Music Programming in *Gibber*

Charles Roberts
charlie@charlie-roberts.com

Matthew Wright
matt@create.ucsb.edu

JoAnn Kuchera-Morin
jkm@create.ucsb.edu

University of California at Santa Barbara
Media Arts & Technology Program

ABSTRACT

We document music programming in Gibber, a creative coding environment for the browser. We describe affordances for a sample-accurate and functional approach to scheduling, pattern creation and manipulation, audio synthesis, using rhythm and harmony, and score definition and playback.

1. INTRODUCTION

First introduced as an environment for live coding performance in 2012 [1], *Gibber* has gradually expanded in scope to include musical instrument design [2], 2D and 3D graphics, and improved affordances for both live coding and musical composition. This paper codifies the compositional affordances of Gibber, both from the perspective of real-time composition (also known as live coding [3, 4]) and more traditional compositional practice.

Gibber embodies an interesting set of design decisions. Although it primarily uses a high-level language, JavaScript, for end-user programming, [1] it also offers low-level affordances not found in many music programming languages, such as the creation of multiple, sample-accurate clocks with audio-rate modulation of timing and intra-block audio graph modification. Efficiency is optimized whenever possible [5, 6] but it will most likely never be as efficient as music programming languages with C bindings; however, we feel the accessibility of the environment (a link is all that is needed to start programming in Gibber), the simplicity of the notation, and interesting approaches to time and pattern make Gibber a compelling choice as a musical programming tool.

In addition to its dedicated web environment for programming, there is also a standalone JavaScript library, gibber.lib.js, that can be included and used inside of any HTML document. This library has been ported to work with *p5.js* [2], the JavaScript implementation of the Processing creative coding system.

[1] Graphical shader programming in Gibber is performed in GLSL.
[2] http://p5js.org

We begin by discussing the synthesis possibilities of Gibber, which include a wide variety of both synthesizers and effects. This is followed by a discussion of musical notation, including the use of symbols to represent concepts from the Western musical tradition. We discuss musical structure in Gibber, beginning with the use of patterns and sequencers and proceeding to higher-level heterarchical structures. We conclude with a discussion of our system in relation to other music programming languages and a brief look at Gibber in use.

2. SYNTHESIS AND SIGNAL PROCESSING

Gibber provides a variety of options for audio synthesis, many of which are wrappers around the unit generators and instruments found in Gibberish.js, a lower-level DSP library for JavaScript [5] written specifically to support Gibber. While users can construct custom instruments from low-level unit generators in Gibber, one of its strengths compared to many music programming languages (particularly JavaScript-based) is an emphasis on usable, precomposed instruments. This frees composers and beginning programmers from the need to also be synthesis experts in order to achieve interesting sonic results. Here we provide a short overview of available synthesis options (some previously discussed in the context of Gibberish.js [5]), along with a more detailed discussion of several high-level unit generators that are more specific to Gibber.

2.1 Raw Oscillators

The canonical waveforms are provided as wavetable oscillators. Additional quasi-bandlimited waveforms (pulse, square and saw) are provided using FM synthesis techniques [7].

2.2 Sample Loading, Playback, and Manipulation

Gibber provides a variety of options for audiofile playback. The standard `Sampler` object can load multiple audiofiles concurrently, and provides methods to schedule changes to the active buffer. It can be used to sample audio input and any signal in Gibber's audio graph, including the master output. A `Sampler`'s built-in `save` method writes its stored audio buffer to a `.wav` file, providing an easy way to record performances or compositions.

The `Freesound` object enables users to easily query and download audiofiles from the Freesound database [8]. The constructor for the `Freesound` object takes three forms. A single integer argument uniquely identifies a particular sound in the database. A single string argument is used to query the database and automatically return the highest-ranked result. Otherwise the argument is a dictionary containing the search query and other options such as minimum and maximum file duration and selection parameters. Once an audiofile has been loaded from the database the functionality of the `Freesound` object mirrors the `Sampler` object.

The `SoundFont` object provides basic support for loading sets of samples that adhere to the General MIDI specification, giving easy access to common synth sounds.

