Closing the Circuit: Live Coding the Modular Synthesizer

Shawn Lawson
Rensselaer Polytechnic Institute
lawsos2@rpi.edu

Ryan Ross Smith
Rensselaer Polytechnic Institute
ryanrosssmith@gmail.com

Frank Appio
frankappio@gmail.com

ABSTRACT

This paper extends upon the real-time audio-visual collaboration that the authors have engaged with since 2014. Previous iterations of this collaboration have focused on formalizing a structured response-relationship between the audio and visual components, but more recently, the integration of improvisatory live coding and modular synthesis techniques have enabled an increasingly responsive and chaotic feedback loop between the audio and visual components. This paper will provide an overview of the technical details of the AppiOSC, a custom device designed to facilitate streams of correspondent yet unpredictable bi-directional communication data between The Force, a live coding environment for graphics, and the modular synthesizer.

1 Introduction

Live coding is a quickly growing field of study that positions the otherwise solitary practice of writing computer code for evaluation only upon its execution as a real-time REPL (read, evaluate, print, loop) process.(Sorenson 2014; D. Griffiths. Fluxus. In A. Blackwell and J. Rohrhuber 2013; Nick Collins and Ward 2003) In order to facilitate an immediacy of response, as is typical of conventional musical performance (ie. one hears a tone the moment a piano key is pressed), the REPL cycle is temporally compressed to be virtually instantaneous. The live coding of audio is perhaps the most common instantiation of these practices, and is enabled by a wide range of integrated development environments (IDEs) and languages. (Alex McLean 2016; Sam Aaron; "SuperCollider"; Ge Wang) In these environments, the *Print* portion of the REPL is sound, although these processes are equally applicable to the real-time generation of graphics.

The live coding of graphics has been far less explored, although historically has been in existence for nearly as long. (Griffiths 2013; Casa and John 2014; Quilez) These IDEs receive and evaluate an incoming audio signal, and transform this signal into accessible data streams for the real-time manipulation of graphics. The integration of visual and sonic live coding into a single IDE has also been explored previously with a variety of base languages. (G. Wakefield and Roberts 2010; C. Roberts and Kuchera-Morin 2012; McKinney 2014; Sorensen 2005; "Shadertone") Dual audio-visual IDEs include direct access to either variable information used for sound generation, or use Fast Fourier Transform (FFT) to extract data from an incoming audio signal in order to manipulate and modify the graphics.

2 Integration

In this section we discuss the existing communication link between visual and audio performers, followed by an overview of the new communication additions and potentials facilitated by the AppiOSC hardware interface.

2.1 Existing Interface

The Force[1] was initially designed to extract audio information through FFT analysis of an external signal, including live audio, fixed audio files, and web streams. By default this audio is passed through a webaudio FFT node and summed into four bins. In order to enable quick access while live coding, the four bin values are passed into the shader through a vector 4 uniform. The information extracted from each FFT band can then be used to (in)directly influence the graphics, creating a malleable link between the audio signal and its *representation* on screen.

[1] https://github.com/shawnlawson/The_Force

Table 1: Frequency bin distribution for *bands* uniform.

GLSL uniform variable	Frequency
bands.x	0 - 215Hz
bands.y	216Hz - 474Hz
bands.z	475Hz - 1034Hz
bands.w	1035Hz - 22,050Hz

As seen in the table, each band is accessible by components x, y, z, w as scalers. In the following example,

```
vec3 aRedColor = vec3(1, 0, 0) * bands.x;
```

the vec3 datatype aRedColor is modulated by the relative intensity of band x, the lowest range of audio frequencies. If aRedColor is used as a color with the vec3 components of red, green, and blue, then there is a direct and perceptible linkage between low frequency sound and intensity of aRedColor. A more complex example,

```
// PI is a float defined as 3.14159
// uv is a vec2 defined as the current (x,y) pixel coordinate
vec2 rotatedPoint = rotate(bands.zw, uv, PI);
```

rotates the current pixel by PI radians from the center point of bands.zw. In this example, the rotation point for the the image updates its position based on the sound frequency intensities of the highest two bins. The results are perceived as connected, but the direct relationship is more visually ambiguous. Previous iterations of Lawson and Smith's collaboration have been primarily based on this communication model, in which incoming frequency intensities were used to create a variety of (un)clear correspondences between the audio and visual components.

2.2 AppiOSC

The AppiOSC is an Arduino based device that creates a data bridge between The Force and the modular synthesizer. On one side of the bridge, AppiOSC communicates with The Force by wireless or serial Open Sound Control (OSC) messages. On the other side of the bridge, AppiOSC communicates with the modular synthesizer using the standard 1/8 inch mono, mini-jack connection, and provides four control voltage (CV) inputs from the synthesizer and four CV outputs to the synthesizer.

