gibberwocky: New Live-Coding Instruments for Musical Performance

Charles Roberts
School of Interactive Games and Media
Rochester Institute of Technology
Rochester, USA
cdrigm@rit.edu

Graham Wakefield
School of the Arts, Media, Performance and Design, York University
Toronto, Canada
grrrwaaa@yorku.ca

ABSTRACT

We describe two new versions of the gibberwocky live-coding system. One integrates with Max/MSP while the second targets MIDI output and runs entirely in the browser. We discuss commonalities and differences between the three environments, and how they fit into the live-coding landscape. We also describe lessons learned while performing with the original version of gibberwocky, both from our perspective and the perspective of others. These lessons informed the addition of animated sparkline visualizations depicting modulations to performers and audiences in all three versions.

Author Keywords

Live coding, Max/MSP, MIDI

ACM Classification

H.5.5 [Information Interfaces and Presentation] Sound and Music Computing, H.5.2 [Information Interfaces and Presentation] User Interfaces, D.2.6 [Software Engineering] Programming Environments.

1. INTRODUCTION

Designing environments for live-coding performance that target external applications requires balancing meaningful integration with systems that are often open-ended (such as Max/MSP or SuperCollider), with idiomatic constraints that aid performance. We recently introduced a live-coding environment, *gibberwocky.live*[6], featuring a high level of integration with Ableton Live.[1] In response to the requests of users, we have since created two new live-coding environments, *gibberwocky.max* and *gibberwocky.midi*, that provide similar performance affordances while targeting an alternative music programming environment (Max/MSP) and communication protocol (MIDI). This paper describes these two new environments and the various design considerations that informed their creation. We also describe experiences performing with *gibberwocky.live* and how these performances affected the development of the new systems presented here. Of particular note is a new visualization system affording animated sparklines that visually depict audio modulations over time, described in Section 5.2.

[1]This version was previously simply named gibberwocky; we changed the name to differentiate from the two new environments described in this paper.

NIME'17, May 15-19, 2017, Aalborg University Copenhagen, Denmark.

2. BACKGROUND AND MOTIVATION

Designers who create live-coding environments targeting external synthesis applications can focus on language development and instrument design instead of low-level DSP algorithms. Many such environments instead place a heavy emphasis on the creation, playback, and transformation of musical pattern [3, 2, 4], influenced both by innate affordances of the computer to manipulate sets as well as serialist techniques that evolved over the course of the 20th century. Our systems adopt a similar approach and feature a variety of tools for pattern creation and manipulation.

However, since we began our development of gibberwocky targeting Ableton Live, we also wanted to take advantage of its specific capabilities. In our analysis, one important component of Live is the quality of the instruments it includes, and their ease-of-use in live performance. Physically manipulating their interfaces yields interesting changes in the sonic character of instruments and often becomes an important component of performances. Accordingly, we wanted to ask how we could perform similar musical gestures algorithmically, enabling live coders to focus on the keyboard interface without having to continuously manipulate the mouse, trackpad, or other external controller devices.

Our interest in using musical gesture led to an emphasis on continuous modulation in gibberwocky. Declaring modulation graphs, changing them over time, and creating visualizations depicting their state are all heavily prioritized. The dual-emphasis of pattern manipulation and modulation positions gibberwocky somewhat uniquely among live-coding environments.

The primary differences between the various flavors of gibberwocky are the applications that are controlled, and the deep level of integration that is achieved with each. We provide access to a well-defined object model in both gibberwocky.max and gibberwocky.live enabling users to begin controlling most parameters immediately, without setting up complex OSC and MIDI routing schemas, as described in the Section 3. We also provide integration with the Gen framework included in Max/MSP, enabling users to define modulation graphs in the live-coding editor that are subsequently transferred, compiled, and run in Live or Max.

