. https://github.com/quil/quil.

[2] SCGraph - a graphics server for SuperCollider. http://scgraph.github.io.

[3] ShaderTone. https://github.com/overtone/shadertone.

[4] S. Aaron and A. F. Blackwell. From sonic Pi to overtone: creative musical experiences with domain-specific and functional languages. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*, pages 35–46. ACM, 2013.

[5] P. Boytchev. Logo tree project. http://elica.net/download/papers/LogoTreeProject.pdf, 2007.

[6] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised Sound*, 8(3), 2004.

[7] D. Della Casa and G. John. LiveCodeLab 2.0 and its language LiveCodeLang. In *Proceedings of the second ACM SIGPLAN workshop on Functional art, music, modeling & design*. ACM, 2014.

[8] G. Essl. *Playing with Time: Manipulation of Time and Rate in a Multi-rate Signal Processing Pipeline*. Ann Arbor, MI: MPublishing, University of Michigan Library, 2012.

[9] I. Greenberg, D. Kumar, and D. Xu. Creative coding and visual portfolios for CS1. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 247–252. ACM, 2012.

[10] D. Griffiths. Fluxus. In A. Blackwell, A. McLean, J. Noble, and J. Rohrhuber, editors, *Collaboration and learning through live coding*, Report from Dagstuhl Seminar 13382, pages 149–150. Dagstuhl, Germany, 2013.

[11] J. Maeda. *Design by numbers*. The MIT Press, 2001.

[12] T. Magnusson. ixi lang: a SuperCollider parasite for live coding. In *Proceedings of the International Computer Music Conference*. University of Huddersfield, 2011.

[13] T. Magnusson. Herding cats: Observing live coding in the wild. *Computer Music Journal*, 38(1):8–16, 2014.

[11] http://gibber.mat.ucsb.edu

[14] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[15] C. McKinney. Quick live coding collaboration in the web browser. In *Proceedings of the 2014 Conference on New Interfaces for Musical Expression*, pages 379–382, 2014.

[16] A. McLean. Tidal – mini language for live coding pattern. http://toplap.org/tidal/.

[17] A. McLean and G. Wiggins. Tidal–pattern language for the live coding of music. In *Proceedings of the 7th sound and music computing conference*, 2010.

[18] C. A. McLean et al. *Artist-Programmers and Programming Languages for the Arts*. PhD thesis, 2011.

[19] J. A. Mills III, D. Di Fede, and N. Brix. Music programming in minim. In *Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME 2010). Sydney, Australia*, pages 37–42, 2010.

[20] S. Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., 1980.

[21] C. Plant. Interview, 12 2013. Technical Director: bitjam, ltd.

[22] C. Reas and B. Fry. Processing: programming for the media arts. *AI & SOCIETY*, 20(4):526–538, 2006.

[23] J. Resig. Javascript as a first language. http://ejohn.org/blog/javascript-as-a-first-language/, 20011.

[24] J. Resig. Processing.js. http://ejohn.org/blog/processingjs/, 2008.

[25] C. Roberts and J. Kuchera-Morin. Gibber: Live coding audio in the browser. *Proceedings of the International Computer Music Conference*, 2012.

[26] C. Roberts, G. Wakefield, and M. Wright. The web browser as synthesizer and interface. In *Proceedings of the 2013 Conference on New Interfaces for Musical Expression (NIME 2013)*, volume 2013, 2013.

[27] C. Roberts, M. Wright, J. Kuchera-Morin, and T. Höllerer. Rapid creation and publication of digital musical instruments. In *Proceedings of New Interfaces for Musical Expression*, volume 2014, 2014.

[28] J. Rohrhuber, A. de Campo, and R. Wieser. Algorithms today notes on language design for just in time programming. In *Proceedings of the International Computer Music Conference*, 2005.

[29] J. Rohrhuber, A. de Campo, R. Wieser, J.-K. van Kampen, E. Ho, and H. Hölzl. Purloined letters and distributed persons. In *Music in the Global Village Conference (Budapest)*, 2007.

[30] A. Sorensen. Extempore. http://extempore.moso.com.au.

[31] A. Sorensen. The Many Faces of a Temporal Recursion. http://extempore.moso.com.au/temporal_recursion.html, 2013.

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

[33] G. Wang and P. R. Cook. *The ChucK Audio Programming Language. a Strongly-timed and On-the-fly Environ/mentality*. PhD thesis, 2008.