INTERNATIONAL CONFERENCE ON LIVE INTERFACES

Colliding: a SuperCollider environment for synthesis-oriented live coding

Gerard Roma

CVSSP, University of Surrey Guildford, United Kingdom
g.roma@surrey.ac.uk

Abstract. One of the motivations for live coding is the freedom and flexibility that a programming language puts in the hands of the performer. At the same time, systems with explicit constraints facilitate learning and often boost creativity in unexpected ways. Some simplified languages and environments for music live coding have been developed during the last few years, most often focusing on musical events, patterns and sequences. This paper describes a constrained environment aimed at exploring the creation and modification of sound synthesis and processing networks in real time, using a subset of the SuperCollider programming language. The system has been used in educational and concert settings, two common applications of live coding that benefit from the lower cognitive load.

Keywords: Live coding, sound synthesis, live interfaces

Introduction

The world of specialized music creation programming languages has been generally dominated by the Music-N paradigm pioneered by Max Mathews (Mathews 1963). Programming languages like CSound or SuperCollider embrace the division of the music production task in two separate levels: a signal processing level, which is used to define instruments as networks of unit generators, and a compositional level that is used to assemble and control the instruments. An exception to this is Faust, which is exclusively concerned with signal processing. Max and Pure Data use a different type of connection for signals and events, although under a similar interaction paradigm. Similarly, Chuck has specific features for connecting unit generators and controlling them along time. The availability of languages capable of generating music in real time has fostered the development of live coding (Collins et al. 2003), which has the advantage of giving the audience the possibility to read computer music performances in a way that is comparable to improvisation with physical instruments. Live coding is also helpful in classroom environments, allowing students to grasp the mental process involved in using a programming language or command line. One feature that has inspired the live coding practice is the freedom and power that a programming language gives to the performer. On the other hand, restricted environments often result in unexpected creative outcomes. Facing the infinite possibilities offered by computers, musicians and artists commonly design their systems on the basis of constraints (Magnusson 2010). Constraints can be seen as the rules that define a game, and thus are considered by many to play an essential role in creativity (Boden 2004; Merker 2006).

Some constrained languages are available for live coding, most often with a strong focus on musical events (Magnusson 2011; McLean 2014). In the Music-N paradigm, this means not creating new instruments on stage, but improvising new control sequences for pre-defined instruments. The system presented in this paper explores the other side. In this sense, the concept of "synthesis-oriented" live coding can be opposed to "event-oriented" live coding. There is, as a matter of fact, a long tradition in challenging the distinction between composition and timbre, precisely on the basis of the possibilities offered by computers (Döbereiner 2011). Under this point of view, musical events can be seen as signals, and music can be created using exclusively signal processing networks.

With the release of version 3 (McCartney 2002), the SuperCollider language was split into two separate programs: the synthesis server (scsynth) and the language interpreter (sclang). A subset of the language is used to specify synth definitions, which the scsynth server can execute. It is not uncommon to find discussions among the SuperCollider community on creating music structures purely in the server side. Among other reasons, creating synth definitions can be less demanding with respect to dealing with the full language and the distributed architecture. Given the amount of unit generators available, focusing on the synthesis side of SuperCollider is both simple and powerful. In terms of user interface, the evolution of SuperCollider as a general purpose language led to a struggle between the need of a "proper IDE" for object-oriented programming, and the interest in the document-oriented rich text editor that was available on OSX.

In this context, Colliding was designed as a constrained interface for creating SuperCollider synth definitions. Apart from music creation and performance, the focus on synthesis provides a compelling environment for music and signal processing education settings, allowing easy experimentation with a wide variety of synthesis techniques. The idea of creating a simplified environment emerged when observing engineering students trying to create procedural programs with all the asynchronous calls required to start the server, create a synth definition and instantiate it. Loading audio buffers and network resources require asynchronous calls as well, which assume an understanding of anonymous functions. Both the classroom and the concert environments benefit from the reduced complexity and focus. In the case of music performance, the constraint is in part aesthetic, but still many different styles of music can be played. The rest of the paper describes the functionalities implemented so far and the use of the program in both education and performance.

