Using the Electric Guitar as a Live Coding Interface

Martin Dupras
The Open University (UK)
martin.dupras@open.ac.uk

Simon Holland
The Open University (UK)
simon.holland@open.ac.uk

Paul Mulholland
The Open University (UK)
paul.mulholland@open.ac.uk

ABSTRACT

The aim of this paper is to explore how electric guitarists can engage in live coding without removing their hands from the guitar to enable them to implement and apply real time signal processing code to their sound. We propose a paradigm where a guitarist plays a guitar capable of outputting a separate audio signal from each string to support the expression of both sound and data, and uses this in combination with a MIDI foot controller to alternate switching between playing and programming modes. Evolving a solution involves addressing two key design challenges: (1) the identification of suitable means to translate playing gestures into recognisable abstract data, and (2) the development of a flexible mapping strategy to allow players of different styles or practices (e.g. metal vs. jazz) to extend the system to suit their needs. The resulting design, which enables guitarists to both play and program, is currently undergoing an iterative process of refinement and evaluation.

1 Introduction

Live coding is generally performed using the computer's human interface devices (HIDs) which typically consist of an alphanumeric keyboard and a pointing device. For many purposes, this hardware interface is adequate since it is simple, ubiquitous, and efficient for textual and numeric data entry. The live coding virtual interface itself usually takes the form of a code editor, interpreter, IDE or specialised GUI, whose design is typically at least partly tied to the physical features of the interface, such as modifier keys (e.g. "ctrl") or a pointing device. Thus, the design of the editor is well suited for HID input.

This paper explores how guitar playing can be used to effect both musical performance and live coding without removing one's hands from the guitar and without disrupting the performance.

While at first sight the guitar may seem an awkward interface for live coding, we will argue that given the skillset of a practiced player of the conventional guitar, some live coding operations might be more easily executed via this interface. Such an interface should present the following:

1. the ability to write code by playing learned musical phrases;
2. the input of musical data (such as sets of notes, scales, intervals, durations, rhythms) without having to translate them to abstract representations;
3. the ability to quickly switch between playing the instrument conventionally and using it as a programming interface.

2 Related work

Piano keyboard interfaces and other interfaces isomorphic to an array of buttons have been used as live coding interfaces, but not, to our knowledge, stringed instruments.

The CodeKlavier (Veinberg and Noriega (2022)) uses a MIDI piano as a live coding interface. This approach exploits the symbolic (MIDI) representation of the playing to send commands during performance. The sound of the piano is always audible, and therefore the musical playing that creates the code is intrinsic to the musical performance. The piano is heard all the time and both as an audible musical performance and possible act of programming. Because the programming actions are audible, they need to also be musically appropriate.

The solution we propose would offer the choice to the player to make the programming audible or silent appropriately to musical context.

The Stenophone (Armitage and McPherson (2017)) is an instrument designed to be "a live coding keyboard which is also a digital instrument." This project seeks to extend the symbolic interaction paradigm "with gestures more readily associated with an embodied instrument" (Armitage and McPherson (2017)). The implementation converts chord key presses into "words" (symbolic to symbolic representation) and uses continuous gestural data mapped to synthesis parameters. The Stenophone does not make use of learned instrumental playing skills, since it is based on the stenotype, a device used by stenographers.

3 Instrumental input interfaces

Modern electric guitar performance often takes advantage of signal processing (typically in the form of a chain of fixed "guitar pedal" effects processors, or "pedalboard".) However, the pedalboard paradigm is limited in what can be achieved with it because the topology of the signal graph must essentially be fixed in advance of any performance by the hardware configuration of the pedal chain(s). During performance, control of the processing is limited to the activation or bypassing of nodes ("pedals") and to the change of parameters using expression pedals.