Besides targeting different applications and the API differences this incurs, we deliberately made as many aspects identical between the various flavors of gibberwocky as possible, to help ensure easy movement of practices between the environments. All versions use the same API for sequencing, harmony, and musical pattern manipulation, much of which was adopted from the end-user API found in Gibber [8]. All three environments also share specialized sequencer objects, such as an arpeggiator (`Arp`), a step-sequencer (`Steps`) and a timeline for executing anonymous functions (`Score`). And they all employ an annotation system that modifies source code to reveal system state and the output of generative functions. Although many of these annotations were first explored in Gibber [7] some are new additions created specifically for gibberwocky, and we have continued to add new affordances for feedback as part of the research presented in this paper (see Section 5.2).

3. GIBBERWOCKY.MAX

The global community of live coders is rapidly growing. The past five years have seen multiple international conferences devoted to the practice in addition to almost a hundred Algorave events [1], and numerous individual performances that took place in the context of more broadly themed electroacoustic concerts. This rise in popularity lead to a growing number of environments for live-coding, with four being introduced at the last International Conference on Live Coding alone. But despite the plurality of live-coding environments, the use of visual programming languages in live-coding performances remains relatively rare. While there is certainly more research to be done in this area, for now interfaces focusing on text, featuring typing as the main input modality, remain the dominant paradigm.

We suggest that some of this is due to the speed and fluidity of defining sequences and generative patterns using keyboard interfaces alone. Even without considering a task-oriented analysis of text-based environments versus graphical patching environments, a simple application of Fitt's law shows that alternating between interface elements (keyboard vs. mouse/trackpad) occupying different physical spaces causes a loss of efficiency as a function of distance, and this does not take into account the potential cognitive burden of shifting between these different input modalities. One of the goals of gibberwocky.max is to isolate many of the tasks required for live coding in Max/MSP to a constrained textual interface that does not require live patching (or significant use of the mouse/trackpad) during performance. At the same time, Max/MSP has a great deal to offer the live-coding community, with a rich history of audiovisual instrument design, experimental objects for sound synthesis, and an efficient low-level language for dynamically creating audio graphs, Gen [11]. We hope that integrating Max/MSP with a textual live coding interface will provide a satisfying new way to explore it, and conversely believe that live-coders will be attracted to the rich palette of sounds (and visuals) that Max/MSP provides.

3.1 Installation and Setup

Gibberwocky.max is distributed using Max's Package format, making it simple to download and install. Once installed, any Max patch can be exposed for live coding by adding a `gibberwocky` Max object to the patcher. The `gibberwocky` object has a single message outlet as well as a number of signal outlets determined by the `@signals` attribute.

Sending a "bang" message to the `gibberwocky` object will open up the client Gibberwocky editor in the system's default web browser. Alternatively, the editor can be accessed online, which permits remote collaboration over a local network, by specifying the IP address of a single host machine running Max/MSP.[2] This permits multiple users to cooperatively live code Max concurrently, and only requires one instance of the `gibberwocky` object instantiated in Max.

