# Live Coding the Digital Audio Workstation

Charles Roberts
Rochester Institute of Technology
charlie@charlie-roberts.com

Graham Wakefield
York University
grrrwaaa@gmail.com

ABSTRACT

We describe a new live-coding system, named *Gibberwocky*, which integrates the Ableton Live digital audio workstation with a browser-based textual coding environment derived from the Gibber project. The live-coding interface emphasizes sequencing of events and parameter changes as opposed to audio synthesis, but also affords rapid construction of audio graphs to modulate any parameter in Ableton Live with much greater resolution than is typically found in live-coding systems outputting MIDI. It also extends earlier research on dynamic code annotations with more specific solutions for individual pattern generators, and abstractions for musical scores.

## 1 Introduction

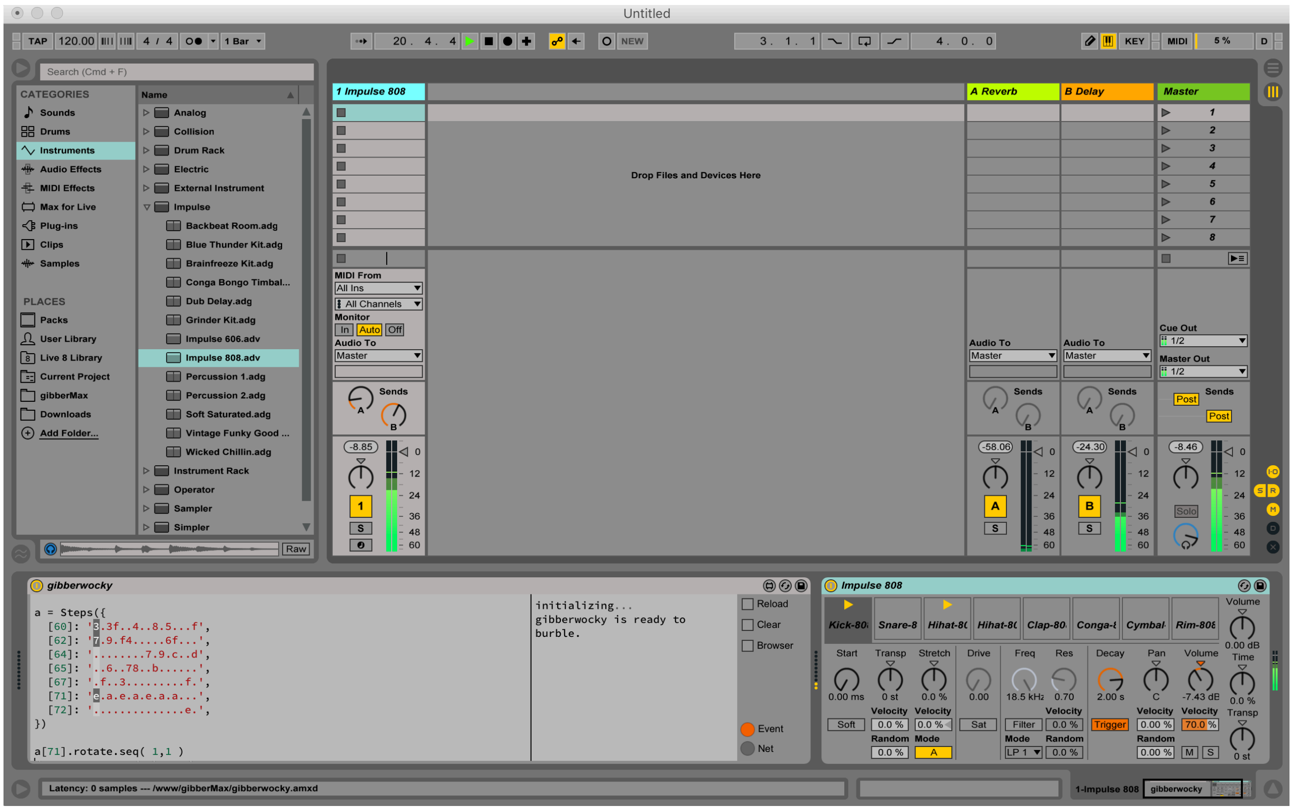
The ecosystem of synthesis plugins for digital audio workstations (DAWs) has never been richer. Hundreds of audio programmers have devoted their careers to creating inventive DSP algorithms that run with incredible efficiency, and, sometimes, incredible ingenuity. Despite this, most commonly used live-coding environments eschew integration with DAWs in favor of integrating with music programming languages and runtime engines such as SuperCollider (Aaron and Blackwell 2013; Magnusson 2011; Bell 2011), or creating custom audio engines from scratch (McLean and Wiggins 2010; Wang, Cook, and others 2003; Wakefield, Smith, and Roberts 2010). There are excellent reasons for developing bespoke audio engines and for building on existing open-source technologies: they may be free as in beer and as in libre; integration can occur at a deeper level than is typically possible with closed-source software; it may be possible to focus on audio algorithms not traditionally part of DAW ecosystems; and a more programmatic environment may be more approachable and indeed familiar to the creator of a live-coding language.