If the guitarist could undertake live coding without taking their hands off the guitar, this could transform the expressive possibilities of electric guitar performance practice. Adding the creative possibilities offered by live coding in this context has the potential to enrich electric guitar performance practice in at least the following two ways. Firstly, by allowing the improvisation of unpremeditated signal processing graphs to suit particular performances and particular improvised musical ideas. Secondly, using facilities described in the next paragraph, by allowing the application of improvised signal processing graphs differently to different guitar strings.

To live code with a guitar, we need to interface it with the computer in a way that allows gestures performed on the guitar by an expert guitar player to be understood by the computer without restricting the normal playing of the instrument. The most obvious starting point might be to analyse the standard monophonic output of the guitar to look for pitches, amplitudes and onsets. However, expert guitarists are typically performing expressive musical gestures not just with the guitar as a whole, but independently on each string. Merging this signal, as a standard electric guitar pickup does, loses a great deal of meaningful gestural information. Separating the mixed signal into discrete signals for each string "is a major issue (…) in the context of audio processing" (Pichevar and Rouat (2007)). For this reason, we need to use a separate pickup for each string, each with its individual audio output. Hexaphonic systems ("six channels for six strings") are typically associated with guitar-to-MIDI systems.

However, our system makes no use of MIDI for two key reasons: high latency and inaccuracy (Puckette (2007)). Translation to MIDI events is not an accurate representation of guitar sound or how notes behave; for instance, if a string is left to decay naturally, a midi note-off event would be generated before the note has stopped. Furthermore, latency problems (due to the need to determine a pitch with great accuracy before committing it to a MIDI event) lose large parts of the real time expressiveness of a guitar. Thus our system rests firmly on extensive use of the separate audio channels associated with each string.

Our approach is to perform real-time analysis of the individual string signals in software. Parallel algorithms can extract data for each string, such as estimated frequency, quantised pitch-class, RMS amplitude and onsets, which can then be used to represent musical gestures. This information can be processed, analysed, represented and used in broadly two different ways in real time as follows. High level representations of performance data such as notes, duration and grid placement are useful because they allow abstract analysis of the playing (e.g. the playing is in the key of D major) and transformations of the data such as musical transposition and rhythmic displacement. At the same time, the raw signal received from each string offers more depth of information, which allows for analysis of features such as direction, rate of change, projected destination, etc. The analysis of both kinds of data offers a rich potential for guitar-driven live coding. For example, a played rhythm can be represented as events on a grid, comprising thin data that can be easily manipulated; and the same played rhythm can also be recorded with time accuracy, which can allow analysis to reveal underlying features such as subtle time variations (a.k.a. "groove").

4 Aim

Given the above issues, the primary aim of this research is to develop a low-latency framework that enables guitar players to create and manipulate guitar-processing programs while playing. The design constraints are that it should:

- allow **real-time** operation during instrumental playing
- foster **musically** meaningful output
- allow the player to use the full range of their playing techniques.

A further aim is to identify how the musical expression and improvisational possibilities of performers change when using such interfaces.

For reasons outlined earlier, our development of the system focuses deliberately on aspects of live coding that fit well with the existing practice of expert guitarists, and in practice we will focus principally on editing the signal graph applied to the guitar sound. However, we will argue that the approach taken is equally applicable to live coding with the guitar more generally.

More specifically we will explore methods that enable the guitarist to spontaneously conceive and implement signal processing structures during performance, quickly and iteratively, without disrupting their connection with the audience.

Naturally, there will be limits to the complexity of graphs that can be assembled or edited during short performances while maintaining high audience engagement at every stage. However, the graph editing techniques that we have already prototyped are sufficiently general to be applied to live coding for the guitar more generally without requiring any additional conceptual steps.

5 Approach

For the system to be usable alongside conventional guitar playing, it needs to offer the player a conventional playing mode, where all the playing features of the guitar are as normal, and the sound of the guitar is audible, as well as, by contrast, marshalling modes which are activated with the foot controller. Each marshalling mode analyses the hexaphonic guitar signal to convert played patterns into relevant data or programming constructs.