3.2 The Scene

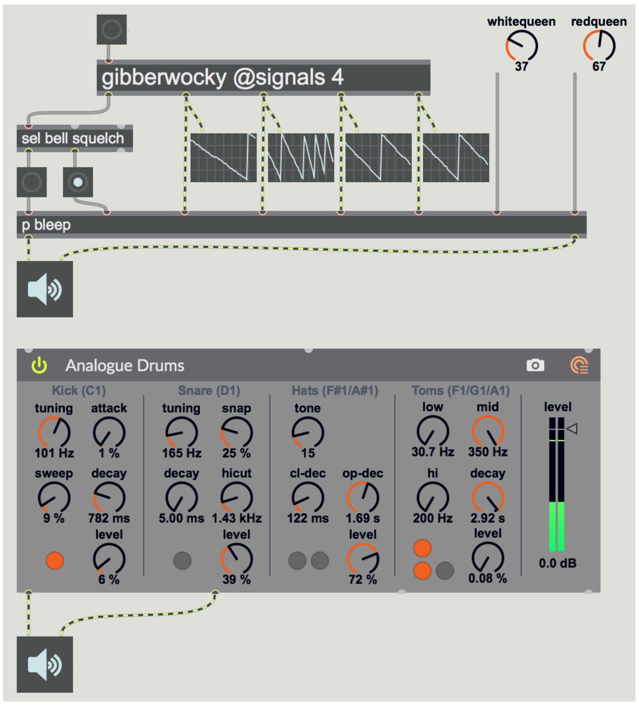


Figure 1: The gibberwocky Max object, with its messaging output routed to a sound-generating subpatcher ("bleep", via "bell" and "squelch") along with the signal outlets, which are also visualized. The gibberwocky object also communicates indirectly with user interface objects ("whitequeen" and "redqueen") as well as an embedded Max for Live device ("Analogue Drums").

Gibberwocky.max permits the user to live code arbitrary sequences and assign modulations to parameters of the Max patcher. This is done through the exposure of four global objects in the gibberwocky.max client: `devices`, `signals`, `params`, and `namespace`. These objects constitute the "scene", a representation of the live scriptable components of the user's patcher that is derived by analysis of the patcher's content.

This scene is derived first when the `gibberwocky` object in Max is created, and sent to every client subsequently that connects to this object. It is then displayed as a treeview, shown in Figure 2, that enables users to drag any leaf into code editor in order to insert the absolute path to a given parameter or instrument. For example, after dragging the leaf "ad-level" shown in Figure 2 into the editor the following source code will be inserted:

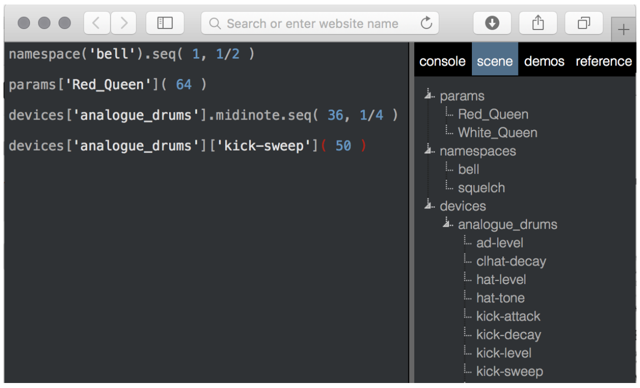
```
devices['analogue_drums']['ad_level']
```

Auto-complete can also be used to quickly enter paths to parameters in the scene. The scene is derived again and broadcast to all clients whenever the user's patcher is saved, in order to reflect any new additions or changes to the patcher in the end-user API. Several different methods are used to identify scriptable components of the patcher as detailed below.

3.3 Messaging namespace

The `namespace` is populated with known message routing names connected to `gibberwocky` in the patcher. These include any `route`, `routepass`, or `select` object that is either directly connected to, or appropriately chained to, the `gib-`

[2]For example: `http://gibberwocky.cc/max/?host=<IPaddress>`



```
namespace('bell').seq( 1, 1/2 )

params['Red_Queen']( 64 )

devices['analogue_drums'].midinote.seq( 36, 1/4 )

devices['analogue_drums']['kick-sweep']( 50 )
```

Figure 2: On the right, a treeview displaying the scene representing the patch shown in Figure 1 in the gibberwocky.max client. Users can drag and drop leafs from the tree into the code editor to insert paths for targeting specific devices / parameters, as seen at left.

`berwocky` object's leftmost (message) outlet (see Figure 3). This accounts for the most common basic methods of direct message routing in Max patching.

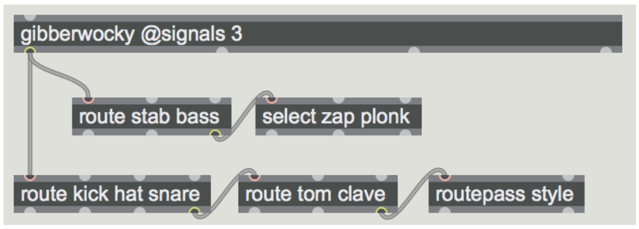


Figure 3: An example of routing messages (as opposed to signals) from the gibberwocky object for Max/MSP. When the patcher is saved, all of "stab", "bass" , "zap", "plonk", "kick", "hat", "snare", "tom", "clave", and "style" will be automatically included in the client's derived `namespace`.