Related work

Constrained environments for live coding are common in classroom-oriented applications. Two well-known examples are Earsketch (Freeman et al. 2014) and Sonic Pi (Aaron and Blackwell 2013). The first is based on an Application Programming Interface (API) which can be used in Javascript or Python in a web environment. This API is complemented with a library of audio loops that can be manipulated and positioned in an audio sequencer time line through the programming API. The second exposes also a basic API, in this case as a Ruby domain-specific language that controls SuperCollider synths. The programming environment is designed to run on a Raspberry-Pi embedded computer. Both environments offer limited capabilities in terms of synthesis. The main idea behind live coding music environments in the classroom is to engage students into programming by doing something fun and creative.

Such environments allow experimentation with generative music, but are not so well suited for synthesis-oriented music or learning. While these systems generally focus on learning programming skills, Colliding emphasizes signal processing, requiring only basic programming concepts. In this sense, perhaps a more similar approach would be using the compiled language Faust in an interactive setting. FaustLive (Denoux et al. 2014) is a just-in-time compiler aimed at facilitating this kind of set-up, however, it does not provide an interface for coding. Faust is used live by Julius O. Smith for teaching signal processing using Emacs[1].

As the name suggests, Colliding is mainly influenced by Processing (Reas and Fry 2006) and its cousin the Arduino IDE. Both have succeeded in creating simplified development environments for activities that traditionally required specialized training. Processing was originally presented as a subset of the Java language that can be embedded in Java programs. Arduino offers a simple C API for embedded systems. Similarly, Colliding uses a subset of the SuperCollider language.

Another major influence is ixiLang (Magnusson 2011). While heavily focused on musical events, ixiLang stresses the importance of reducing complexity for music live coding. Like ixiLang, Colliding can be seen as a "SuperCollider parasite", in this case for synthesis-oriented live coding.

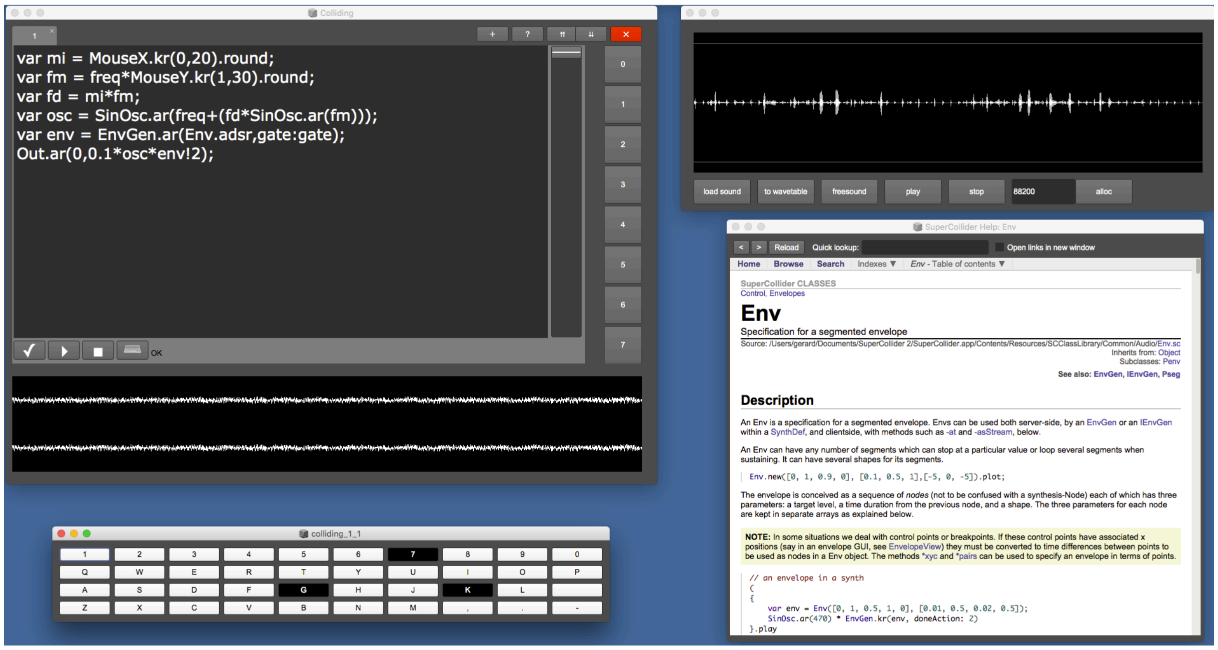
Interface

Overview

The program follows the interaction paradigm of prototyping editors like Processing. The interface (Figure 1) allows the user to create up to 8 tabs, each with a code editing window configured with a large font size. This encourages short snippets and facilitates readability. The number of tabs (and also of buffers, as described below) is not completely