For this research we are using an electric guitar (Figure 1) equipped with a Cycfi Research Nexus hexaphonic system, which outputs each string individually to an RME Fireface UCX audio interface, connected to a Macbook Pro; and a Keith McMillen Instruments SoftStep 2 MIDI foot controller (Figure 2), which is used to access the marshalling modes. The prototype software[1] is written in PureData (Puckette et al. (1996)).

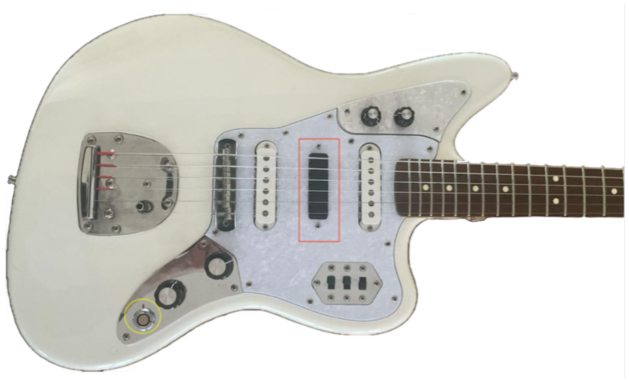


Figure 1: Squier jaguar with Cycfi hexaphonic pickup (red) and multi-pin connector socket (yellow)

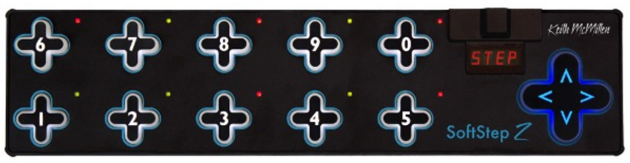


Figure 2: KMI Softstep2 MIDI foot controller.

The hexaphonic guitar input is fed into parallel signal analysis algorithms that output data for each string, such as continuous pitch and RMS amplitude estimation, onset detection and threshold detection with hysteresis. These data streams are further turned into processed data streams adapted to the needs of each of the marshalling modes.

Recall that a key aim of the system is to allow guitarists to live code without taking their hands off the guitar. Many guitarists are used to operating foot-controlled effects. Consequently, in our system live coding is carried out by operating foot controller switches to select marshalling modes (described below) and playing short, typically silent note sequences

[1]Code available by email request to the first author.

which are converted in system navigation actions that navigate an n-tree of programming nouns and verbs that can be turned into PureData objects insertable into the audio processing graph.

The foot controller serves as the marshalling controller, managing **marshalling modes** (Figure 3) that guide listening agents to navigate, assemble and dispatch command queues, and to collect playing data such as pitch sequences, pitch sets, chords, rhythms and patterns. Programming nouns and verbs, abstract data and musical data are created, entered, and manipulated through normal guitar playing actions, marshalled by the foot controller.

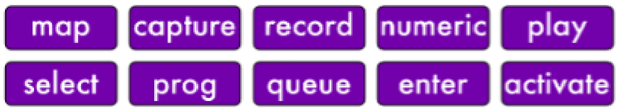


Figure 3: Marshalling commands mapped to the Softstep 2 foot controller.

Crucially, the design of the system allows for new marshalling modes and computational listeners to be added incrementally to the system by individual users and communities.

The following section presents a summary of key design features and two thumbnail scenarios, all based on the current implementation. The design is rapidly evolving as experience in its use grows. (Figure 4) shows how live coding interactions in the current implementation are organised.

The **Play** mode allows the signal from the guitar to be heard by the audience, and for it to be processed by any active processing graph. Signal processing **graphs** are created in **Codeboards**, which are empty containers capable of holding graphs. In the current implementation, the containers are PureData *subpatches* that are visible on the live coding screen.