2.3 Synthesizers

Gibber includes a variety of pre-built synthesis objects with varying degrees of complexity. The most basic, `Synth`, comprises an oscillator and an envelope. The `Mono` object is a three-oscillator monosynth with a 24db resonant filter, while the `FM` object provides simple two-operator FM synthesis with an envelope controlling both output amplitude and the modulator index. A software emulation of the Roland TR-808 drum machine is also available, providing a synthetic counterpoint to the sample-based `Drums` object.

2.4 Audio Effects

Gibber's selection of audio effects includes filters, delay, reverb, distortion, bit-crushing, flanging, chorus, vibrato, ring modulation, etc.; most work in either mono or stereo. Every audio generator has a built-in array of effects applied in series to the generator's output. Convenience methods add effects to this array, remove effects by name or by number in the effects chain, or remove a particular matched object. The `Bus` object can easily create parallel effects chains.

2.5 Bussing, Grouping and Soloing

The `Bus` object simply sums its inputs and then routes them to another destination, typically either an effect or Gibber's master output. Busses also have panning and amplitude properties that can be modulated and sequenced. The `Group` command takes an existing set of synthesis objects, disconnects them from whatever busses they are currently connected to and routes them all into a single bus. This can be used to quickly apply a single instance of an effect to a mix of sound sources, as opposed to creating separate copies of the effect for each individual ugen.

The `solo` function mutes all unit generators except those passed as arguments to the function. Repeated calls to solo can reintroduce elements to a live performance as they are added to the argument list. A call to `solo` with no arguments unmutes all previously muted objects.

2.6 Speech Synthesis and Vocoding

The `Speech` object controls a speech synthesis algorithm. Users can enter English text and customize its pitch and intra-word spacing. A background thread then renders text into the buffer of a `Sampler` object for playback and manipulation.

The `Vocoder` object operates via dual banks of band-pass filters; the first analyzes the frequency content of an input *modulator* signal, while the second applies imparts the resulting spectrum on an input *carrier* signal. The `Robot` object is a vocoder with a PWM polysynth as the carrier and a `Speech` object as the modulator, producing classic robot voice effects.

```
robot = Robot()
  .say.seq([ 'this','is','a','test' ], 1/2 )
  .chord.seq( Rndi(0,8,3) )
```

This example uses a different word every half note as the modulator, simultaneously triggering a new chord (three random notes from Gibber's global scale object) as the carrier.

3. MUSICAL NOTATION IN GIBBER

Gibber offers multiple paradigms for representing musical concepts such as pitch and rhythm, including both raw numeric values (e.g., Hertz, ms) and terminology from Western music theory (e.g., C4, half note).

3.1 Rhythm and Meter

Gibber's master `Clock` object defaults to 4/4 meter at a tempo of 120 beats per minute. Users can easily change these values, and such changes can be scheduled over time.

```
// set clock to 90 beats per minute
Clock.bpm = 90
// set time signature to be 6/8
Clock.timeSignature = '6/8'

// change tempo every four measures
Clock.bpm.seq(
  [ 90, 120, 140 ],
  measures( 4 )
)
```

Gibber users can indicate timing and duration using notations typical to Western music. For example, `1/4` indicates a quarter note, `1/13` indicates a thirteenth note, and `1` represents a whole note. Any valid JavaScript numeric expression is accepted, enabling math such as `10*3/16`. Gibber also enables specifying duration by number of samples, so there is a `maxMeasures` property of the master clock which sets the threshold between interpreting small numbers as counting measures and large numbers as counting samples. For example, with the `maxMeasures` property set to its default value of 44, the duration `34.5` would indicate thirty-four whole notes plus a half note, while the duration `50` would indicate 50 audio samples. To indicate a number of measures above this ceiling, the `measures` function return the number of samples corresponding to a given number of measures at the current tempo. Users who wish to avoid this amibiguity can set `maxMeasures` zero and then use explicit function calls

to the `beats`, `measures`, `samples`, `ms`, and `seconds` methods.

3.2 Pitch, Frequency, Chords, and Scales

Gibber understands both numeric and symbolic representation of pitch and chords. Although it comes with a variety of available tunings and modes, users are may also define their own.

3.2.1 Pitch and Frequency

When the `note` method is called on a synthesis object, Gibber examines the first value argument to it in order to determine what pitch the unit generator should begin to output.

