A Block Based Music Live Coding Environment for Kids

Scott Fradkin
Flexion, Inc
scott@fradkin.com

ABSTRACT
There are a number of great live coding languages available for creating music, many of which are inaccessible to young children for a variety of reasons. Sonic Pi is perhaps the most kid friendly live coding environment for music currently available, but requires the user to be able to read and write. I propose an environment, Snap Music, that is accessible for young children, the same audience that uses applications such as Scratch or ScratchJr to learn how to code. This would be a similar block based environment that can be used for live coding music and learning how to code at the same time. To this end, I have begun to create this environment as a browser based application that uses the Snap/ environment and the Tone.js library.

1. INSPIRATION
The initial inspiration for this environment came from Sonic Pi. Sonic Pi is a wonderful environment that merges together learning how to code with creating your own music with near immediate feedback. A stated goal for Sonic Pi is that it needs to be accessible to a 10 year old (Blackwell & Aaron 2015), however in workshops I have taught Sonic Pi to children as young as 8 or 9 years old. Sonic Pi does require the user to be able to read and write, but has a nice API that is simple to use and can scale more complex if the user wishes. Sonic Pi uses many blocks to define various functionality and transforms (figure 1). This translates well to a block-based language.

```plaintext
live_loop :bassline do
    with_fx :slicer, phase: 0.25, pulse_width: 0.25 do
        with synth :prophet do
            play :c3, release: 4
            sleep 4
        end
    end
end
```

Figure 1

Scratch is perhaps the most well known block-based language/environment and is used worldwide to teach computer science. It is geared towards an age range of 8 - 16 (Resnick, et al 2009). Its younger sibling ScratchJr is made for ages 5 - 7 (Flannery, et al 2013).

Scratch has a few music blocks that allow a user to play single notes or samples (figure 2). The interfaces for these blocks require a user to either type numbers or select values from a drop-down menu. One exception to this is the play note block which displays a small keyboard when the user selects the arrow next to the note number (figure 3).
As a comparison, ScratchJr is extremely limited as far as sound is concerned (figure 4). ScratchJr blocks use symbols to denote the block and limits text entry for the user which simplifies the interface for younger kids.

Snap Music is intended to be an amalgamation of a music API similar to Sonic Pi combined with the block-based structure of Scratch with the symbolic interface simplification that ScratchJr gives the user. Its aim is to provide a robust live coding interface for creating music within the familiar environment that many kids are already used to using. It also should interoperate with the existing Snap! blocks.

Snap Music is an application that is a custom clone of Snap!. It is written in JavaScript and HTML5. The audio is implemented using the Tone.js library which uses the Web Audio API.

2. PROGRESSION OF API DESIGN

2.1 Introduction

Snap Music is in a state of constant development. This paper describes the progression of API design up until the current version under development. There is much functionality that will be discussed later in the document that is not currently implemented but will be in the future.

2.2 Version 0.1

From the beginning, Snap Music was pictured as a sort of block-based native clone of the Sonic Pi API. Sonic Pi uses a Ruby DSL to create an easy to use API that is intuitive to use and has its roots in using Ruby blocks to denote scope for various things.

The initial release of Snap Music did not have very many blocks (figure 5).
To loop music the user placed oscillate note and silence blocks within the play notes in simple synth c-block. Additionally, the add fx c-block could surround note blocks to apply one of the Tone.js pre-built effects.

One of the major drawbacks to this release other than the limited number of blocks available is that this is too close to just using Sonic Pi. The blocks are wordy and hard to use. These blocks don’t quite meet a goal of being easier to use than the original Sonic Pi API.

This first release was all about trying to prove out the architecture and make something work. This release included a custom Snap! block and much custom code within Snap!. Having a custom block isn’t a bad thing, but the way the Snap! code was customized made this release nearly unmaintainable for the future.

One thing that a programmer needs to know about using Sonic Pi is that he or she must work with the concept of time. In Sonic Pi the user must use the sleep function in order to provide silence or space between playing notes. Each live loop in Sonic Pi keeps track of it’s own internal virtual time. In this way, synchronicity between loops can be handled and all audio events scheduled to SuperCollider are timed correctly by mapping the virtual loop time to the real time (Aaron, Orchard, Blackwell 2014).

The first attempt at scheduling in Snap Music using the Tone.js scheduling system emulates Sonic Pi’s system to a degree. However, this fell a bit flat within the JavaScript and Tone.js frameworks. This version of Snap Music used release 5 of Tone.js and scheduled all audio events to the global timeline. This was cumbersome, required a lot of tracking of details, and calculations that really shouldn’t have been necessary in order to loop block stacks.

2.3 Version 0.2

Version 0.2 of Snap Music aimed to fix some of the large issues as previously noted. The first major change was to separate the custom modifications to the core Snap! code into their own files that define functions that override the original Snap! functions. This allows the Snap Music project to remain up to date with the most recent upstream Snap! code without causing any other issues.

The second major change was to upgrade to release 6 of Tone.js. The newer version of Tone.js includes some abstractions for multiple timelines. The implementation was changed so that each block stack is implemented as a Tone.Part. These Parts can be individually scheduled and looped which streamlines a lot of the original code.