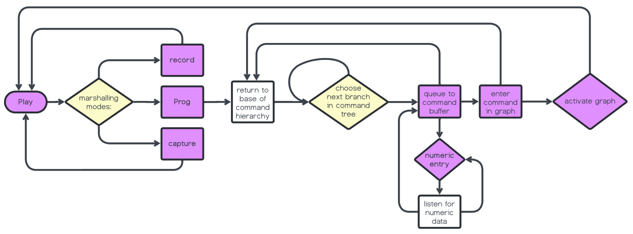


Figure 4: Diagram showing current implementation.

The **Prog** (“programming”) marshalling mode allows the creation of processing graphs. Entering **Prog** mode initiates a command selection mechanism which navigates a hierarchical tree of programming command elements (Figure 5) which we shall sometimes refer to for purposes of generality as ‘*nouns and verbs*’ — allowing independence from the underlying implementation. In this mode, fret n of the lowest guitar string selects the nth branch of the lowest level of the tree; fret n of the next string selects the nth subbranch, until a leaf is reached. This allows quick “played” navigation of a tree containing potentially hundreds of command elements. Once the desired command element (“leaf”) is found, it can be **queued** using the **Queue** switch on the foot controller. The **Enter** switch then executes the assembled command, typically adding or modifying a node in some processing graph.

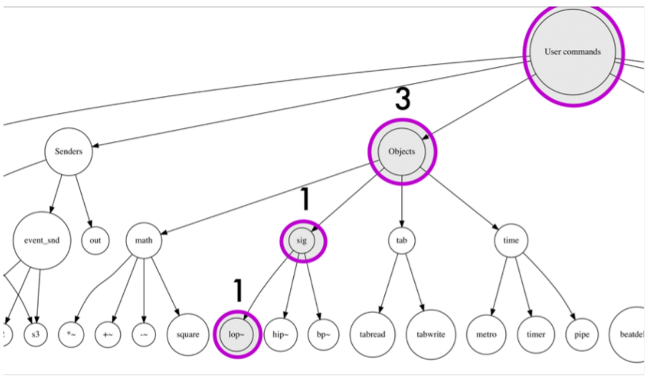
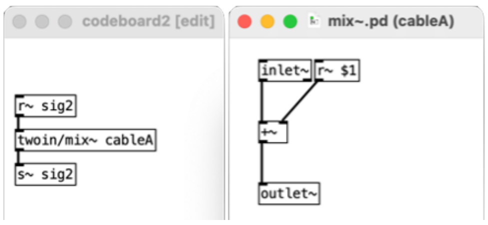


Figure 5: Diagram showing the command tree; highlighted: navigation to branch 3, sub-branch 1, sub-sub-branch 1.

Extensive edits to a graph can be effected by successive sequences of **Prog**, **Queue** and **Enter** presses. Once a graph is complete it can be activated by a press of the **Activate** footswitch, which makes the signal graph active and makes the next Codeboard (Figure 6) available for the creation of another **graph**.



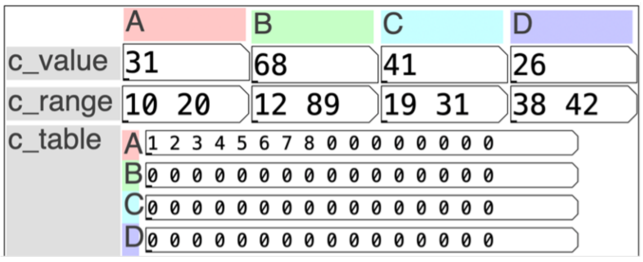
```
r~ sig2
twoin/mix~ cableA
s~ sig2
```

```
inlet~  r~ $1
+~
outlet~
```

Figure 6: Example (left) showing a played-programmed graph mixing two sources. The first source is the argument of the first object and the second source the argument of the second. (Right) The contents of the mix pre-built block.