1. If this first argument is a string, this denotes a note and octave, e.g., 'c4' indicates the note C in the fourth octave, while 'e#2' represents E-sharp (enharmonic F) in the second octave.

2. If the first argument is a number greater than the global default variable `Gibber.minFrequencyValue`, the number is interpreted as raw frequency in Hertz.

3. If the numeric first argument is below this threshold, it is interpreted as a scale degree. Non-integer scale degrees are supported, with any fractional part dictating the amount of upward shift between the integer part and the next scale degree. For example, in the scale of C major, a value of 3.5 would resolve to F#, half way between the fourth scale degree F and the fifth scale degree G (scale indexing is zero-origin). By default, this is relative to Gibber's default scale object `Gibber.scale`, but separate scales can also be assigned to individual unit generators.

These examples show these alternatives:

```
// examples assume Gibber.minFrequencyValue is 50
mySynth = Synth()
// plays C in the fourth octave
mySynth.note( 'c4' )
// plays the frequency 440
mySynth.note( 440 )
// plays the root scale degree
mySynth.note( 0 )
// plays the third scale degree
mySynth.note( 2 )
```

3.2.2 Chords and Arpeggios

A similar concept also holds true for calling the `chord` method on synths. Strings enable users to define chords with root, octave and sonority. Example valid chords in Gibber include "c4maj7", "d#3min7b9" and "ab2dim7". Instead of passing strings, end-users can also pass arrays of numbers; the same rules apply for these numbers as for the `note` method. Thus, to play the root, the third and the fifth of the current scale we would write `mySynth.chord( [0,2,4] )`.

Gibber's arpeggiator object `Arp` accepts a chord argument (again, either a string or array of numbers), a duration for each note, a pattern for the arpeggiation, and the number of octaves to extend the chord over. The following code plays a C major seventh chord over four octaves:

```
arp = Arp( 'c4maj7', 1/16, 'updown', 4 )
```

3.2.3 Scales

Scales and tunings are somewhat conflated in Gibber, where a 'Scale' consists of a root note and a series of ratios in frequency between that root note and the subsequent scale degrees. Examples of alternate scales / tunings in Gibber include just intonation, Pythagorean tuning, Messiaen's limited modes of transposition, and the Shruti of Indian classical music. Custom scales be easily defined by giving a root note (either as Hertz or symbolic pitch) and an array of frequency ratios, as shown below:

```
//  5 tone equal temperament
fiveTET = Theory.CustomScale(
  'c4',
  [ 1, 1.149, 1.32, 1.516, 1.741 ]
)
```

4. SEQUENCING AND PATTERN

Gibber uses single-sample processing both in its audio engine and in its scheduler, so functions can be executed with sample-accurate timing and create modifications to the audio graph that take effect on the next sample, as opposed to having to wait for the next block of samples. The `future` function schedules another function to be invoked at some point in the future, with sample-level precision, but the most idiomatic method of dealing with time in Gibber is through its sequencer objects.

4.1 Sequencers

A Gibber sequencer consists of two functions: `values` determines each successive output and `durations` schedules each output. Users can supply arbitrary functions for both purposes; they can also pass single literals or arrays of values to the sequencer constructor, which will then wrap the arguments in `Pattern` functions, discussed in Section 4.2. Unlike some domain-specific and mini-languages that consider both timing and output concurrently when defining musical pattern [9, 10], sequencers in Gibber typically treat these separately. Two exceptions to this are the `Drums` and `EDrums` objects, which enable users to quickly define both timing and sounds in drum patterns using a mini-language.

Although sequencers were originally stand-alone objects in Gibber,[3] we created an abstraction that makes sequencing much more immediately accessible, as first reported in [12]. This abstraction adds a built-in sequencer to every audiovisual object in Gibber, and a `seq` method to all properties and methods enabling users to quickly assign sequences to the sequencer for playback. Compare the old and new syntaxes for sequencing below:

[3] Sequencers are still available as the standalone `Seq` object, which is comparable to the `Pbind` object in SuperCollider [11]

```
synth = Synth2()

// old syntax
myseq = Seq({
  target:   synth,
  note:     [0,1,2,3,4],
  cutoff:   [.1,.2,.3,.4],
  durations:[1/2,1/4,1/8]
})

// new syntax
synth.note.seq( [0,1,2,3,4], [1/2,1/4,1/8] )
  .cutoff.seq( [.1,.2,.3,.4] )
```