In the end-user API, namespaces are created with a call to the `namespace` function. A single string is passed as the argument to this namespace. Any arbitrary member of that namespace can subsequently be sequenced.

```
ns = namespace('synth1')

// send synth1 1 every half note:
ns.seq( 1, 1/2 )

// send 'synth1 foo 0' every half note:
ns.foo.seq( 0, 1/2 )

// send list 'synth1 foo bar baz 0'
// every half note:
ns['foo bar baz'].seq( 0, 1/2 )

// alternately send 'synth1 foo bar 0 1'
// and 'synth1 foo bar 2 3':
ns['foo bar'].seq( [[0,1],[2,3]], 1/2)
```

Although available namespaces are populated in the treeview depicting the current scene and are also discoverable via autocomplete, performers can start any arbitrary sequences of messages and subsequently perform the appropriate patching in Max. There is no requirement that a routing must exist in a patch for a particular message before that message can be sent from the client.

3.4 Parameters

Another common idiom in Max is to expose user interface (UI) objects to Max's parameter system ("pattr"), and other forms of scripting and modulation, by assigning a unique identifier that is commonly referred to as the "scripting name". Conveniently, some UI objects (such as `live.slider`, `live.dial`, etc.) receive scripting names automatically when created, and retain well-defined parameter types and ranges. All such named parameters found in the patcher are added to the scene via the `params` array, and can also be found in the drag-and-drop interface of the scene browser of the gibberwocky.max client.

3.5 Devices

The scene description sent by Max to the gibberwocky clients also includes an array of named `devices`, containing every Max for Live device found. Each device represented in the array contains information about all the parameters it exposes for control, and these can be easily manipulated using the gibberwocky.max interface and API.

In addition to enabling control of instrument parameters, gibberwocky also provides a simple way of sending MIDI note messages to instruments without requiring any patching or routing in the Max/MSP patch. An example of the end-user API for both sending MIDI messages and control messages to a Max for Live device is given below, and assumes the instantiation of a Max for Live device given the scripting name "drums" by a user; default unique identifiers are also provided.

```
// store reference to Max for Live device
drums = devices['drums']

// set the kick-sweep parameter to 50
drums['kick-sweep']( 50 )

// sequence kick-level parameter (in %)
drums['kick-level'].seq( [10,50,75], 1/4 )

// send MIDI note messages using provided
// durations and velocities
drums.duration( 125 ) // ms
drums.velocity.seq( [64,127], 1/4 )
drums.midinote.seq( [36,38], 1/8 )
```

3.6 Audio signals

The scene's `signals` is an array of functions, with each function corresponding to one of the signal outlets of the `gibberwocky` object. When a modulation expression is passed to one of these functions, a message is sent to the Max object that creates an audio function via Gen, which is routed out of the corresponding signal outlet of the `gibberwocky` object as a regular MSP audio signal.

The following line of end-user code generates a gen expression creating a phasor with its output scaled by .5, and then sends the expression to gibberwocky.max so that the resulting output will be mapped to the second outlet of the `gibberwocky` object (i.e., the first audio outlet):

```
signals[0]( mul( phasor(2), .5 ) )
```

In the same manner as gibberwocky.live, any parameter of a gen expression can be sequenced:

```
_phasor = phasor( 20 )
_scale = mul( _phasor, .5 )
signals[0]( _scale )

// sequence first parameter of phasor ugen
    (frequency)
_phasor[0].seq( [20,40,60], 1/2 )
```

```
// sequence second operand of mul ugen
_scale[1].seq( [.5,.25,.1,.05], 1/16 )
```

Assigned Gen expressions broadcast "snapshots" of their current values to all connected clients, approximately thirty times per second, for the purposes of rendering the sparklines in the client editors. If a signal expression is unassigned, these snapshots are no longer broadcast and the audio signal in Max holds the last computed value as a constant.

