A WEB-BASED MIDI CONTROLLER FOR MUSIC LIVE CODING – SUPPLEMENTAL MATERIAL –

Frank Heyen **Dilara Aygün** **Michael Sedlmair**
VISUS, University of Stuttgart, Germany
frank.heyen@visus.uni-stuttgart.de, michael.sedlmair@visus.uni-stuttgart.de

1. OVERVIEW

This document contains supplemental descriptions and figures, see our main paper for the most important information. Our implementation is publicly available both as open source code and web app at github.com/visvar/sonic-pi-controller.

2. DESIGN

We aspire *Sonic Pi Controller* (SPC) to be a simple, efficient, and beginner-friendly way to enhance the musical and creative experience of live coding. Therefore, we kept the design simple and used familiar user interfaces that resemble instruments or hardware MIDI controllers. In this section, we first introduce common tasks we identified for music live coding and then explain how our design addresses these tasks.

2.1 Concept

Sonic Pi's design is based on three core principles [1] :

- "Simple enough for the 10 year old within you
- Joyful enough for you to lose yourself through play
- Powerful enough for your own expressions"

We designed our controller to complement Sonic Pi according to these core principles by tackling some shortcomings in regard to usability and expressiveness. While Sonic Pi is powerful for diverse musical expression, it comes with a trade-off regarding simplicity. Live-coding allows fine adjustments of different musical effects, for instance altering volume, reverb, or overdrive. If the musician wants to perform a dynamic adjustment, they have to either retype a value several times and re-run, or write a loop which changes the value for them. The former is hard to achieve quickly enough and the latter requires planning ahead and prevents spontaneous changes, limiting musical expressiveness. Our *Sonic Pi Controller* provides an easier solution for certain tasks, with interactive inputs that the user can customize for different use cases.

[1] github.com/sonic-pi-net/sonic-pi

2.2 Tasks

Common user tasks we want to address with our design include:

Task 1: Trigger actions ***(trigger)****.* Users may want to spontaneously play samples or synthesized melodies while their code performs a backing track. For example, they could improvise a short synth solo over a repeating drum loop that is already written. Even this simple action allows for expressiveness, as users might trigger arbitrary complex functions in the code.

Task 2: Toggle state ***(toggle)****.* We want to allow users to toggle certain components of the music. This action should not be limited to playing or stopping a sample, but rather allow switching between different sounds and toggling loops. Examples include toggling a bass loop or turning a low pass filter on or off.

Task 3: Change parameter values ***(change)****.* There are several audio effects commonly used in music creation that allow fine-grained control trough parameters. Changing those parameters quickly and precisely during a live coding performance is hard to do with only code, since changes might happen faster than the user can type. For instance, changing the gain of a track or the threshold of a filter requires updating parameters continuously to achieve a smooth transition.

In this work, we focus on the above common tasks as a baseline. As music is a creative endeavor, artists might come up with further tasks that could be supported in the future by either custom Sonic Pi code or extension to our design.

2.3 Sonic Pi Controller

We implemented SPC's controller frontend as a React app that runs next to Sonic Pi. The communication between frontend and Sonic Pi uses MIDI messages. SPC therefore requires a browser that implements the Web MIDI API.

2.3.1 Interactive Controller Inputs

Instruments. We implemented two examples for instruments, a piano keyboard and a drum kit (Figures 1 and 2). By clicking on the keys or the drums, the user triggers sending a MIDI signal to Sonic Pi, which then plays the corresponding note or sample (**trigger**). Users can also play the drum pad without a mouse, over the computer keyboard.

Late-Breaking / Demo Session Extended Abstract, ISMIR 2022 Conference

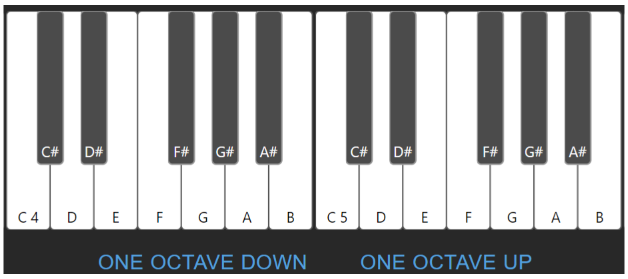


Figure 1. Users play SPC's piano keyboard by clicking or tapping on keys.