In addition to brevity, one advantage of the newer method is the ability to use different timings for sequencing different properties or methods with a single sequencer; the older syntax defines a single array of timings which is used for sequence changes to all audiovisual properties. Although there are times where such varied timing is required, it can also be musically useful to have changes to multiple properties to occur simultaneously. The most common use case for this is triggering changes to synthesis properties whenever a note is played. E.g, we might want to ensure that, even if the timing of note messages is determined stochastically, changes to other properties (such as panning or filter cutoff) are triggered at the same time. This can be accomplished in Gibber by omitting information about timing from a call to the `.seq` method; in this case the corresponding sequence is triggered *any time changes to another property or calls to a method on the same synthesis object are triggered via sequencing.* We illustrate this in the examples below: in the first, a FM synthesizer has separate timings for manipulating calls to its `note` method and changes to its `index` and `cmRatio` properties (carrier-to-modulation ratio); in the second, changes to the `cutoff` and `resonance` properties of a monosynth are triggered whenever a `note` message occurs.

```
// make a note every quarter note, while also
// changing modulation index every sixteenth note
fm = FM()
  .note.seq( [0,1,2,3], 1/4 )
  .index.seq( Rndf(5,20), 1/16 )

// calls to cutoff.seq and resonance.seq
// don't specify timing, so the note sequence
// triggers them, producing a new cutoff
// and resonance for each note.
synth = Mono()
  .note.seq( [0,1,2,3], [1/4,1/8].rnd() )
  .cutoff.seq( [.1,.2,.3] )
  .resonance.seq( [.5,.75] )
```

The sequencers built-in to audiovisual objects in Gibber have a `rate` property that supports audio-rate modulation of phase (and thus scheduling), however, this setting affects all sequences created for a given object. Any of the various sequences managed by an object's internal sequencer can be individually started, stopped and modified at will.

4.2 Patterns

Patterns in Gibber are functions that repeatedly return members of underlying lists. By default these lists are ordered, and subsequent calls to the pattern functions will simply return the next item in the list. For example:

```
ptrn = Pattern( 1,2,3 )
ptrn() // outputs 1
ptrn() // outputs 2
ptrn() // outputs 3
```

However, every pattern has an array of `filters` that can change the pattern's return values or how it operates. Each time the pattern is to output another value it invokes each filter, passing its current index position, phase increment (i.e., how far the pattern advances through the list for each function call), the provisional value to be output, and a pointer to the pattern object itself. A pattern filter returns an array containing the output value, index, and phase increment, which may have any relationship to the filter's inputs. Below is an example of a typical pattern filter that selects a random pattern value for output.

```
random = function( args, pattern ) {
  var index = rndi( 0, pattern.values.length - 1 )
      ,
      out = pattern.values[ index ]

  args[ 0 ] = out
  args[ 1 ] = index

  return args
})
```

Another typical pattern filter is `repeat`, which examines the output of the pattern and, if it matches a particular value, subsequently repeats that output on a defined number future invocations of the pattern function. This can be used to line up complex polyrhythmic phrases, even if they are stochastically determined. Consider the following:

```
a = FM().note.seq(
  [ 0,7 ],
  [ 1/4,1/3,1/12,1/8 ].rnd( 1/3,3,1/12,6 )
)
```

This example calls the timing array's `rnd` method, causing two filters to be applied to the array when it is converted to a pattern object. The first filter is identical to the `random` function defined above, picking a random array value on each call. The second filter is the `repeat` filter, enabling a user to specify that particular values should repeat a certain number of times whenever they are selected. In the above example, whenever a 1/3 note duration is selected it will be played three times (in effect, a half note triplet); whenever a 1/12 note duration is selected it will be played six times.