While gibberwocky.live uses Gen to create modulation signals for controlling parameters of Ableton Live, in gibberwocky.max they can be used both for modulation or for directly synthesizing arbitrary audio graphs. In short, these Gen graphs can be used as entire synthesis engines for instruments in addition to modulation sources, creating the possibility of hybrid live-coding performances involving both high-level control of predefined instruments as well as the low-level creation of DSP algorithms for experimental sound synthesis.

3.7 Timing

Gibberwocky.max synchronizes communication to the browser according to Max's "Transport", a global timing source oriented to musical meter directly integrated with a wide range of Max and MSP objects. Gibberwocky shares any changes to timing state, including beats per minute, time signature, the current bar and beat index, and the playing status, with all connected clients.

In a similar fashion to gibberwocky.live this transport information is used to both request messages from the client and to drive source code annotations and visualizations in the client code editor. By default, gibberwocky.max requests messages from the client one beat in advance, providing a window for messages to be transferred and parsed on local area networks. On each beat gibberwocky.max requests new events for the next beat from all connected clients. When the client receives a request for messages spanning the next beat, it asynchronously calculates the corresponding messages and sends them back to Max along with phase offsets.

For example, after receiving a request for the next beat (beat number four) a message to schedule a "midinote" event halfway through the beat, would be sent over WebSockets using the following ASCII text:

```
"add 4.5 midinote foo 64 127 2000"
```

When the gibberwocky object within Max receives this message, it stores and schedules the message `"midinote foo 64 127 2000"`, to be sent at the precise time of 4.5 beats, by means of the `seq~` object in Max. (This particular scheduled message will send a MIDI NoteOn and NoteOff event pair spanning a duration of 2000 milliseconds, at pitch 64 and velocity 127, to the parameter or device destination `"foo"`.)

4. GIBBERWOCKY.MIDI

gibberwocky.midi outputs MIDI NoteOn, NoteOff and Control Change messages. It syncs to MIDI Clock messages in order to integrate with digital audio workstations, but can also use its own internal clock for timing, which is potentially useful for controlling external MIDI hardware.

The setup for gibberwocky.midi is simple. Users enter the gibberwocky.midi URL[3] in any browser that supports the WebMIDI protocol (at the time of this writing this includes Google Chrome and Opera); no additional software is required. After selecting an output port for MIDI messages the software is ready for use.

One notable difference between gibberwocky.midi and the other gibberwocky projects is the lack of Gen integration, as Max/MSP and the gen~ object are not part of its toolchain. However, we have partially overcome this limitation by integrating a JavaScript port of Gen, *genish.js*, created by the first author[4]; this affords authoring modulation graphs using an (almost) identical syntax. The main difference is that the graphs are run at control-rate in the browser instead of at audio-rate in the target application. The resulting modulations can be assigned to any MIDI CC message as described in 4.1. A global sampling rate, which defaults to 60 Hz, controls the rate of modulation output messages and is freely user-definable.

Because we are using genish.js to send MIDI CC messages instead of creating audio signals, all graphs created are wrapped in ugens that transform the resulting signal to a range of 0–127 by default. As part of the API this scaling and translation can optionally be disabled; however, users must then be careful to ensure that their graphs only generate signals consisting of seven bytes of integer data (values ranging from 0–127).

4.1 End-User API

Messages in gibberwocky.midi are sent through a global channels array. Each channel in this array sends messages to an associated MIDI channel on a MIDI port selected in the gibberwocky.midi GUI.