Nevertheless, integrating with DAWs may provide significant advantages as well. First and foremost is the quality of synthesis algorithms available. For existing users of DAWs unfamiliar with musical programming languages, It is a much smaller leap to add generative capabilities to existing musical practice than to start over with built-from-scratch sounds as well as a brand new method of production. Integration provides an opportunity to consider live coding from the less-considered angle of the traditional multi-track music environment, bringing the "studio-as-instrument" concept to the stage. Conversely, it introduces far richer algorithmic capabilities to tools that a wide audience of users are already familiar with. The DAW considered in this paper, Ableton Live, itself originated more as a tool for live performance than non-realtime editing, however its algorithmic capabilities remain limited: a few clip trigger follow rules, cross-rhythmic periods of parameter automations, and some classic MIDI effects such as arpeggiators, velocity randomizers, chord generators, etc. Beyond immediate triggers, cues, and knob-twiddles, live algorithmic control multiplies the capabilities of the performer to concurrently manipulate larger portions of the total sound. Blackwell and Collins (2005) directly contrasted Ableton Live with a live-coding programming language in terms of cognitive dimensions, revealing pros and cons of both. Notably, the *abstraction hating* interface of Live is linked to its *closeness of mapping* to conventional studio equipment and stylistic bias "indicative of a corresponding reduction in potential for creative exploration". However it is also noted that it could be "exploited as a processing engine", and we suggest that through live-coding the DAW we might leverage the relative merits of both.

In this paper we describe research adding live-coding capabilities to Ableton Live, making extensive use of the plugin architecture and application program interface exposed in Cycling '74's Max/MSP/Jitter programming environment via the Max For Live toolkit. The sequencing capabilities of Ableton Live are not used per se, only the underlying timing transport, the plugin hosting, the mixer, and the parameter automation. Thus it remains trivial to, for example, insert a limiter into the master output channel to protect ears and loudspeakers, and to control the mixer via regular hardware fader banks, granting more immediate control over live diffusion than by typing code. Sequencing and parameter modulation however are provided by a text-based end-user programming environment, written in JavaScript and heavily inspired by the live-coding platform Gibber; we have named it *Gibberwocky* (see Fig. 1).[1] Although the scope of work

[1]It is available for download at https://github.com/charlieroberts/gibberwocky.

is currently limited to use in Ableton Live via Max For Live, we are considering broader potential impact with other DAW software.



```
a = Steps({
  [60]: '3.3f..4..8.5...f',
  [62]: '7.9.f4.....6f...',
  [64]: '........7.9.c..d',
  [65]: '..6..78..b......',
  [67]: '.f..3........f.',
  [71]: 'e.a.e.a.e.a.a...',
  [72]: '.............e.',
})

a[71].rotate.seq( 1,1 )
```

```
initializing...
gibberwocky is ready to
burble.
```

Latency: 0 samples --- /www/gibberMax/gibberwocky.amxd

Figure 1: *A screenshot of a Live set with Gibberwocky on a single track controlling an Impulse. A Steps object is outputting notes at different velocities while visual annotations depict its phase.*

## 2 Related Work

Two notable live-coding systems that enable use of commercial musical plugins are *Impromptu* (Sorensen and Gardner 2010) and *Wulfcode* (Ratliff). Impromptu provides support for Apple's AudioUnits framework inside of a custom audio engine, enabling fine-grained control of scheduling. Wulfcode is instead designed as a generic sequencer for any music application that accepts MIDI messages. Its syntax draws heavily from the *ixilang* live-coding environment, which also provides support for basic MIDI output in addition to integrating with the SuperCollider music programming system. The potential of live-coding in general purpose languages to overcome the concurrent inflexibility of graphical DAWs was also explored in Conductive (Bell 2011), which manipulates other music software via OSC messaging from within Haskell. In contrast to Gibberwocky the integration is more general-purpose (the examples given target the Jack audio router and again the SuperCollider synthesis engine) though some issues of latency and synchrony were observed.

In addition to the live-coding environments above that provide external control over music production systems or host commercial plugins, two notable live-coding systems have drawn inspiration from music production technologies to influence the design of their own live-coding systems. *Manhattan* by Chris Nash (2014) adds live-coding capabilities to the *tracker* genre of music production tools. His goals for Manhattan are similar to our own:

> The Manhattan project seeks to develop a unified environment supporting both contemporary and traditional music creativity, by flexibly combining low-level (concrete) and high-level (abstract) composition practices, in 'mixed-mode' works and workflows, supporting fusion and crossover music that bridges mainstream and avant-garde aesthetics.