The **Numeric** marshalling mode enables entry of numeric data. Pressing the **Numeric** switch enters this mode if coming from another mode, or cycles through three numeric entry methods. The **Numeric-repetition** mode counts the number of notes (allowing the quick entry of small numbers.) The **Numeric-interval** mode allows entry of the number of semitones between two notes, which also allows the entry of negative numbers. **Numeric-fixed** maps pitch classes (C, C#, D, etc.) to digits, which allows the entry or arbitrarily large numbers (e.g. 6234).

The **Capture** marshalling mode allows the capture of musical "symbolic" structures such as pitch sets or grid rhythms to be captured and stored to the last selected **capture container**. Different types of containers (e.g. value, value range, pitch set, pitch sequence, chord, quantized rhythm) are available (Figure 7), visible to the player and audience, and can be chosen by pressing the **Select** foot switch and navigating the selection tree using the guitar playing in the same way as **Prog** mode. The selected container listens to the mode appropriate for its data type (e.g. a pitch-set container would listen to a **pitch listener**.)



	A	B	C	D
c_value	31	68	41	26
c_range	10 20	12 89	19 31	38 42
c_table	A 1 2 3 4 5 6 7 8 0 0 0 0 0 0 0 0			
	B 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			
	C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			
	D 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			

Figure 7: Examples of containers which can be accessed by played-programming structures

The Record mode allows the recording of audio data into arrays, which can be chosen by using the Select switch and navigating a tree pointing to the available arrays using guitar playing.

Three brief very simple worked scenarios created using the guitar interface in the current prototype system are outlined below.

5.1 Scenario 1: inserting a tremolo with a rate of 3Hz:

The player does the following (Figure 8) :

1. press PROG on the foot controller, which brings up the verbs and noun tree navigation.
2. play fret n on string 6 to select branch ("modulation")
3. play fret p on string 5 to select branch ("tremolo")
4. press QUEUE on the foot controller to add "tremolo" to the queue
5. press NUMERIC on the foot controller, which brings up numeric mode (default: numeric-repetition)
6. play any three notes on the guitar (which creates the argument "3")
7. press ENTER on the foot controller to create a "tremolo 3" object in the Codeboard
8. press ACTIVATE to make the Codeboard active

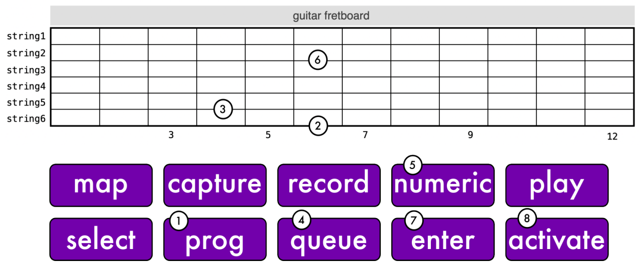


Figure 8: Scenario 1

5.2 Scenario 2: Creating a note counter (count stored in container A):

The player does the following (Figure 9):

1. press PROG on the foot controller, which brings up the verbs and noun tree navigation.
2. play fret n on string 6 to select branch (“receiver”)
3. play fret p on string 5 to select branch (“onset”)
4. play fret q on string 4 to select branch (“midipitch”)
5. (optional) press QUEUE on the foot controller
6. press ENTER to create “midipitch” object in the Codeboard
7. press PROG on the foot controller, which brings up the verbs and noun tree navigation.
8. play fret n on string 6 to select branch (“counters”)
9. play fret p on string 5 to select branch (“count A”)
10. (optional) press QUEUE to add “count A” to the queue
11. press ENTER on the foot controller to create a “count A” object in the Codeboard
12. press ACTIVATE to make the Codeboard active

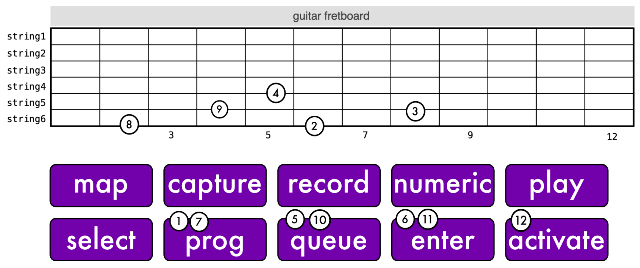


Figure 9: Scenario 2

5.3 Scenario 3: Inserting a chorus and a delay of five beats:

The player does the following (Figure 10):