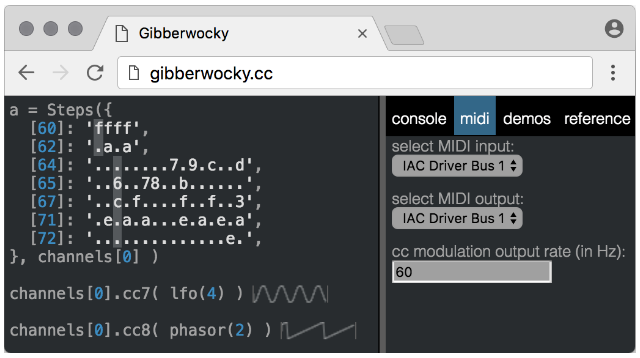
```
// send note messages to channel 1
// using a zero-indexed array
channels[0].midinote( 64 )

// use global scale to determine midinote
Scale.root( 'c2' )
Scale.mode( 'phrygian' )
channels[0].note.seq( [0,1,2,3], 1/8 )

// sequence values to cc7
channels[0].cc7.seq( [32,64,96,127], 1/2 )

// route modulation to cc8
channels[0].cc8( lfo(.5) )

// disable scaling and translation
channels[0].cc9(
  round( phasor(2,0,{min:0, max:64} )),
  false // do not scale / transform
)
```



```
a = Steps({
  [60]: 'ffff',
  [62]: '.a.a',
  [64]: '........7.9.c..d',
  [65]: '..6..78..b......',
  [67]: '..c.f....f..f..3',
  [71]: '.e.a.a...e.a.e.a',
  [72]: '..............e.',
}, channels[0] )

channels[0].cc7( lfo(4) )

channels[0].cc8( phasor(2) )
```

select MIDI input: IAC Driver Bus 1

select MIDI output: IAC Driver Bus 1

cc modulation output rate (in Hz): 60

Figure 4: The gibberwocky.midi interface.

5. PERFORMING WITH GIBBERWOCKY

[3] `http://gibberwocky.cc/midi`

[4] `http://charlie-roberts.com/genish/`

We have given a half-dozen performances with gibberwocky to date, in settings ranging from juried conference events to informal lectures / demonstrations. These experiences directly impacted the design of all gibberwocky environments, most notably with the introduction of sparkline visualization discussed in Section 5.2. In addition to describing our personal experiences performing with gibberwocky, this section also draws from an interview with Lukas Nowok, who gave a pair of gibberwocky performances over the last six months.[5]

5.1 Differences From Other Environments

One significant difference between the various gibberwocky environments and some other live-coding systems is that instruments and effects need to be instantiated before a performance begins. While it is certainly possible to move between the live-coding interface and Live to add new instruments and effects, in practice keeping track of both interfaces and bouncing between them is distracting and time-consuming while performing. In gibberwocky.max it is especially problematic to patch during a performance, as changes to the topology of audio graphs in Max/MSP typically result in a brief audio dropout. Although these problems directly impact the potential of these environments for live-coding performance, they do not affect their usefulness when using live-coding techniques to compose or experiment. But for performances they do have the effect of constraining sonic palettes to a pre-defined palette of sounds. Coming from live-coding environments lacking this constraint, we found it limiting.

Despite the constraint of having to instantiate instruments and effects ahead of time, the quality of the DSP algorithms and instruments opens new opportunities for exploration, which is one of the original motivations for all gibberwocky environments. Whether or not the quality and range of sounds available in Live, Max/MSP, and MIDI-controlled hardware and software outweighs the ability to easily instantiate new synthesizers and effects during performance is a question best decided by individual performers.

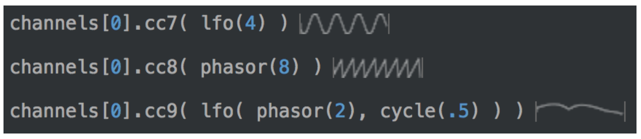
Nowok notes that the musical output of his performances with gibberwocky were perhaps not significantly different than what he would have created using Ableton Live alone. He added:

> But the process of getting to that outcome is more natural and expressive than linear representations like the pianoroll, the arrangement view or the clip view. That might depend largely of my personal idea of music though, as I express structure and form through continuous and flowing sound and more in terms of contour than in terms of beats and bars. That means I don't use the pitch and rhythm sequencing of gibberwocky (a lot). gibberwocky speaks to my idea of music more than any other environment, in that it suggests continuous, flowing and precise control of sound parameters and complex layering of these modulations (again, that is probably very individual and only true for my approach).