Hutchins sees the modular synthesizer as performance tool that sits in the middle of a continuum between a binary of embodiment and un-embodied; where the synthesizer itself is, "...both a chainsaw and an idea." (Hutchins 2015; "Manifesto Draft" 2010) He finds commonalities between live coding and the synthesizer in algorithm complexity, graph changes, decisions, and scheduling. Furthermore, Hutchins displays a video feed of his patching as a way to show his *screen*. Finally, he makes a closing argument that, "It's no longer necessary to retreat into our imaginations to experience live coding without a computer." Which supports the idea by McLean that Hutchins quotes, "We can think of live coding in the sense of working on electricity live, re-routing the flows of control ... ."(McLean 2008) The AppiOSC supports this by being a module of the synthesizer that can add to the complexity of the patching, become part of the graph changes, be a module that makes decisions, be employed to schedule events, and is working directly with electricity by re-routing flows of control.

2.3 Visual to Audio

In order to enable a two-way connection between The Force and the modular synthesizer, Lawson and Smith collaborated with Frank Appio to develop a communication bridge in which digital information derived from The Force is converted to musically-useful control voltages (see the left column of figure 1). As the performer edits the shader, each key press triggers a 200ms-long timer. The OSC messaging is attached to this timer for the following reasons. First for convience and efficiency, this timer is also used to auto-compile the shader code that updates the graphics. Second, a continous stream of duplicate messages was deemed unneccesary if the code had not changed. Lastly, large volumes of OSC messages would overload the Arduino.

When that timer fires it executes a search function in which one or more predefined strings are identified within the code. The resulting sum of each searched symbol can then by modulated by any number of mathematical functions. For instance,

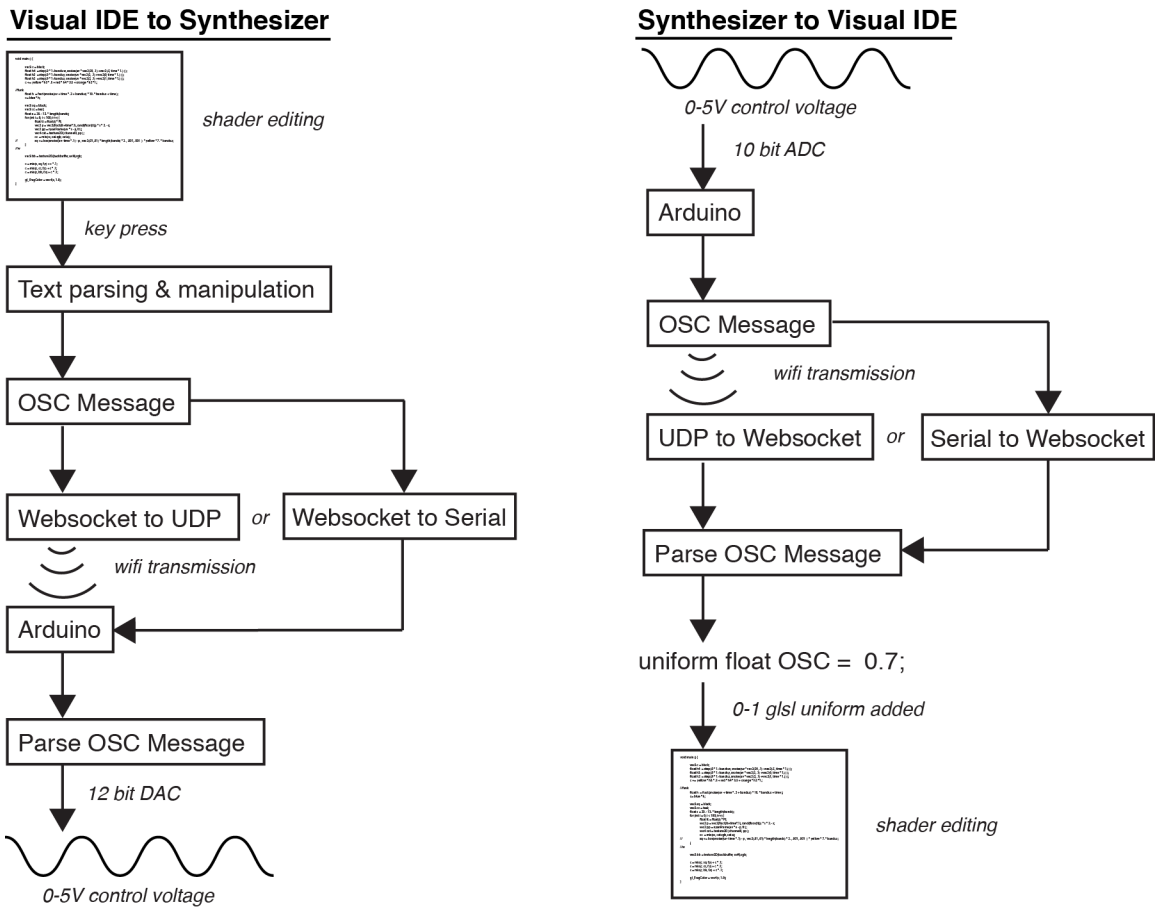

Figure 1: *Left column shows path of communication from The Force to modular synthesizer. Right column shows path of communication from modular synthesizer to The Force.*