In addition to filters that are applied each time a pattern function is executed, there are also a variety of methods that can be used to change the underlying lists owned by pattern objects; inspiration for these manipulations came from other live coding environments (Tidal [10] in particular), Laurie Spiegel's summary of pattern manipulation techniques [13], and extensive experimentation:

- *Reverse/Retrograde* - reverse the order of the underlying list

- *Invert* - As in twelve-tone technique, invert intervals of all list members as compared to the first note in the row
- *Rotate* - shift the items in the list by a provided value. For example 1,2,3,4 rotated +1 would be 4,1,2,3.
- *Range* - Restrict the list indices available for selection to a provided range
- *Store* - store the current list values for later recall
- *Switch* - switch list values to a previously stored set
- *Reset* - reset list to the original values from when the list was first created.
- *Transpose* - modify each member of list by adding an argument value to it
- *Scale* - scale each item in the list by an argument value
- *Shuffle* - randomize the order of the list
- *Flip* - this method maintains the current members of the list, but changes their order such that the lowest element now occupies the former position of the highest element, the second lowest occupies the position of the second highest, etc.

In addition to these methods, users can also set the `stepSize` property for any pattern, which determines how much the internal phase of the pattern is modified after each execution. Negative step sizes traverse the pattern values in reverse.

An important aspect of all these transformations is that they can easily be scheduled over time using the sequencing abstractions discussed in Section 4.1. For example, in the code below, a melody (held in the `values` array of the note method) is transposed up one scale degree every measure. Every four measures the `reset` method is called, returning the pattern to its original state. This process loops indefinitely.

```
a = Mono('lead').note.seq( [0,1,2,3], 1/4 )

// transpose by one each measure
a.note.values.transpose.seq( 1,1 )

// reset values list every 4 measures
a.note.values.reset.seq( null, 4 )
```

A novel visualization scheme reported in [14] reveals both transformations to the patterns as well as which list member is outputted whenever a pattern function is executed using the source code itself as the primary visualization component. This can help performers, composers, students and audiences understand the algorithmic processes at play in a given performance.

5. HIGH-LEVEL MUSICAL STRUCTURE

Although Gibber was initially created as a live coding environment, many users expressed interest in using it for non-realtime composition. Initially, Gibber's `future` method was used by some users to define compositional structure. However, we consider this method to be too low-level for many compositional purposes, as the absolute time values used in calls to future are inflexible in accommodating change and/or interactivity. For example, relative durations would make it possible to easily insert new sections into an existing piece. We wanted to create a more flexible abstraction for defining musical structure while still providing for interactivity and dynamism. The `Score` object achieves these goals.

5.1 The Score Object

The `Score` object in Gibber consists of JavaScript functions and corresponding timestamps for execution. Unlike `Seq` objects, where output values and timing can be manipulated independently, each score timestamp is uniquely associated with a single function. Timestamps are relative to the previous entry in the score; in the case of the first entry, the timestamp is relative to when the score object's `start` method is called. The simplest use of a score object is to create a form of note list, although it can be somewhat verbose for this purpose:

```
synth = Mono()
score = Score([
  0, function() { synth.note('c4') },
  beats(2), function() { synth.note('c5') },
  measures(1), function() { synth.note('d5') }
])
```

One problem with this technique stems from the need to specify commands to be executed in the future, handled here by wrapping the code in anonymous functions. Especially for short commands such as playing a single note, these anonymous functions increase the viscosity of the notation and make it harder to read the score. Most other solutions involving existing JavaScript techniques are similarly verbose and conceptually challenging for beginning programmers, motivating Gibber's deferred execution notation.

5.2 Deferred Execution

Gibber allows an underscore after the name of any audiovisual method or property as syntactic sugar to create a function for deferred execution. For example, the code `syn.note_('c4')` is equivalent to `function() syn.note('c4')` — creating a zero-argument function that will play the note when invoked. Using this syntax, scores can look like this:

```
score = Score([
  0,      syn.note_( 'c4' ),
  beats(1), syn.note_( 'd4' ),
  beats(1), syn.attack_( ms(500) ),
  beats(1), syn.note_( 'e4' )
])
```

Although not a perfect solution, we feel this provides the best available option in Gibber. Of course, end-users are always free to wrap code blocks in functions (and this is necessary for score element that consists of multiple lines of code) or use other methods built-in to JavaScript to create functions for later execution.