Along with those changes, the blocks were modified to make things clearer (figure 6).
Notice that some blocks have disappeared, simplifying the API temporarily. The ultimate goal is to add those blocks back, but for the time being, they have been removed. The language on the blocks has been simplified, but still retains the Sonic Pi semantics of adding notes and samples and silence inside a live loop.

This gets closer to a better starting point for this environment, but isn’t quite what the goal is.

2.4 Version 0.3

In thinking about how the programming interface can be simplified even more, TidalCycles became the inspiration. TidalCycles is a Haskell based DSL for live coding music. Where it differs from something like Sonic Pi is that TidalCycles is concerned about pattern. Because it’s all about pattern, it takes the concept of time out of the equation for the most part. Users create patterns of sample and synth invocations and the underlying TidalCycles engine calculates when each sample or synth should be played (McLean 2014). This can be a freeing thing for a live coder.

The thought is that taking timing out of the equation for a child can be an equally freeing experience, allowing the child to concentrate on creating code patterns that make music rather than worrying about defining space between sound.

To that end, version 0.3 of Snap Music by default will auto-calculate time offsets for notes using the current tempo. Though the blocks themselves haven’t changed at this point (figure 7), Snap Music will ignore the times specified by the blocks. Also added in version 0.3 is the ability to modify the tempo.

3. LIVENESS

The liveness of the UI is realized by the user creating one or more block stacks with a live loop block and adding some combination of play note, silence, or sample blocks inside them. The user must also put the green flag block (the green flag block is a built in Snap! block that captures the event of someone pressing the green flag control as the usual mechanism for starting a Snap! program) at the top of each block stack. Pressing the green flag button will start the live loops.

Now the user can modify the blocks inside one or more of the live loops. When the user presses the green flag button again, the block stacks will be re-evaluated, and re-started after the current loop is completed.

In this way, the user can create a changing palette of audio.

4. FUTURE STATE

This is a good start, and at the time this paper is written, this is all that is implemented. However, the blocks and interfaces defined do not go far enough towards achieving the goal of having a more visual experience for creating music.

Keeping in mind that as previously stated, the age range that Scratch is trying to reach is 8 - 16 years old and ScratchJr is for ages 5 - 7, this interface aims for the lower age range similar to ScratchJr. ScratchJr uses more symbols on the blocks so that children just starting out how to read have an easier time navigating the interface. The question is how to create a more visual interface.
for creating music within the Snap! environment to minimize the complexity. One of the ultimate goals of Snap Music is to have an interface that doesn’t require or requires very little typing. These visual interfaces will be the next pieces of functionality to be created in Snap Music.

For choosing a note to play, rather than using a traditional keyboard, a “keyboard” with 12 different colored keys will be displayed. Each note will have a unique color and will represent a full octave of notes. Up and down arrows can be clicked on in order to modify the octave. Drums will be represented as a new block that will trigger a drum synth behind the scenes. Users will choose a drum by picture. Snap Music will curate a small set of other samples that will use symbols rather than names for the user to choose. The tempo will be modified using a slider control.

Other functionality will be added as well. The synth c-block will be added back and either use symbols or colors for choosing a specific synth. FX will work similarly. A limited number of controls will be added to synth and FX blocks that will directly modify specific parameters of a synth or FX so that the user can modify these parameters while coding. A chord block will be added that will use a visual interface similar to the play note interface to choose a chord to play. The list block will be overridden to allow adding multiple notes to a list in order to play them all at the same time.

One thing not decided upon yet is an interface for modifying the duration of a note.

TidalCycles has the ability to modify patterns using a number of built-in functions (TidalCycles website, McLean 2014) such as: rev to reverse a pattern, density to increase the speed of a pattern, slow to slow down a pattern, and iter to shift a pattern over multiple cycles. These functions and perhaps more are fun to use for a live coder to easily introduce changes over a pattern. In Snap Music these kinds of pattern modifiers can be implemented as c-blocks that surround a pattern for easy visualization of what the functions are modifying. This may be a situation where a small amount of typing will be needed in order to express a numerical parameter.

A major infrastructural modification that will be added is that all changes to a block stack will be re-evaluated immediately. The user should not need to press the green flag button in order to hear his or her changes.

Other functionality to manipulate the sound can be added, but care will need to be taken that any new blocks added should have a visual interface.

There have been thoughts about including two different block modes. One block mode has been described where all blocks have a visual interface with little to no typing required. The other block mode will be more advanced and would have blocks similar to the current blocks where typing is necessary. This could open up a wider range of possibilities for different, more advanced functions. However, it should be possible to create simplified visual interfaces for any function.

5. CONCLUSIONS

A live coding environment for younger children can be another tool to introduce coding to kids who are interested in creating their own music. Though this environment is only in its infancy, I expect to continue to develop it and have kids test it out to see how they use it. I would eventually like to use this environment to teach a beginning programming class. Kids should be able to have fun creating music that they like to hear while learning programming concepts at the same time.

The current implementation of this environment can be found at http://www.fradkin.com/snap-music.html.

Acknowledgments

The author would like to acknowledge Dr. Sam Aaron, Yotam Mann, and Jens Mönig and Brian Harvey for providing wonderful software that inspires creativity for kids.
REFERENCES


Tone.js website. http://tonejs.org

Snap! website. http://snap.berkeley.edu