We anticipate that many users of Gibberwocky will employ it to provide the abstract, 'avant-garde aesthetics' that Nash describes, while continuing to use Ableton Live to author the kinds of concrete representations of musical pattern that it excels at. While Manhattan enables end-users to perform algorithmic manipulations, its goals and functionality are different from Gibberwocky in that it is not intended for live-coding performance, and code is triggered via a timeline as opposed to immediate user action.

*EarSketch* (Mahadevan et al. 2015) is a browser-based environment that combines live-coding techniques with a multi-track editing view that is very similar to those found in commercial DAWs such as GarageBand and ProTools. EarSketch enables users to author in Python or JavaScript to insert music into the multitrack editor, which provides immediate visual feedback of the output of their code. EarSketch was not originally intended for live-coding performance (although it has been successfully used for this purpose by Jason Freeman (2016)), and requires users to recompile and execute entire programs after each modification. Gibberwocky also seeks to provide fine-grained visual information about how source code affects musical output through the use of code annotations (as described in Section 3.3) but focuses on affordances typical of live-coding environments, such as keyboard shortcuts to selectively execute blocks of code after they have been edited by users.

*LNXStudio* by Neil Cosgrove[2] is a DAW written entirely in SuperCollider. LNXStudio provides the ability to directly modify the source code of FX and instruments from within the DAW environment; we hope to provide similar functionality in Gibberwocky using the synthesis capabilities of gen~ in the future. Although LNXStudio doesn't currently provide an end-user API for live-coding performance, the author has expressed interest in adding live-coding affordances (personal correspondence, April 12, 2016).

There have been a number of projects incorporating audio domain-specific or general-purpose languages within plugin hosts, some with live-coding potential, however in general these were not designed to manipulate the host beyond the plugin itself.[3] There has also been a Node.js project utilizing the Max For Live bridge to control Ableton Live from browser-based clients, including via touch-based tablets[4], however it does not privilege temporal precision nor provide extensive parameter modulation.

## 3 Implementation

In this section we describe the current state of our system, discussing the engineering, end-user API, and source code annotations that make the system possible.

### 3.1 Terminology

- *plugin* - We use the term plugin to refer to the Gibberwocky Max For Live plugin (in Ableton's terminology, a *device*). The plugin contains a reference to Gibberwocky's singleton *server* object, and forwards MIDI messages to Live instruments and assigns values to device *parameters*.
- *server* - A single instance of the `ws` WebSocket external for Max/MSP/Jitter, which handles communication between *editors* and *plugins*.
- *editor* - Refers to a webview hosting a live-coding environment that sends WebSocket messages to the *server*. Each editor is typically associated with a single *plugin* instance; the server handles routing messages from the editor to the appropriate plugin.

### 3.2 Engineering

Gibberwocky uses custom Max/MSP/Jitter and Max For Live plugins alongside a browser-based editing platform and JavaScript API. Ableton Live is centered around the idea of *tracks*, in which each track represents either a virtual instrument or a collection of audio samples, that all run through a set of shared audio effects, sends, and gain / pan controls. One intended use case of Gibberwocky follows this track-based paradigm, with an instance of the Gibberwocky plugin running on each track of an Ableton Live set. Each plugin incorporates an editor in an embedded web viewer, suitable for smaller programs, but this editor can also be opened in a separate browser window for more space. In the end-user API (discussed further in Section 3.2) the track in which a plugin is inserted is denoted using the JavaScript `this` keyword. However, each instance of the plugin is also globally accessible from all other plugin instances, which means that a single editor can be used to control each track in an Ableton set that contains a plugin instance; we imagine this will be a typical scenario for live-coding performances.

Communication between browser and plugin is achieved via a websocket external for Max/MSP/Jitter that was developed to support this research[5], but more broadly enables any Max/MSP/Jitter plugin to easily communicate with web pages without the need for additional server software. The `ws` external opens a singleton socket server instance configured to a chosen port: all Gibberwocky plugins within an Ableton Live set thus share the same socket server.

[2] http://lnxstudio.sourceforge.net/index.html

[3] See for example http://www.osar.fr/protoplug/, http://worp.zevv.nl/live_coding.html, and https://github.com/pierreguillot/Camomile/wiki.

[4] The http://www.atmosphery.com/#fingz project, which is built upon https://www.npmjs.com/package/max4node.

[5] https://github.com/worldmaking/Max_Worldmaking_Package

This allows global sharing of information between multiple tracks, plugins/devices, and browser clients, as well as easy filtering of messages to specific tracks according to their unique ID.