5.3 Interactive Scores

Score playback can be paused via method calls occurring outside the score and by timestamps in the score itself. Whenever a timestamp is equal to the special class property `Score.wait`, playback pauses until the score object receives a call to its `next` method.[4] Calls to `next` can be made from within the live coding environment, but could also be generated, for example, by incoming OSC messages or one of the user interface elements that Gibber offers. In the example below the score halts after a single note is played and immediately plays a second note after its `next` method is called.

```
score = Score([
  0, function(){ synth.note( 'c4' ) },
  Score.wait, null,
  0, function(){ synth.note( 'd4' ) },
])
```

5.4 Hierarchical Scores

Multiple `Score` objects can operate concurrently. Each score object can launch other score objects and base their timing on the completion of other scores. This open-ended approach enables users to define their own hierarchical and heterarchical structures. Consider the canonical form of a pop song: verse, chorus, verse, chorus, bridge, chorus. Although this form could be achieved with a single score object, encapsulating each of the three section types in a score while using a fourth score to link them together in the appropriate order provides greater flexibility in making modifications. The simplified example below provides this structure, with three sections that each play a single note for demonstrative purposes and a fourth score which orders and times their playback. When Gibber sees the `oncomplete` property of a score object given as a timestamp, it will wait for the identified score to finish playback before advancing to the next timestamp / function pair. The convenience method `combine` can also be used to quickly sequence scores one after another.

```
verse =  Score([ beats(1/2), synth.note_('c4') ])
chorus = Score([ beats(1/2), synth.note_('d4') ])
bridge = Score([ beats(1/2), synth.note_('e4') ])

song = Score([
  0,                verse,
  verse.oncomplete,  chorus,
  chorus.oncomplete, verse,
  verse.oncomplete,  chorus,
  chorus.oncomplete, bridge,
  bridge.oncomplete, chorus
])

// convenience method
song = Score.combine(
  verse,chorus,verse,chorus,bridge,chorus
)
```

[4] 'next' was chosen as the name for this method due to its similarity in function with a method used to resume execution by *generators*, a system for coroutines that will be integrated into the next version of JavaScript (ES6).

6. DISCUSSION AND COMPARISONS

In this section we relate our work to other musical programming languages, in particular discussing time, pattern, and musical structure.

6.1 Time

While Gibber lacks support for strongly-timed semantics found in languages like ChucK [15] and LC [16], it nonetheless provides sample-accurate scheduling with a level of granularity not found in audio programming languages that use block-rate processing, and one that is unique to JavaScript libraries as of February of 2015. There are no other JS libraries that provide sample-accurate scheduling, the ability to dynamically change the audio graph from one sample to the next, or audio-rate modulation of timing with multiple clock sources. Although a per-sample processing approach to DSP incurs performance penalties, we believe the experimental approaches to time it enables are often worth the cost.

Although use of cyclic time with the `.seq` method is the most idiomatic approach to time in Gibber, other techniques can also be used. For example, *temporal recursion* [17], used heavily in the live coding environment Impromptu [18] is fairly straightforward using Gibber's built-in `future` method. The goals of temporal recursion can be considered similar to the approach taken by the live coding environment *Sonic Pi* [19] and by ChucK; however, in these systems a loop with the ability to *sleep* (in the case of Sonic PI) or explicitly advance time (in the case of ChucK) is used instead of a recursive function. LuaAV [20] provides a similar approach through its use of co-routines. Despite different methodologies, all these methods focus on repeated execution of code blocks over time, whether in a loop iteration or in a function call. A simple example of temporal recursion in Gibber is given below; more advanced examples would include branching and, potentially, launching other temporally recursive functions.

```
syn = Synth()

recurseMe = function() {
  syn.note( rndi(0,10) )

  // repeat at random every 1-4 measures
  future( recurseMe, rndi(1,4) )
}

recurseMe()
```

After its initial creation, the `recurseMe` function can be freely redefined, enabling performers to gradually develop complex ideas and themes.