[1] https://www.youtube.com/watch?v=2lEt7dsziO0

arbitrary. It is generally agreed that similar numbers of elements are related with working memory capacity (Miller 1956). It is common to find 8-channel limits in music production hardware (small mixers, old tape recorders) or software. The main actions consist in compiling the code (for error checking), running it (which results in a potentially infinite sound stream), and stopping the sound. These actions can be run through keyboard short-cuts or using a set of buttons below the code window. Another set of buttons in the top right corner exposes project-level operations (adding tabs, getting help for the currently selected text, loading and saving projects, and a panic button that stops all running processes). Text color in the code editing window is used to indicate compilation state. White means the code has been compiled, grey means it is being edited, and red means there is some error.



```
var mi = MouseX.kr(0,20).round;
var fm = freq*MouseY.kr(1,30).round;
var fd = mi*fm;
var osc = SinOsc.ar(freq+(fd*SinOsc.ar(fm)));
var env = EnvGen.ar(Env.adsr,gate:gate);
Out.ar(0,0.1*osc*env!2);
```

SuperCollider CLASSES
Control, Envelopes

Env

Specification for a segmented envelope

Source: /Users/gerard/Documents/SuperCollider 2/SuperCollider.app/Contents/Resources/SCClassLibrary/Common/Audio/Env.sc
Inherits from: Object
Subclasses: Penv

See also: EnvGen, IEnvGen, Pseg

Description

An Env is a specification for a segmented envelope. Envs can be used both server-side, by an EnvGen or an IEnvGen within a SynthDef, and clientside, with methods such as -at and -asStream, below.

An Env can have any number of segments which can stop at a particular value or loop several segments when sustaining. It can have several shapes for its segments.

```
Env.new([0, 1, 0.9, 0], [0.1, 0.5, 1],[-5, 0, -5]).plot;
```

The envelope is conceived as a sequence of *nodes* (not to be confused with a synthesis-Node) each of which has three parameters: a target level, a time duration from the previous node, and a shape. The three parameters for each node are kept in separate arrays as explained below.

NOTE: In some situations we deal with control points or breakpoints. If these control points have associated x positions (say in an envelope GUI, see EnvelopeView) they must be converted to time differences between points to be used as nodes in a Env object. The methods *xyc and *pairs can be used to specify an envelope in terms of points.

```
// an envelope in a synth
(
{
    var env = Env([0, 1, 0.5, 1, 0], [0.01, 0.5, 0.02, 0.5]);
    SinOsc.ar(470) * EnvGen.kr(env, doneAction: 2)
}.play
```

Figure 1: The Colliding interface

Colliding offers two modes of operation: "synth editing" and "advanced". The synth editing mode affords the metaphor of editing a patch in a hardware synthesizer, but it can also be used for live coding. In this case, the code is assumed to be a synth definition, and the environment provides some predefined variables and the means to trigger notes, if desired. In advanced mode, the code can be anything "playable". In SuperCollider this is achieved by compiling the code and making it the source of a NodeProxy, which allows using the interface with JITLib (Rohrhuber, de Campo, and Wieser 2005) as a back-end for live coding. This includes also event patterns, although this possibility has not been explored or specifically supported. One important difference is that in advanced mode the audio output channel is controlled by JITLib, while in synthesis editing mode it is specified explicitly. Thus in "synth editing" mode it is simpler to address multiple outputs. For synthesis oriented live coding, i.e., in SuperCollider talk, if only server code is evaluated, both modes can be used, although the advanced mode allows terser code.