1. press PROG on the foot controller, which brings up the verbs and noun tree navigation.
2. play fret n on string 6 to select branch (“modulation”)
3. play fret p on string 5 to select branch (“chorus”)
4. (optional) press QUEUE on the foot controller
5. press ENTER to create “chorus” object in the Codeboard
6. press PROG on the foot controller, which brings up the verbs and noun tree navigation.
7. play fret n on string 6 to select branch (“delays”)
8. play fret p on string 5 to select branch (“beatdelay”)
9. press QUEUE on the foot controller to add “beatdelay” to the queue
10. press NUMERIC on the foot controller, which brings up numeric mode (default: numeric-repetition)
11. play any five notes on the guitar (which creates the argument “5”)
12. press ENTER on the foot controller to create a “beatdelay 5” object in the Codeboard
13. press ACTIVATE to make the Codeboard active

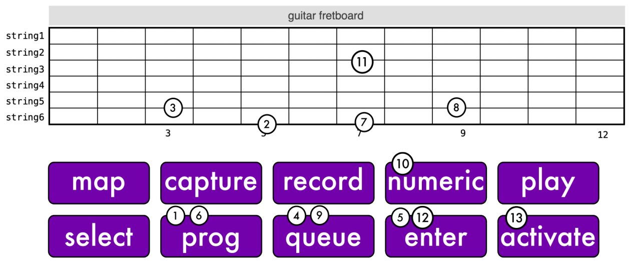


Figure 10: Scenario 3

6 Conclusion and future work

Real-time programming during instrumental performance with a guitar raises several questions: (1) what factors will govern how well coding and conventional playing can be interleaved artistically and effectively; (2) what methods are fast enough ("low-latency") to allow programming constructs to be entered by instrumental playing; and (3) how flexible must the system be to accommodate a variety of playing techniques.

Development of the system needs to maintain a balance between programming flexibility and expressive guitar playing. Therefore, solutions will be evaluated on how seamlessly they integrate programming actions with guitar performance.

Future work will consist of: (1) contextual design user study using technology probes (Hutchinson et al. (2003)) to map the conceptual programming space that exists for performers; (2) iterative design of a live programming instrument-driven user interface for live performance; (3) a small-scale user study of skilled performers learning to use the system in their practice.

6.1 Acknowledgements

This work has been carried out with financial support from EPSRC Training Grant DTP 2022-2024 Open University.

References

Armitage, Jack, and Andrew McPherson. 2017. "The Stenophone: Live Coding on a Chorded Keyboard with Continuous Control."

Hutchinson, Hilary, Wendy Mackay, Bo Westerlund, Benjamin B Bederson, Allison Druin, Catherine Plaisant, Michel Beaudouin-Lafon, et al. 2003. "Technology Probes: Inspiring Design for and with Families." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 17–24.

Pichevar, Ramin, and Jean Rouat. 2007. "Monophonic Sound Source Separation with an Unsupervised Network of Spiking Neurones." *Neurocomputing* 71 (1-3): 109–20. https://doi.org/10.1016/j.neucom.2007.08.001.

Puckette, Miller et al. 1996. "Pure Data: Another Integrated Computer Music Environment." *Proceedings of the Second Intercollege Computer Music Concerts*, 37–41.

Puckette, Miller. 2007. "Patch for Guitar." In *Proc. PureData Convention*, 7:21–26. Citeseer.

Veinberg, Anne, and Felipe Ignacio Noriega. 2022. "The CodeKlavier: Appropriating the Piano As a Live Coding Instrument." *Rethinking the Musical Instrument*, 316.