if the search string is “Black” then a count of all instances of “Black” are found within the code and returned. A more complex case would be a search string of " " that is further modulated by *sin() * 10*. In this case, the resulting count values would oscillate between -10 and 10 as spaces are added and removed. We perceive this as a secondary stream of data that is both relevant to the primary intent of live coding algorithm creation and as an untapped source of “left-over” applicable information. At the completion of these operations, an OSC message is created using the modulated results of this search and sent out through a websocket connected to a Python bridge script.

Python retransmits the OSC message by either the User Datagram Protocol (UDP) on a local wireless network or Serial Line Interface Protocol (SLIP) by a USB cable to an Arduino.(“Arduino”) The numerical content of the message can be configured to represent a variety of waveform shapes, parameters, or frequencies. After receiving the OSC message, the Arduino produces the corresponding waveform. This information is converted into 0 to 5 volt CV by the Arduino with a 12-bit digital to analog converter (DAC), and is accessible to the modular synthesizer with a standard mono 1/8 inch connection. CV from The Force via the AppiOSC can then be used to control a variety of elements within the modular synthesizer, including but not limited to, function generation and modulation, triggering (single triggers and bursts), and frequency control over low frequency oscillators (LFO)s and voltage controlled oscillators (VCO)s.

The AppiOSC generates one of four wave shape types: sine, triangle, saw accending, and saw descending. Figure 2 shows the two core wave properties controled by the Arduino: amplitude of 0.0 volts to 5.0 volts and frequency of 0.1 Hz to 20 Hz. The voltage values are dictated by the synthsizer, which the frequency range was an observational aesthetic decision. The Amplitude property can be further specified to set a minimum, a maximum, or random. The frequency property can also be set to random. Figure 2, Example A, demonstrates a sine wave with constant frequency, constant amplitude minimum, and a AppiOSC controlled amplitude maximum. Example B in figure 2 shows a sine wave with an AppiOSC controlled frequency, AppiOSC controlled amplitude minimum, and AppiOSC controlled amplitued maximum. Example B is more indicative of the real-time stochastic processes that enable a wide variety of randomly-determined, yet constrained, wave shapes.

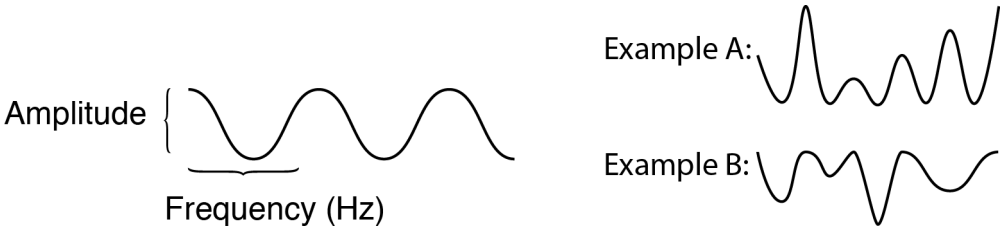

Figure 2: *Left side shows an example with the two primary parameters for function shaping. The right side shows two examples of function output by controlling parameters.*

2.4 Additional Audio to Visual

The AppiOSC device provides an additional method for the transmission of information from the audio performer to visual performer (refer to the right column of figure 1). When the audio performer connects a CV source to one of the AppiOSC inputs, that CV is transformed by a 10bit analog to digital converter (ADC), rescaled as a float value ranging from 0.0 to 1.0, and sent to either a predefined IP address as an UDP OSC message or serial address as a SLIP OSC message. In both communication scenarios, a Python script captures the message packets and passes them to a websocket connection opened within The Force. The Force then parses the OSC message, and assigns a glsl uniform vec4 value with the float data for access within the shader.

3 Aesthetics

To reiterate, the AppiOSC’s primary function is to provide bi-directional communication between The Force and a modular synthesizer, and the aesthetic potentials this communicative bridge may enable. A significant portion of audio-visual performances demonstrate tightly integrated perceptible relationships between the audio and visual.[2] With the AppiOSC, our intent is to provide a technology that enables the transmission of (un)controlled data between the performers, in order to complicate an otherwise simple, and direct aesthetic connection.