A feedback panel below the code window and buttons is used for error reporting. Long stack traces can be overwhelming for beginners, and are also undesirable in live situations. When the code results in a stack trace, the system selects the relevant message and highlights the offending line (Figure 2). In order to provide such feedback without disrupting the current interpreter, it was necessary to implement compilation as a separate process, by writing the snippet to file and calling sclang to parse it. All of this happens under the hood.

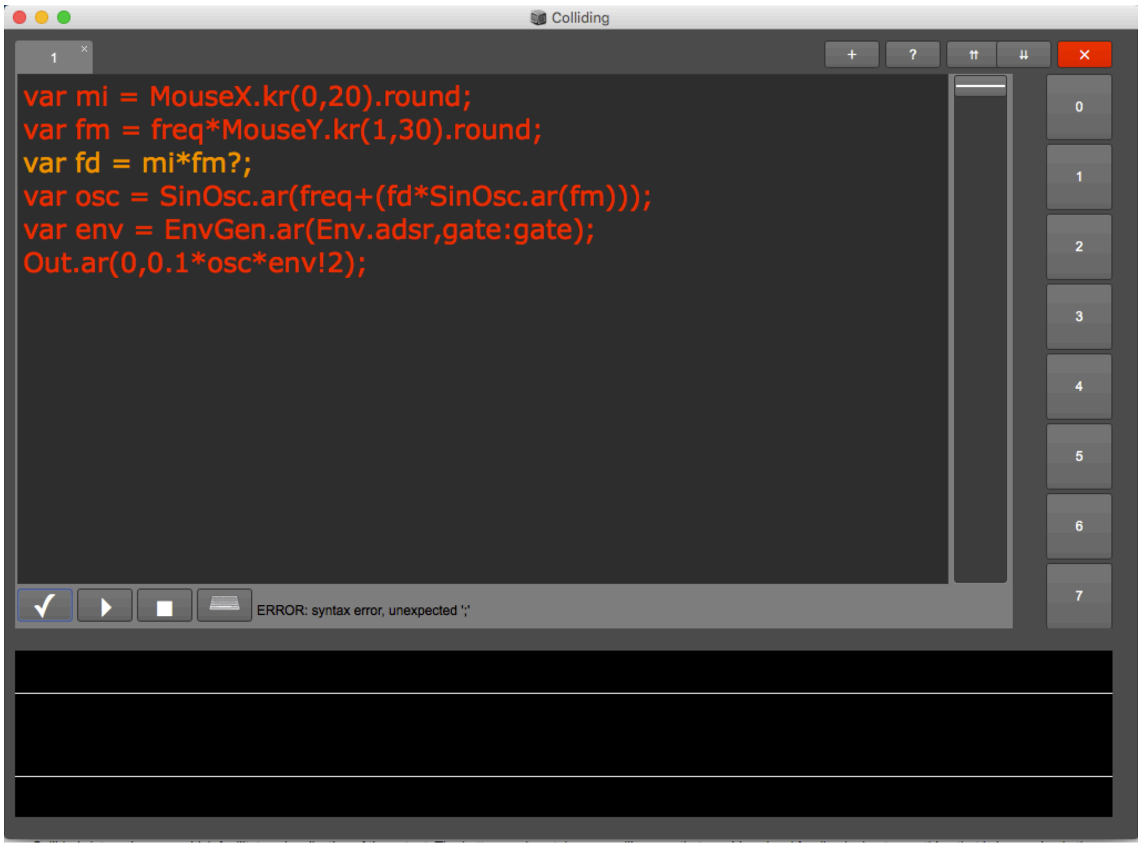
All code is run in SuperCollider's internal server, which facilitates visualization of the output. The bottom end contains an oscilloscope that provides visual feedback about everything that is happening in the server.

Synth editing

Additional features are provided in the synth editing mode. In this mode, the code is wrapped to build and instantiate a synth definition, with several pre-defined parameters. A window representing the computer keyboard allows triggering notes using the synth that has been defined in the current tab. The keyboard uses an isomorphic mapping. A slider next to the code window allows controlling the gain. This results in the following pre-defined variables. It is up to the coder to make use of them.