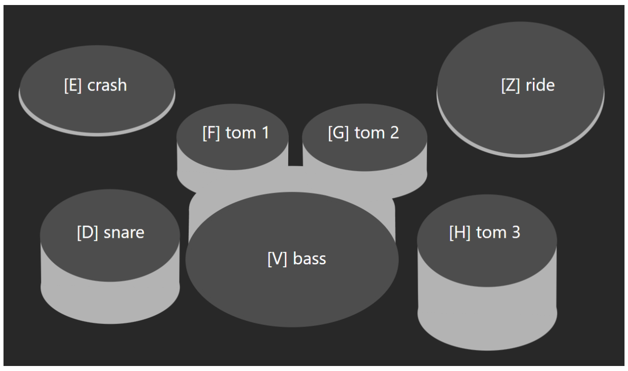


Figure 2. Users play our drum kit either played through mouse clicks, touch, or pressing the corespondent keys, such as *D* for the snare.

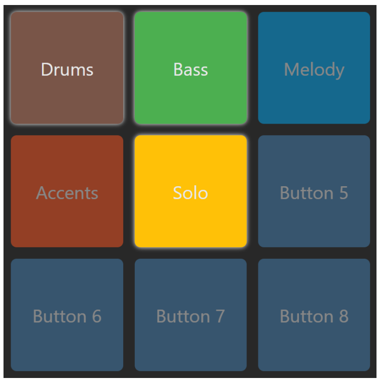


Figure 3. Each button of our DJ Pad corresponds to a Boolean variable in Sonic Pi that can be used to toggle a sample loop or function that plays a melody. The buttons' labels and colors can be customized to make their purpose easier to identify.

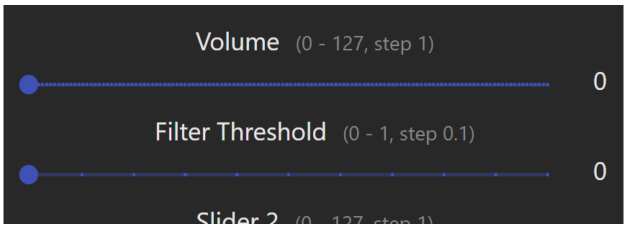

Volume (0 - 127, step 1)
0
Filter Threshold (0 - 1, step 0.1)
0
Slider 2 (0 - 127, step 1)

Figure 4. Sliders for changing numerical parameters allow to quickly and spontaneously adjust effects.

Button matrix. We kept our design close to familiar button grid interfaces (Figure 3) and simulate the illumination, which signals whether a certain state if active, by changing the opacity of the button's color. With Sonic Pi code, the users can implement functionality based on a Boolean variable for each button. They then toggle these variables between true and false by clicking the buttons, which also makes them visually update their state. Buttons can be customized by changing their color and label, to make them easier to relate to their functionality.

Sliders. Our sliders allow users to change any discrete or continuous numerical values, such as volume or effect parameters (Figure 4). With code, users can define the features to be altered by the sliders. Interacting with slides changes integer or float variables and these can be used in any part of the code. For example, a slider could be used to interactively transpose a melody loop, providing a simple way to make a creation sound more interesting, or to test different values before choosing one. Similar to our buttons, sliders can be labelled. The user can further adjust their range, by choosing minimum, maximum, and step values. These values have to be set inside the corresponding code, so values can be mapped to MIDI ranges and back.

2.3.2 *MIDI Communication*

There are projects for CLI-based control[2]. We chose MIDI for communication nevertheless, as it is a commonly used, open, and flexible standard. For example, MIDI allows to also control other applications such as stage lighting. Our users could also replace or complement our approach by other MIDI controllers at any time, avoiding a technological lock-in. Moreover, MIDI allows to implement our design as a web app that users can access without any setup or technical experience, simply by following a https://visvar.github.io/sonic-pi-controller.

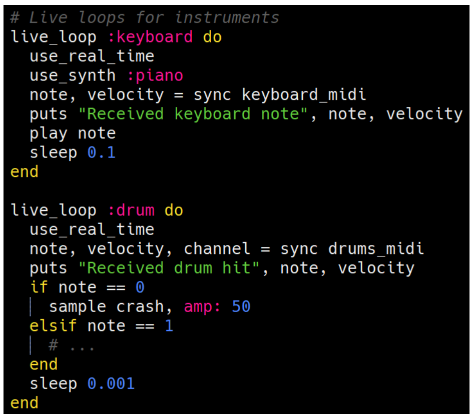
We encode user inputs as follows: When interacted with, our instruments, pad buttons, and sliders send MIDI *note_on* messages consisting of two MIDI values: note and channel. Each type of input communicates over its own channel to differentiate where a message comes from, whereas the note determines the instrument's note or sample, the button, or the slider. For sliders, the MIDI velocity encodes the current value, requiring to map values to a range of 0 to 127 before transmission and then back inside the code. We provide boilerplate code for this mapping.