5.2 Sparkline Visualizations

One strategy we found for performing with gibberwocky.live is to spend time at the beginning of the performance sequencing musical material, and then subsequently tweak synthesis and effect parameters via modulations created using `gen~` graphs. As previously mentioned, gibberwocky.live contains a variety of dynamic annotations that visually display the progression of musical sequences and the results of various algorithmic process. In our opinion this transforms source code documents into both a valuable source of feedback for the performer and a source of information and spectacle for the audience [5]. However, when our performances shifted to signal processing and modulation instead of musical sequencing this feedback was lost; the resulting code seemed dull and lifeless in comparison to the animated and evolving character of earlier sections.

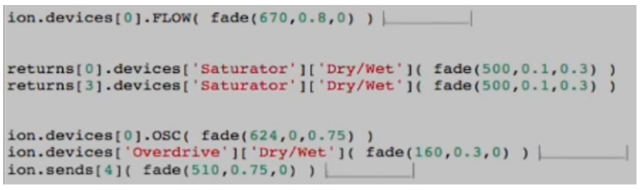
To improve this we implemented animated sparkline visualizations [10] adding feedback for both performers and audience members; these sparklines are now present in all three gibberwocky environments. The sparklines appear alongside the code fragments responsible for creating their corresponding modulation graph; in this fashion they are similar to research conducted in the Impromptu and Extempore live-coding environments by Swift et al. [9]



```
channels[0].cc7( lfo(4) )
channels[0].cc8( phasor(8) )
channels[0].cc9( lfo( phasor(2), cycle(.5) ) )
```

Figure 5: Three sparklines depicting modulations assigned to various MIDI CC messages in gibberwocky.midi.

Although informal feedback about the sparklines has been positive, there is work to be done concerning both the timescale and the range of values that sparklines in the gibberwocky environments display. For example, in gibberwocky.live the Max for Live API requires that all signals assigned to parameters be in the range of 0–1. This makes it much easier to implement effective sparklines as we always know the maximum range of values possible. However, even when focusing on this limited range smaller micro-modulations often become imperceptible in the sparklines, despite possibly having a large impact on the final rendered audio. There are also problems at larger temporal scales. One of the most common uses for modulations is simple fades of audio parameters; however, these fades often take place over long periods of time, while the sparklines in gibberwocky, as currently implemented, only display the one second of sampled output at any moment.



```
ion.devices[0].FLOW( fade(670,0.8,0) )


returns[0].devices['Saturator']['Dry/Wet']( fade(500,0.1,0.3) )
returns[3].devices['Saturator']['Dry/Wet']( fade(500,0.1,0.3) )


ion.devices[0].OSC( fade(624,0,0.75) )
ion.devices['Overdrive']['Dry/Wet']( fade(160,0.3,0) )
ion.sends[4]( fade(510,0.75,0) )
```

Figure 6: A partial screenshot from a performance by Lukas Nowok. Note the slow fades, lasting dozens of measures, and the resulting flat sparklines.

The result during long fades is a horizontally flat line that gradually rises or falls, which is ineffective in revealing the overall musical gesture of the modulation, as shown in Fig 6. Resolving these issues on both the horizontal and vertical axes by dynamically changing scale in response to the displayed signal is an interesting subject for future research. Nevertheless, even as currently implemented they do still

[5]https://www.youtube.com/watch?v=gl4JLCkbrfU

provide an indication of activity in these situations, as Nowok notes:

> ...in my performances it usually takes some time for the sound to react to changes/executions in the code (fades over many minutes, oscillators with a frequency of under 0.01Hz etc.). Maybe it is just the immediate reaction of the visual representation in the gibberwocky editor that makes it more understandable?