- key: The MIDI number corresponding to the key pressed in the computer keyboard. 4
- freq: The frequency in Hz corresponding to the key.
- gate: A gate input for envelopes.
- amp: The value defined by the slider.

It is trivial to extend this system with other widgets, for example additional sliders could be attached to the keyboard. For the moment, the system provides the minimum to leverage the input devices already available in every computer. The mouse is accessed using the traditional SuperCollider unit generators. This is a convenient setup for classroom PCs and laptop performances. It is also easy to extend the concept to MIDI controllers.



```
var mi = MouseX.kr(0,20).round;
var fm = freq*MouseY.kr(1,30).round;
var fd = mi*fm?;
var osc = SinOsc.ar(freq+(fd*SinOsc.ar(fm)));
var env = EnvGen.ar(Env.adsr,gate:gate);
Out.ar(0,0.1*osc*env!2);
```

```
ERROR: syntax error, unexpected ';'
```

Figure 2: Error reporting

Buffers and files

Besides synth definitions, another area that requires asynchronous calls is dealing with audio files. This is often a source of confusion in SuperCollider. Graphical interfaces are generally appreciated for browsing the file system and accommodating the time for file loading. Colliding provides 8 slots for buffer management. Buffers are simply accessed via their buffer number (0-7). A separate window is used for loading, visualizing and pre-listening each buffer. Empty buffers can be allocated for recording signals, and small buffers can be converted to wavetable format for wavetable synthesis. In addition to loading sound files from disk, the system includes an interface for the Freesound quark[2], so that the user can make basic text queries to the Freesound database (Akkermans et al. 2011), and download sounds. For the moment, the search is restricted to wav files. However, if ogg support is compiled with the SuperCollider binaries, it should be possible to access the whole Freesound database via high-quality ogg previews[3].

Finally, some basic project management facilities are provided. A project consists of the text in each tab and all the sound files corresponding to the buffer slots. This is especially useful for learning environments, but can also be useful to preserve the state of a live coding session. In the future it might be interesting to add some simple interface to version control (e.g. git) which is also usually hard for beginners.

Initial Experiences

Colliding was developed while the author was at the Sonology department at Escola Superior de Música de Catalunya (ESMUC). While no formal evaluation was conducted, it was used in 4 overlapping semester courses between 2013 and 2015, covering basic and advanced synthesis techniques. During the previous year, the SuperCollider language and IDE were used. The difference with respect to using plain SuperCollider was dramatic. By freeing students from learning to use the SuperCollider editor and language, it was possible to focus on synthesis and processing as opposed to programming. At the same time, thanks to the large number of unit generators available and the general power of SuperCollider it was possible to explore many different algorithms and concepts, from basic subtractive and modulation techniques to physical and spectral synthesis. Colliding was used as a kind of lab notebook editor that could run in classroom PCs and student laptops. Assignments consisted in Colliding projects where different variants were explored in each tab. A small code snippet was usually seeded.

Since students were generally beginners in computer programming, a very simple grammar was adopted which worked for all assignments (as can be seen in the figures): each line, except the last one, is a variable definition and assignment, where part of the signal processing graph is represented. This suggests that, for the purpose of defining synth definitions, the language could be further simplified.

The only limitation was the project management feature, which was not always trusted by the students because of its simplicity. However in case of doubt the project structure was simple enough and text snippets could be open with a standard text editor. Some synthesis techniques that require additional processing, such as in the case of wavetable synthesis, were more difficult to accommodate. For advanced techniques, such as spectral modeling or vocoding, it was generally convenient to use "black-box" unit generators available from sc3-plugins[4].

Colliding has also been used in several live coding performances by the author. Up to 8 synthesis processes may run in parallel while one of them is being edited. The big font, oscilloscope, and the constraint to synthesis-based coding contribute to the readability of the performance. In live coding, the simplified grammar is not needed, and instead a terse syntax is more common, in line with the use of Twitter for sharing small SuperCollider programs.