Upon receiving a client websocket connection from an editor, the plugin responds by sending a JSON-encoded data structure representing the current Ableton Live set. For each track (including return tracks and the master track), the track name, unique id, track parameters, and a list of devices are given. For each device, the name, unique id, type (MIDI effect, instrument, or audio effect), and a list of parameters are given. For each device and track parameter, the name, unique ID, min/max value and quantization are given. Tracks parameters include volume, panning, and return send levels. Moreover, whenever the set is modified, such as by adding or removing devices, this dictionary is regenerated and rebroadcast to all connected clients. Tempo and time signature, and the current bar and beat, are also broadcast to all clients at connection time, and whenever they change. In this way, all connected editors retain an accurate current model of the Ableton Live set.

All scheduling information is transferred from browser-based editors to Gibberwocky plugin instances via WebSocket messages. These messages are then translated into various MIDI-like note data and modulation controls inside the Gibberwocky plugin. There is an artificially imposed buffer of one beat between the browser-based engine's outputs and the scheduler processing in the Gibberwocky plugin, to cover any network latency or client browser hiccups. At the start of each beat in Ableton Live, the message to update all connected browser clients with the new beat value serves as a request for the subsequent beat's worth of scheduling information. Any event messages received by the plugin in response to this update are added as appropriate to the plugin's audio-thread scheduler, implemented around Max/MSP/Jitter's `seq~` object. Even at a rapid "gabber" tempo of 200bpm this grants a very generous 300ms buffer – in practice we see communication over a localhost connection returned consistently well below 100ms. The events sequenced within the `seq~` object are then performed in accordance with signal-rate timing provided by Ableton Live via the Max For Live interface.

Ableton Live has for several years had a remote-control API, and although not officially documented, there have been a number of projects utilizing this API to control Ableton Live via Python, MIDI, and OSC.[6] However there is significant latency in using this API (at least 60ms for the Python bridge). The Max For Live toolkit exposes a stable, documented, and more performant version of this API. Still, most interactions with the API involve generalized messaging between high- and low-priority threads, incurring latencies of a few milliseconds. Moreover, device parameter setting is limited to a temporal resolution of 64 samples (around 1.5ms at a system sampling rate of 44.1kHz). The exceptions are signal-rate objects sample-synchronized to Ableton Live's transport, such as `plugphasor~`, and the `live.remote~` object, which can modulate Ableton Live device parameters at signal rate, ensuring much tighter timing than any other Ableton Live API method.

An important characteristic that distinguishes Gibberwocky from live-coding environments outputting MIDI is its integration of `gen~` with the `live.remote~` signal-rate parameter modulation. Briefly, `gen~` is a tool for meta-programming highly performant arbitrary audio graphs in Max/MSP/Jitter (Wakefield 2012), and its built-in operators are made available as modulation tools in Gibberwocky's end-user API. JavaScript representations of audio graphs are translated into `gen~` code that is then dynamically compiled to machine code within Max/MSP/Jitter, and can be assigned to any parameter in an Ableton set, enabling users to freely define audio-rate modulations with significantly better timing and (floating-point) resolution than the seven bit signals and lower baud rates of MIDI. Operator arguments in these `gen~` graphs can also be sequenced using the same syntax as other components of the Gibberwocky end-user API, as discussed in the next section.

## 3.3 End-User API

The end-user programming language for Gibberwocky is JavaScript, and its API is derived from abstractions for musical programming found in the online Gibber environment (Roberts, Wright, and Kuchera-Morin 2015b; Roberts et al. 2014). The API focuses on creating and manipulating musical *patterns* that are combined into musical *sequences*; these sequences schedule output to devices in the current Ableton Live set. A small set of methods can be sequenced to generate MIDI output:

- note - note can accept strings providing note names, accidentals, and octaves (such as 'c4' or 'Fbb5') or scale positions into a global scale object (0 would indicate the root of the current global scale)
- chord - output a cluster of notes simultaneously, notated by either a string indicating root and quality (such as `C#dim7`) or an array of integers indicating zero-indexed indices in the current global scale object. For example, given a current global scale of C Major, the chord description [0,2,4,6] would indicate a C major-7th chord (root, third, fifth and seventh).
- midinote - output a note message using a MIDI pitch (0–127) and the current velocity and duration

[6]For example https://code.google.com/archive/p/liveapi and http://livecontrol.q3f.org/ableton-liveapi/liveosc.