Remarking further on the importance of the annotations and sparklines in Gibberwocky, Nowok states:

> The beautiful thing about gibberwocky is that the code with the visualizations is a close representation of a musical situation at every moment — the distance between the notation and the outcome is small which isn't the case for other notations like SuperCollider for example. And this makes it much easier for an audience to understand the relation between notation and music and in turn the role of the performer. That was very apparent to me when I played a live coding performance with SuperCollider last week in Tallinn. The musical aesthetic and overall 'feel' was very similar to the performance in Helsinki that I did with gibberwocky but the feedback I got was drastically different in that people in Tallinn couldn't follow the unfolding of the musical structure and connect it with what I was writing. I got many questions asking if I was playing back prerecorded material.

6. CONCLUSIONS

We presented two new live-coding environments, gibberwocky.max and gibberwocky.midi. gibberwocky.max provides deep integration with Max/MSP, including dynamically creating Gen graphs, snooping patcher objects in order to determine valid messaging targets, and the ability to access many targets with no visual patching required. gibberwocky.midi attempts to provide a similar experience to gibberwocky.max and its predecessor, gibberwocky.live, but solely outputting MIDI messages and with no reliance on Max/MSP. By using a new software library, genish.js, that emulates the Gen library, the entire environment runs in the browser with no additional software required. All three environments feature a new visualization system that depicts the output of modulation algorithms with time-varying sparklines, inline with the code that creates the algorithms in the browser-based editor; this feature was informed by our experiences performing with the gibberwocky system.

Opportunities remain for improved integration with Max/MSP. For example, many instruments contain parameters designed to be controlled by messages; these parameters do not enable users to control them with audio signals in the manner that Max for Live instruments do. We have begun work on a port of Gen to regular Max objects that can be used to generate message appropriate for controlling such parameters. Other opportunities include improving the algorithms used for detecting potential messaging targets, such as auto-discovering the parameters of VST and AudioUnit plugins hosted in Max.

The sparkline visualizations used in all gibberwocky environments are an interesting area for future research. In addition to resolving issues of scale across both time and value, many modulations would be better served by visualizations that depict progress through an overall gesture as opposed to a simple history of previous values.

Finally, there is also the potential for integration with more platforms, such as Pd or Bitwig Studio. As we create further integrations, a general refactoring will need to be performed so that all integrations use a single environment supporting a variety of targets; such an environment could also provide an exciting opportunity for controlling a range of software platforms from a single live-coding environment concurrently, using a API that is unified whenever possible.

7. 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.

8. REFERENCES

[1] *https://algorave.com*, 2013 (accessed January 29th, 2017).

[2] R. Kirkbride. FoxDot: Live Coding with Python and SuperCollider. In *Proceedings of the International Conference on Live Interfaces*, 2016.

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

[4] 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.

[5] C. Roberts. Code as information and code as spectacle. *International Journal of Performance Arts and Digital Media*, 12(2):201–206, 2016.

[6] C. Roberts and G. Wakefield. Live Coding the Digital Audio Workstation. In *Proceedings of the 2nd International Conference on Live Coding*, 2016.

[7] C. Roberts, M. Wright, and J. Kuchera-Morin. Beyond Editing: Extended Interaction with Textual Code Fragments. In *Proceedings of the New Interfaces for Musical Expression Conference*, 2015.

[8] C. Roberts, M. Wright, and J. Kuchera-Morin. Music Programming in Gibber. In *Proceedings of the International Computer Music Conference*, pages 50–57, 2015.

[9] B. Swift, A. Sorensen, H. Gardner, and J. Hosking. Visual Code Annotations for Cyberphysical Programming. In *1st International Workshop on Live Programming (LIVE)*, pages 27–30. IEEE, 2013.

[10] E. Tufte. *Beautiful Evidence*. Graphics Press, Cheshire, CT, 2006.

[11] G. Wakefield. *Real-Time Meta-Programming for Interactive Computational Arts*. PhD thesis, University of California Santa Barbara, 2012.