6.2 Patterns and Sequencing

A number of other music languages devote significant resources to pattern manipulation. In Tidal [10], a terse yet flexible mini-language enables the definition of patterns that contain both timing and output information; this mini-language is used in conjunction with Haskell to send OSC messages to control various audio synthesis systems. Since patterns in

Tidal contain both timing and values, different types of transformations are available compared to those found in Gibber, although many are the same: reversing, rotating, limiting the range of values available etc. While we believe separation of output values and timing provides the potential for additional fine-grained control over pattern manipulation, we also think there are a variety of useful manipulations in Tidal that work best when considering timing and output together. To enable this functionality in Gibber we are considering adding transformations at the sequence level in addition the pattern level; these transformations could affect both timing and output of sequences as needed.

One precursor to Gibber that shares Gibber's decoupling of output and timing in sequences is the *Conductive* library for Haskell by Renick Bell [21]. The `Player` object in Conductive acts in a similar fashion to Gibber's `.seq` method: there is a function which carries out an action and a function that determines the timing of actions. We feel this is an intuitive, yet flexible, model. In addition, Conductive also adds a `TempoClock` data structure to each `Player` object, which affords adjustments to both tempo and time signature. In Gibber, tempo is controlled via the `rate` property on each sequence, which can be modulated at audio-rate. However, time signature is only available as a control on the master clock in Gibber; there is currently no mechanism for specifying it on a per-sequence level.

SuperCollider [22] is another language that offers detailed control over pattern. Although SuperCollider provides more options for manipulating and organizing patterns than Gibber, it does so by exposing a variety of different classes that each treat patterns somewhat differently. In Gibber there is a single pattern class, but it is capable of mimicking many of the various options in SuperCollider with functions that act as filters on pattern output and phase, as discussed in Section 4.2. Although we feel our approach is perhaps conceptually more elegant, it does pose a disadvantage in terms of verbosity as the various `Pattern` filters in Gibber have to be explicitly enabled.

The system of filters used in Gibber for manipulating pattern function resembles the *behaviors* found on *collections* in the Hierarchical Music Specification Language (HSML) [23]. Just as the authors of HSML encourage users to generate their own behaviors, Gibber enables users to easily define their own custom filters for manipulating patterns, as shown in Figure 4.2.

6.3 Musical Structure

The best comparison to Gibber's `Score` object is SuperCollider's `Task` object. SClang support for coroutines means that code can be written synchronously with delays as needed, which is more elegant than the array-based approach provided here. However, as JavaScript support for coroutines is not standardized across browsers (as of February of 2015 [5]), we feel our solution is an efficient, lisible compromise within our language constraints.

Buxton et al. were perhaps the first to explore this type of open-ended musical hierarchy in their Structured Sound Synthesis Project [24]. The SmOKe music representation system [25, 26] also contains a open-ended approach to hierarchical structure similar to the `Score` object in Gibber. While the low-level unit of a `Score` object in Gibber is an executable function, SmOKe *EventList* objects contain *Events*, which define abstracted key/value dictionaries to be used in processing. Each event has a start time that it is associated with in the event list, and event lists can nest other event lists arbitrarily.

In the Java Music Specification Language(JSML) [27], a successor to HSML, a variety of collection objects are available for scheduling the launch of objects that implement the *Composable* Java interface. These collection objects can be freely nested, and provide affordances not currently available in Gibber's `Score` object. For example, the JSML `ParallelCollection` object launches a set of multiple objects or collections in parallel, and waits for all of them to finish before continuing to the next set. We plan to extend Gibber's score object to support similar functionality.

7. CONCLUSIONS

Gibber has been used for a variety of purposes since its creation. It has been used to teach music principles and digital instrument design to middle school and high school students, in a variety of both solo and ensemble live coding performances, and to publish compositions online. Whenever a user saves a file to Gibber's central database, they are given a URL that links directly to their file for sharing. This URL not a link to a rendered audio file, but rather to the source code itself, enabling consumers to modify and experiment with published compositions.

In June of 2014 the band 65daysofstatic published a piece in Gibber and promoted the URL on their Facebook page [6] ; Gibber's server was rapidly overwhelmed by people trying to view the composition. After re-configuring the server to better accommodate the load, thousands of people successfully viewed and ran the composition inside of Gibber; we believe this to be the first time an algorithmic composition by a noted popular music group was published via real-time rendering in the browser. The features we have added to Gibber since then, particularly the `Score` and `Pattern` objects, should ease the compositional process in future works, and we look forward to seeing how users take advantage of them.