- midichord - output a cluster of notes simultaneously, notated as an array of MIDI pitches.
- velocity - set the velocity of subsequently outputted notes
- duration - set the duration of subsequently outputted notes

These methods can be easily sequenced by appending calls to .seq after their name. The two arguments to seq specify a pattern of values to output, and a pattern of timings. Below are examples of creating sequences:

```
// repeat the note c4 every quarter note
this.note.seq( 'c4', 1/4 )

// repeat an upward series of notes, alternating between quarter and eighth notes
this.note.seq( [ 0, 2, 4, 6 ], [ 1/4, 1/8 ] )
```

Arbitrary JavaScript functions can be used to generate either the output or the scheduling for any call to .seq. There are also convenience functions for randomization:

```
// randomly select an output of 0,3, or 5 every quarter note
this.note.seq( [ 0, 3, 5 ].rnd(), 1/4 )

// Randomly choose an integer between 0 or 15 to index the global scale
// Randomly trigger output using timings of either a 1/2, 1/4, or 1/8 note
this.note.seq( Rndi( 0, 15 ), [ 1/2, 1/4, 1/8 ].rnd() )
```

Multiple, independent sequences can be created to call a particular method by appending a unique ID number to the end of a call to .seq; if no number is appended, the default ID is 0. The example below creates independent melodic lines creating a polyrhythmic texture:

```
this.note.seq( [ 2,4,6,5,3 ], 1/8 )       // default id #0
this.note.seq( [ 7,9 ], 1/6, 1 )          // id #1
this.note.seq( [ 11,3,14,5,16 ], 1/5, 2 ) // id #2
```

Each sequence is stored as an array-indexed property of the method it is sequencing; the above example creates sequences at this.note[0], this.note[1], and this.note[2]. Independent access to each of these sequences enables them to be selectively started and stopped, and also provides a reference for pattern manipulation. Transformations to patterns can also be easily sequenced in Gibberwocky, enabling performers / composers to treat patterns in a serialist or evolutionary fashion, and live coders to tersely express complex sequences of musical output that vary over time. For any given sequence, the scheduling information is stored in its timings pattern, while the output is stored in its values pattern.

```
this.note.seq( [ 0, 4, 6, 9, 8, 7 ], 1/8 ) // a basic pattern of notes
this.note[0].values.transpose( 1 )  // transpose values by +1
this.note[0].values.transpose.seq( [1,1,-2], 1 ) // sequence transpositions to occur each measure
this.note[0].values.rotate.seq( 1,1 ) // rotate the pattern every measure
```

In addition to programming patterns, Gibberwocky comes with a number of pattern generators built-in, such as an arpeggiator (Arp) and a Euclidean rhythm generator (Euclid). These enable the quick generation of more complex patterns that are still subject to all the diverse pattern transformations that Gibberwocky offers. As a higher-level temporal construct, Gibberwocky provides a Score object enabling programmers to place functions on a timeline that will automatically be executed. The Score object can be started, stopped, rewound and paused at will; by default these actions occur in sync with the transport in Ableton Live. A step-sequencer, Steps, provides a quick way to sequence drum beats and other polyrhythmic patterns. Steps, Score, and Euclid all have visual annotations associated with them that are described in Section 3.3. Although there are Ableton Live and Max For Live plugins that possess similar functionality, the patterns that these plugins create cannot typically be manipulated algorithmically. For example, after the Euclid function generates a Gibberwocky pattern, that pattern can easily be transformed over time via the sequencing techniques described in this section (as shown in Figure 2); this is also true for patterns generated by the Arp and Steps objects. This enables users to easily define morphing patterns and avoid static repetition.

### 3.3.1 Accessing and Sequencing the Live Object Model

The Live Object Model (LOM) is generated when an instance of Gibberwocky is first instantiated, and subsequently re-generated whenever devices are added or removed from an Ableton Live set. The LOM contains a list of all parameters for every device and track in an Ableton Live set, assigning each a unique identifier. This LOM is accessible from within the code editors to assign values to parameters of Ableton Live devices, and to easily sequence them, as shown below. Note that Ableton Live parameters expect to receive values in the range {0,1}, which will be automatically mapped to their actual ranges; this makes it easy to re-use code for different parameters.

```
// store a reference to the impulse device on the plugin's track
impulse = this.devices['Impulse 606']

// set the value of Global Time property, which temporally stretches / compresses sample playback using
// granular techniques. All Ableton Live parameters expect a value in the range of {0,1},
// which is automatically mapped to their real range.
impulse['Global Time']( .25 )

//sequence the same property
impulse['Global Time'].seq( [.1, .25,.5,.75, .9], [1/4,1/2] )
```

### 3.3.2 Audio-rate Modulation Using gen~

The primitive functions and operators supported by gen~ have JavaScript representations in the end-user API of Gibberwocky. These functions can easily be combined to create graphs for modulation. For example, in the following code slithy returns a graph that represents the absolute value of an 8Hz sine wave:

```
slithy  = abs(cycle(8))
```