Each performer live codes or patches with a bimodal intent. Primary concern is for the immediate domain of graphics or sound, and the secondary concern is the left-overs of live coding or patching that are sent or received to the other performer. The data streams are representative of each performer’s actions and a performer can mentally flip into secondary concerns

[2]The author’s previous collaborative efforts are clearly engaged in this kind of performance.

by either specifically adding functional or nonfunctional text to modulate the synthesizer, or patching a signal that triggers a conditional sequence in the code. Although, our aesthetic choice has been to focus on the primary concerns and allow the left-overs to be a new form of communication that requires each performer to relinquish some control of their tool and use a chaotic, yet relevant stream of data in their algorithm or patch.

4 Conclusions

In its current build, the AppiOSC includes four discreet input and output CV connections between The Force and the modular synthesizer, and it may be possible to increase the number of inputs and outputs. Future builds will also improve the wave-shaping capabilities and introduce new control parameters from within The Force. The current websocket<->Python<->Arduino connection is adequate, although alternative hardware options will be investigated in order to better streamline this approach. Finally, multicasting between a collection of performers using either The Force, AppiOSC enabled synths, or other OSC enabled software/hardware solutions is an area for investigation.

Preliminary experimentation with the The Force communicating CV to the modular synthesizer via the AppiOSC has been encouraging. And while the previous iterations that utilized FFT-based analysis generated interesting results, the limitations of the largely one-way communication model had come to feel stifling. It is our hope that this two-way communication model will enable interesting aesthetic opportunities beyond what was available with the largely audio-led collaboration.

4.1 Acknowledgments

Special thanks to Signal Culture for providing Toolmaker Residency time and space to research and develop the OSC network with Arduino Wi-Fi and CV hardware and software.(D. Bergnagozzi and Bergnagozzi)

References

Alex McLean, et. al. 2016. “Tidal.” Accessed April 5. http://tidal.lurk.org.

“Arduino.” https://www.arduino.cc.

Bergnagozzi, Deborah, and Jason Bergnagozzi. “Signal Culture.” http://signalculture.org.

Casa, D. Della, and G. John. 2014. “Livecodelab 2.0 and Its Language Livecodelang.” In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling and Design*, 1–8. ACM.

D. Griffiths. Fluxus. In A. Blackwell, J. Noble, A. McLean, and editors J. Rohrhuber. 2013. “Collaboration and Learning Through Live Coding, Report from Dagstuhl Seminar 13382.”

G. Wakefield, W. Smith, and C. Roberts. 2010. “LuaAV: Extensibility and Heterogeneity for Audiovisual Computing.” In *Proceedings of Linux Audio Conference*.

Ge Wang, et al., Perry Cook. “ChucK.” http://chuck.cs.princeton.edu.

Griffiths, D. 2013. “Fluxus.” In *Collaboration and Learning Through Live Coding, Report from Dagstuhl Seminar*, 13382:149–50.

Hutchins, Charles Celeste. 2015. “Live Patch / Live Code.” In *Proceedings of the First International Conference on Live Coding*, edited by Alex McLean, Thor Magnusson, Kia Ng, Shelly Knotts, and Joanne Armitage. ICSRiM, University of Leeds.

“Manifesto Draft.” 2010. http://toplap.org/wiki/ManifestoDraft.

McKinney, Chad. 2014. “Quick Live Coding Collaboration in the Web Browser.” In *NIME’14 Proceedings*, 379–82. New Interfaces for Musical Expression.

McLean, Alex. 2008. “Live Coding for Free.” In *FLOSS + ART*, edited by Marloes De Valk Aymeric Mansoux, 224–31. Poitiers: GOTO 10 in assocation with OpenMute Publishing Ltd.

Nick Collins, Julian Rohrhuber, Ales McLean, and Adrian Ward. 2003. “Live Coding in Laptop Performance.” *Organised Sound* 8 (03): 321–30.

Quilez, Inigo. “ShaderToy.” https://www.shadertoy.com.

Roberts, Charle, and J. Kuchera-Morin. 2012. “Gibber: Live Coding Audio in the Browser.” In *ICMC Proceedings*.

Sam Aaron, et. al. “Sonic Pi.” http://sonic-pi.net.

“Shadertone.” https://github.com/overtone/shadertone.

Sorensen, A. 2005. “Impromptu: An Interactive Programming Environment for Composition and Performance.” In *In Proceedings of the Australasian Computer Music Conference*.

Sorenson, Andrew. 2014. “Live Coding for Real-Time Systems.” goto; conference. Kenote Presentation: https://www.youtube.com/watch?v=Sg2BjFQnr9s.

“SuperCollider.” http://supercollider.github.io.