Acknowledgments

We gratefully acknowledge the Robert W. Deutsch Foundation for providing graduate and postdoctoral fellowships enabling this work.

[5] http://kangax.github.io/compat-table/es6/#generators

[6] http://on.fb.me/1Je2PGS

8. REFERENCES

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

[2] C. Roberts, M. Wright, J. Kuchera-Morin, and T. Höllerer, "Rapid Creation and Publication of Digital Musical Instruments," in *Proceedings of the New Interfaces for Musical Expression Conference*, 2014, pp. 239–242.

[3] T. Magnusson, "Herding cats: Observing live coding in the wild," *Computer Music Journal*, vol. 38, no. 1, pp. 8–16, 2014.

[4] N. Collins, A. McLean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 03, pp. 321–330, 2003.

[5] 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, pp. 313–318.

[6] C. Roberts, G. Wakefield, M. Wright, and J. Kuchera-Morin, "Designing Musical Instruments for the Browser," *Computer Music Journal*, vol. 39, no. 1, pp. 27–40, 2015.

[7] P. Schoffhauzer, "Synthesis of quasi-bandlimited analog waveforms using frequency modulation," 2008, [Online at:http://scp.web.elte.hu/papers/synthesis1.pdf (checked February 11th, 2015)].

[8] V. Akkermans, F. Font, J. Funollet, B. De Jong, G. Roma, S. Togias, and X. Serra, "Freesound 2: An improved platform for sharing audio clips," in *Klapuri A, Leider C, editors. Proceedings of the 12th International Society for Music Information Retrieval Conference*. International Society for Music Information Retrieval (ISMIR), 2011.

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

[10] 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.

[11] R. Kuivila, "Events and patterns," *The SuperCollider Book*, pp. 179–205, 2011.

[12] C. Roberts, M. Wright, J. Kuchera-Morin, and T. Höllerer, "Gibber: Abstractions for Creative Multimedia Programming," in *Proceedings of the ACM International Conference on Multimedia*. ACM, 2014, pp. 67–76.

[13] L. Spiegel, "Manipulations of musical patterns," in *Proceedings of the Symposium on Small Computers and the Arts*, 1981, pp. 19–22.

[14] C. Roberts, M. Wright, and K.-M. JoAnn, "Beyond Editing: Extended Interaction with Textual Code Fragments," in *Proceedings of the New Interfaces for Musical Expression Conference*, 2015.

[15] G. Wang and P. R. Cook, "The ChucK Audio Programming Language. a Strongly-timed and On-the-fly Environ/mentality," Ph.D. dissertation, 2008.

[16] H. Nishino, N. OSAKA, and R. NAKATSU, "LC: A New Computer Music Programming Language with Three Core Features," *submitted to Proc. ICMC-SMC*, 2014.

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

[18] ——, "Impromptu: An interactive programming environment for composition and performance," in *Proceedings of the Australasian Computer Music Conference 2009*, 2005.

[19] S. Aaron, D. Orchard, and A. F. Blackwell, "Temporal semantics for a live coding language," in *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*. ACM, 2014, pp. 37–47.

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

[21] R. Bell, "An Interface for Realtime Music Using Interpreted Haskell," in *Proceedings of the Linux Audio Conference*, 2011.

[22] J. McCartney, "Rethinking the computer music language: SuperCollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[23] L. Polansky, P. Burk, and D. Rosenboom, "HMSL (Hierarchical Music Specification Language): A Theoretical Overview," *Perspectives of New Music*, pp. 136–178, 1990.

[24] W. Buxton, W. Reeves, R. Baecker, and L. Mezei, "The use of hierarchy and instance in a data structure for computer music," *Computer Music Journal*, pp. 10–20, 1978.

[25] S. T. Pope, "The SmOKe music representation, description language, and interchange format," in *Proceedings of the International Computer Music Conference*. Citeseer, 1992, pp. 106–106.

[26] S. Travis Pope, "Object-oriented music representation," *Organised Sound*, vol. 1, no. 01, pp. 56–68, 1996.

[27] N. Didkovsky and P. Burk, "Java Music Specification Language, an introduction and overview," in *Proceedings of the International Computer Music Conference*, 2001, pp. 123–126.