This is translated fairly directly into equivalent textual code for gen~, however any numeric arguments are replaced with named param objects so that they can be readily modified and sequenced after creation:

```
// the gen~ expression language equivalent of the above Gibberwocky code:
Param p0(8);
out1 = abs(cycle(p0));
```

A few additional functions can be used in the graph that expand into more complex gen~ code, such as the beats(n) function, which produces a ramp synchronized to a multiple (or division) of Ableton Live's meter. In the example below, we assume that an instance of the Gibberwocky plugin has been placed on a track in front of an instance of Ableton Live's "Impulse 606" sample playback device. We assign an LFO (low frequency oscillator) to control the transpose parameter of this device, which controls the pitch of sample playback via granular synthesis techniques. Finally, we sequence the LFO using the same syntax previously discussed in this section:

```
// store a reference to the impulse device
impulse = this.devices['Impulse 606']

// create a function that outputs a ramp in the range {0,1} at 0.15Hz and store a reference
gyre  = phasor( .15 )

// lfo() accepts frequency, amplitude (defaults to 1) and bias (defaults to .5) arguments
// create a compound parameter modulator using our previous ramp function
gymble = lfo( 2, 0.5, gyre )

// assign the modulator to the global transposition parameter of the Impulse device
impulse['Global Transpose']( gymble )

// sequence the first parameter of the cycle function in our lfo, which corresponds to its frequency
gyre[0].seq( [ .1, .5, 1, 2 ], 1 )
```

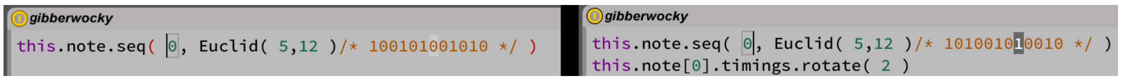
Of course LFOs can also be created that are far more complex than what is typically provided by a DAW – to the point where they approach the complexity of simple synthesizers:

```
vorpal = mod(add(mul(noise(), lfo(1, 0.01, 0.01)), max(beats(2), mul(lfo(2, 0.5, lfo(8)),2))),1)
```

### 3.4 Code Annotations for Audiences and Performers

Gibberwocky contains a number of code annotations providing visual feedback about the state of running algorithms, as first implemented in Roberts, Wright, and Kuchera-Morin (2015a). While our prior research on annotations focused on general purpose techniques that illuminated the behavior of a wide variety of patterns, our more recent research builds on this foundation to focus on solutions for specific types of patterns and scheduling objects.

In our first example we show the annotation for the `Euclid` pattern generator. The `Euclid` function returns a pattern that fits `n` pulses into `N` slots (Toussaint 2004). In the annotation we created for `Euclid`, a code comment displays the pattern of pulses and rests created by the `Euclid` function. This pattern can be transformed and manipulated using techniques such as *rotation* and *reversal.* The images in Figure 2 show a pattern generated by the `Euclid` function both before and after rotation.

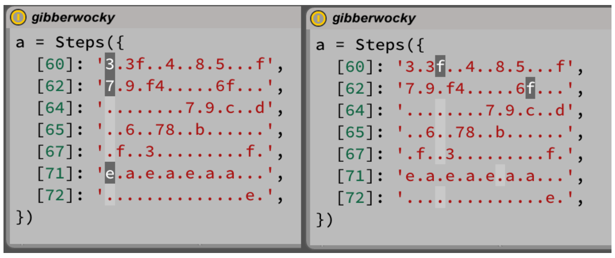


```
this.note.seq( 0, Euclid( 5,12 )/* 100101001010 */ )
```

```
this.note.seq( 0, Euclid( 5,12 )/* 101001010010 */ )
this.note[0].timings.rotate( 2 )
```

Figure 2: *The annotation for a pattern generated via the Euclid function, before and after rotation. Notice the highlighting that occurs when outputting a pulse (1) as opposed to a 0.*

A grey background block depicts the current phase of the `Euclid` pattern as it is called by its sequencer. When a value of `1` is output (indicating a pulse) the background briefly flashes pure black to provide a visual pop that accentuates the generated rhythms. The `Steps` object has a similar annotation to patterns generated by the `Euclid` function, where each sequence in the step sequencer is animated to show the current output, as shown in Figure 3. The values held in sequences of the `Steps` object can either consist of a period, which indicates a rest, or a hexadecimal value from 0—E which indicates the velocity of a note to be produced.



```
a = Steps({
  [60]: '3.3f..4..8.5...f',
  [62]: '7.9.f4.....6f...',
  [64]: '........7.9.c..d',
  [65]: '..6..78..b......',
  [67]: '.f..3.........f.',
  [71]: 'e.a.e.a.e.a.a...',
  [72]: '..............e.',
})
```

```
a = Steps({
  [60]: '3.3f..4..8.5...f',
  [62]: '7.9.f4.....6f...',
  [64]: '........7.9.c..d',
  [65]: '..6..78..b......',
  [67]: '.f..3.........f.',
  [71]: 'e.a.e.a.e.a.a...',
  [72]: '..............e.',
})
```