2.3.3 *Sonic Pi Implementation*

In the following, we provide examples for how to use Sonic Pi with SPC. For the complete implementation, see our GitHub repository [link will appear here after acceptance]. Sonic Pi listens to incoming MIDI ports and decodes messages to note, velocity, and channel. Our boilerplate code reacts to those messages by directly playing notes or samples for keyboard and drums, or updating Boolean and floating point variables for buttons and sliders (Figures 5 and 6). Live coders can access these variables inside con-

[2] For example sonic-pi-tool (github.com/lpil/sonic-pi-tool) and sonic-pi-cli (github.com/Widdershin/sonic-pi-cli)

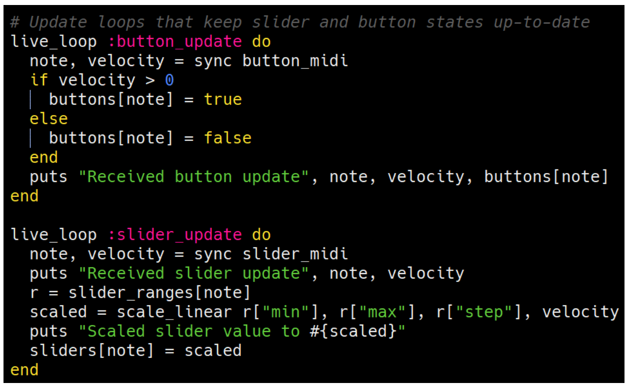
ditionals or effect parameters – or in any other way they like, for example to control their custom functions.



```
# Live loops for instruments
live_loop :keyboard do
  use_real_time
  use_synth :piano
  note, velocity = sync keyboard_midi
  puts "Received keyboard note", note, velocity
  play note
  sleep 0.1
end

live_loop :drum do
  use_real_time
  note, velocity, channel = sync drums_midi
  puts "Received drum hit", note, velocity
  if note == 0
    sample crash, amp: 50
  elsif note == 1
    # ...
  end
  sleep 0.001
end
```

Figure 5. Boilerplate for triggering instruments.



```
# Update loops that keep slider and button states up-to-date
live_loop :button_update do
  note, velocity = sync button_midi
  if velocity > 0
    buttons[note] = true
  else
    buttons[note] = false
  end
  puts "Received button update", note, velocity, buttons[note]
end

live_loop :slider_update do
  note, velocity = sync slider_midi
  puts "Received slider update", note, velocity
  r = slider_ranges[note]
  scaled = scale_linear r["min"], r["max"], r["step"], velocity
  puts "Scaled slider value to #{scaled}"
  sliders[note] = scaled
end
```

Figure 6. Boilerplate for updating variables.

3. CASE STUDIES

We evaluate our idea and implementation through case studies that compare code-only implementations to our combined approach. The usage scenarios reflect the tasks we described in subsection 2.2.

3.1 (De-)Activating a Track

The live coder might want to stop a drum loop for a while and activate it again later in the performance. Without SPC, this would be realized by searching for and then commenting out the code responsible for the loop, followed by a re-run. With SPC, the user only needs a few lines of boilerplate code (Figure 6) and is now able to start or stop loops by simply pressing a button on the graphical interface.

3.2 Fading

While live-coding, it is common practice to fade different parts of music in or out. With live coding alone, the performer can achieve this effect by constantly changing the volume over time, requiring to quickly type the new value and re-running the code repeatedly. While fading could be implemented in code, there might be no easy or elegant solution [3]. Implementing such functionality on the fly distracts the live coder from actual music creation. Instead, our sliders can be used to easily realize fading effects without loosing touch with creative thinking.

3.3 Improvisation

Suppose that the user wants to improvise a song with a melody. With Sonic Pi alone, the user has to write down actual notes and then press play. The temporal disconnect between writing down the melody and hearing it gets in the way of improvising, since this resembles more composing than actually playing the music. Through the SPC interface, users can quickly play new melodies, before writing them down as code. This method is not only faster, but allows for more realistic improvisation, closer to playing an actual instrument.

4. ACKNOWLEDGEMENTS

This work was funded by the Cyber Valley Research Fund.

[3] https://in-thread.sonic-pi.net/t/fading-live-loop-in-out/2464