Conclusions

The SuperCollider syntax for building synthesizers allows for creating sophisticated patches with very few lines of code. Such concise descriptions often contrast with the complications one ends dealing with when trying the same things with graphical patching systems and modular synthesizer metaphors. However, learning an object-oriented language

[2] https://github.com/g-roma/Freesound.sc
[3] This depends on the version of libsndfile, which for historical reasons was fixed to an older version on OSX binaries. At the time of this writing the problem is fixed in SuperCollider 3.7 binaries, so the restriction to wav for Freesound searches will be removed.
[4] https://github.com/supercollider/sc3-plugins

enriched with functional programming constructs can easily scare tinkerers and musicians with little programming background. The Colliding environment provides a simplified interface that is particularly useful when focusing on sound synthesis and processing, as opposed to event-based music composition and performance. This focus can be seen as a design constraint that is useful in educational environments and in synthesis-oriented live coding. While the current implementation mainly stresses the concept and interface, these ideas can be further extended without leaving the SuperCollider language, by defining more helper variables and functions. Also, the system is amenable to encapsulation of synthesis processes as unit generators, a practice used by some live coders which allows growing a personal sonic palette without losing simplicity. Adding support for this kind of encapsulation will be investigated in the future. The code is available as a SuperCollider quark, and can be downloaded from the github repository[5].

Acknowledgements

The author was supported by Escola Superior de Musica de Catalunya (ESMUC) and the Music Technology Group (MTG) at Universitat Pompeu Fabra (UPF) during the development of this work.

References

Aaron, Samuel, and Alan F Blackwell. 2013. “From sonic Pi to overtone: creative musical experiences with domain-specific and functional languages.” Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design. ACM, 35–46.

Akkermans, Vincent, Frederic Font, Jordi Funollet, Bram De Jong, Gerard Roma, Stelios Togias, and Xavier Serra. 2011. “Freesound 2: An improved platform for sharing audio clips.” Klapuri A, Leider C, editors. ISMIR 2011: Proceedings of the 12th International Society for Music Information Retrieval Conference; 2011 October 24-28; Miami, Florida (USA). Miami: University of Miami; 2011. International Society for Music Information Retrieval (ISMIR).

Boden, Margaret A. 2004. The creative mind: Myths and mechanisms. Psychology Press.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. “Live coding in laptop performance.”

Organised sound 8 (03): 321–330.

Denoux, Sarah, Stéphane Letz, Yann Orlarey, and Dominique Fober. 2014. “FAUSTLIVE, Just-In-Time Faust

Compiler... and much more.” Technical Report, GRAME.

Döbereiner, Luc. 2011. “Models of constructed sound: Nonstandard synthesis as an aesthetic perspective.” Computer Music Journal 35 (3): 28–39.

Freeman, Jason, Brian Magerko, Tom McKlin, Mike Reilly, Justin Permar, Cameron Summers, and Eric Fruchter. 2014. “Engaging underrepresented groups in high school introductory computing through computational remixing with EarSketch.” Proceedings of the 45th ACM technical symposium on computer science education. ACM, 85–90.

Magnusson, Thor. 2010. “Designing constraints: Composing and performing with digital musical systems.” Computer Music Journal 34 (4): 62–73.

Magnusson, Thor. 2011. “the Ixi Lang : a Supercollider Parasite for Live Coding.” Icmc 2011, no. August:503–506.

Mathews, Max V. 1963. “The digital computer as a musical instrument.” Science 142 (3592): 553–557.

[5] https://github.com/g-roma/Colliding

McCartney, James. 2002. “Rethinking the computer music language: SuperCollider.” Computer Music Journal 26 (4): 61–68.

McLean, Alex. 2014. “Making programming languages to dance to: live coding with tidal.” Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design. ACM, 63–70.

Merker, Bjorn H. 2006. “Layered constraints on the multiple creativities of music.” Musical Creativity: Multidisciplinary Research in Theory and Practice, pp. 25–41.

Miller, George A. 1956. “The magical number seven, plus or minus two: some limits on our capacity for processing information.” Psychological review 63 (2): 81.

Reas, Casey, and Ben Fry. 2006. “Processing: programming for the media arts.” AI & SOCIETY 20 (4): 526–538.
Rohrhuber, Julian, Alberto de Campo, and Renate Wieser. 2005. “Algorithms today notes on language design for just in time programming.” International Computer Music Conference. 291.