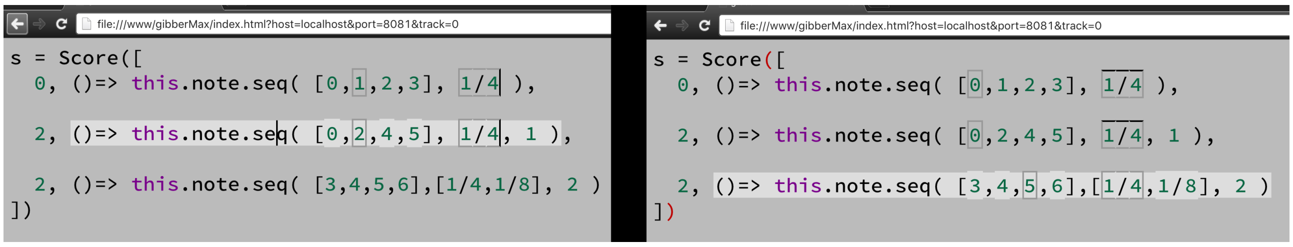
Figure 3: *Two snapshots of a step sequencer at different phases. Note that whenever a value is output it is highlighted briefly via white text on a black background. Also note that the various sequences comprising the step sequencer can have different behaviors attached to them; in this image the sequences for MIDI notes sixty-two and seventy-one are running at different rates than the other sequences.*

The annotation for the `Score` object indicates the most recently executed function in its timeline. When functions in the `Score` timeline contain additional objects with their own annotations, the creation of these annotations is delayed until their associated objects are instantiated. Thus, the `Score` object provides a visual history of its timeline, as shown in Figure 4.

## 4 Present and future potential

At present multiple clients on a local network may connect to a running instance of Ableton Live, providing immediate support for collaborative performance. The `ws` socket server incorporates a simple method to broadcast received messages to all other connected clients, which could form the basis of a shared data model or code document in the manner of Wakefield et al. (2014) or through connection with an *extramuros* server (Ogborn et al. 2015).

Beyond what has been described so far, the Ableton Live API also provides support for creating and modifying the content of MIDI clips in a track. Conceivably this may be useful for visualizing and capturing notes generated during



```
file:///www/gibberMax/index.html?host=localhost&port=8081&track=0
s = Score([
  0, ()=> this.note.seq( [0,1,2,3], 1/4 ),

  2, ()=> this.note.seq( [0,2,4,5], 1/4, 1 ),

  2, ()=> this.note.seq( [3,4,5,6],[1/4,1/8], 2 )
])
```

```
file:///www/gibberMax/index.html?host=localhost&port=8081&track=0
s = Score([
  0, ()=> this.note.seq( [0,1,2,3], 1/4 ),

  2, ()=> this.note.seq( [0,2,4,5], 1/4, 1 ),

  2, ()=> this.note.seq( [3,4,5,6],[1/4,1/8], 2 )
])
```

Figure 4: *Two snapshots of a score object at different points in its timeline. Annotations are enabled for each function in the score as it is triggered. In addition, the background of each function in the timeline is highlighted when it becomes the active function in the score.*

a performance; however it does not support creating parameter envelopes at present. A more serious limitation of the Ableton Live API is that it does not currently provide a method for instantiating new effects or instruments on a track, which means that the palette of instruments and effects in a performance must be defined in advance or modified through Ableton Live's graphical user interface during a performance. This might not be as great a limitation as it first appears: arguably the graphical user interface is a more expedient way to browse and deploy plugins, and in our experience live-coding performers tend to spend more time sequencing musical parameters than browsing for, or authoring, new synths and effects. That said, although in this paper we utilize gen~ for parameter automation, it could equally be used for general-purpose audio signal processing (audio effects and instruments) within Ableton Live, all completely under the control of live coding, if desired.

Given that much of the infrastructure for remotely sequencing events and modulating parameters of Ableton Live is complete, and mediated via a simple websocket protocol, this work readily facilitates the integration of other end-user APIs and domain-specific languages for controlling Ableton Live. Conversely, to go beyond the constraints of Ableton Live, we are keen to explore integration with other music production tools. Max/MSP/Jitter itself is an obvious candidate, as sequencing and scoring in Max/MSP/Jitter is an ongoing active area of research (Agostini and Ghisi 2015; Didkovsky and Hajdu 2008) and live coding languages excel in these areas. Additionally, our current Ableton Live integration depends heavily on Max/MSP/Jitter, so much of the development work for a Max-specific version is already complete. We anticipate that a Max/MSP/Jitter version of Gibberwocky would contain an API similar to that of the Ableton Live integration; however, instead of targeting parameters of tracks and devices in Ableton Live, the interface would control parameterized UI objects in a Max patch, as well sending direct messages to object outlets as defined within the JavaScript environment itself.

### 4.1 Acknowledgments

Supported by the Social Sciences and Humanities Research Council, and the Canada Research Chairs program, Canada. Also supported by the Rochester Institute of Technology, with funding from Sponsored Research Services and the Golisano College of Computing and Information Sciences.

## References

Aaron, Samuel, and Alan F Blackwell. 2013. "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*, 35–46. ACM.

Agostini, Andrea, and Daniele Ghisi. 2015. "A Max Library for Musical Notation and Computer-Aided Composition." *Computer Music Journal* 39 (2). MIT Press: 11–27.

Bell, Renick. 2011. "An Interface for Realtime Music Using Interpreted Haskell." In *Proceedings of the Linux Audio Conference*, 115–21.

Blackwell, Alan, and Nick Collins. 2005. "The Programming Language as a Musical Instrument." *Proceedings of PPIG05 (Psychology of Programming Interest Group)* 3: 284–89.

Didkovsky, Nick, and Georg Hajdu. 2008. "Maxscore: Music Notation in Max/msp." In *Proceedings of the International Computer Music Conference*, 483–86.

Freeman, Jason. 2016. "Live Coding with EarSketch." In *Proceedings of the 2nd Annual Web Audio Conference.*

Magnusson, Thor. 2011. “Ixi Lang: A SuperCollider Parasite for Live Coding.” In *Proceedings of the International Computer Music Conference*, 503–6.

Mahadevan, Anand, Jason Freeman, Brian Magerko, and Juan Carlos Martinez. 2015. “EarSketch: Teaching Computational Music Remixing in an Online Web Audio Based Learning Environment.” In *Proceedings of the 1st Annual Web Audio Conference*.

McLean, Alex, and Geraint Wiggins. 2010. “Tidal–pattern Language for the Live Coding of Music.” In *Proceedings of the 7th Sound and Music Computing Conference*.

Nash, Chris. 2014. “Manhattan: End-User Programming for Music.” In *Proceedings of the International Conference on New Interfaces for Musical Expression 2014*, 221–26. Goldsmiths, University of London.

Ogborn, David, Eldad Tsabary, Ian Jarvis, Alexandra Cárdenas, and Alex McLean. 2015. “Extramuros: Making Music in a Browser-Based, Language-Neutral Collaborative Live Coding Environment.” In *Proceedings of the First International Conference on Live Coding*.

Ratliff, Brendan. “Wulfcode: A Midi Live-Coding Environment for Performance or Composition.” https://github.com/echolevel/wulfcode.

Roberts, Charles, Matthew Wright, and J Kuchera-Morin. 2015a. “Beyond Editing: Extended Interaction with Textual Code Fragments.” In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 126–31.

Roberts, Charles, Matthew Wright, and JoAnn Kuchera-Morin. 2015b. “Music Programming in Gibber.” In *Proceedings of the International Computer Music Conference*, 50–57.

Roberts, Charles, Matthew Wright, JoAnn Kuchera-Morin, and Tobias Höllerer. 2014. “Gibber: Abstractions for Creative Multimedia Programming.” In *Proceedings of the ACM International Conference on Multimedia*, 67–76. ACM.

Sorensen, Andrew, and Henry Gardner. 2010. “Programming with Time: Cyber-Physical Programming with Impromptu.” *ACM Sigplan Notices* 45 (10). ACM: 822–34.

Toussaint, Godfried. 2004. “The Geometry of Musical Rhythm.” In *Discrete and Computational Geometry*, 198–212. Springer.

Wakefield, Graham. 2012. “Real-Time Meta-Programming for Interactive Computational Arts.” PhD thesis, Santa Barbara, USA: University of California Santa Barbara.

Wakefield, Graham, Charlie Roberts, Matthew Wright, Timothy Wood, and Karl Yerkes. 2014. “Collaborative Live-Coding with an Immersive Instrument.” In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 505–8.

Wakefield, Graham, Wesley Smith, and Charles Roberts. 2010. “LuaAV: Extensibility and Heterogeneity for Audiovisual Computing.” In *Proceedings of Linux Audio Conference*.

Wang, Ge, Perry R Cook, and others. 2003. “ChucK: A Concurrent, on-the-Fly Audio Programming Language.” In *Proceedings of International Computer Music Conference*, 219–26.