Soundcipher: A music and sound library for processing

Author

Brown, AR

Published

2009

Conference Title

Proceedings of the 2009 International Computer Music Conference, ICMC 2009

Version

Version of Record (VoR)

SOUNDCIPHER: A MUSIC AND SOUND LIBRARY FOR PROCESSING

Andrew R. Brown

Queensland University of Technology,
Creative Industries Faculty

ABSTRACT

SoundCipher is a software library written in the Java language that adds important music and sound features to the Processing environment that is widely used by media artists and otherwise has an orientation toward computational graphics. This article introduces the SoundCipher library and its features, describes its influences and design intentions, and positions it within the field of computer music programming tools. SoundCipher enables the rich history of algorithmic music techniques to be accessible within one of today's most popular media art platforms. It also provides an accessible means for learning to create algorithmic music and sound programs.

1. INTRODUCTION

SoundCipher is a Java software library that facilitates the integration of music and sound into media art works created with Processing [1]. It is built on top of the JavaSound API and provides an easy-to-use interface to many of the features of JavaSound and presents them in a style sympathetic to the conventions of Processing.

Processing is a Java programming environment that provides infrastructure and functionality allowing users, typically media artists, to focus on the core algorithms of their work while hiding some technical details, such as animation threads and class instantiation.

The SoundCipher library augments Processing with music and sound extensions that provide structures and functions to support algorithmic composition. There were a number of design goals for the SoundCipher library including simplicity, familiarity, extendibility, ease of learning, and real-time interaction. The remainder of this section will outline these design decisions in more detail.

Many of the design decisions were taken to facilitate ease of use for Processing users. As a result, SoundCipher commands are relatively abstract compared to the underlying JavaSound interface upon which they are primarily built. For example, the function `playNote()` encapsulates the process of creating two musical events, within a track, within a sequence, and within a sequencer. The size of the library is intentionally small to minimize potential confusion and distraction. For example, there are only two classes that can be instantiated, the `SoundCipher` and `SCScore` classes. The 'style' of SoundCipher syntax generally conforms to the programming conventions of Processing, which differ from those often found in Java. An example of this is the use of public, rather than private, class variables and unconventional accessor method conventions. Processing uses direct accessing of variable values and the absence of 'set' as a preface to setter methods.

Given that novice programmers, who may already have more musical than computing background, may use SoundCipher, the library uses familiar musical terms and metaphors where possible. This is intended to provide an infrastructure to support 'musical' thinking. The library has a sound-event orientation using conventional terms that result in functions, such as `playNote()`, `addChord()`, `tempo()`, and `instrument()` while the main container for these events and parameters is a 'score'. Temporal organization is calculated using musical units of beats and speed is set in beats per minute.

Like most programming libraries SoundCipher allows users to extend its functionality. In particular, however, SoundCipher has been designed to integrate external interaction via MIDI and to support relevant Processing libraries for audio signal processing and OSC communications. Specifically, the SoundCipher scheduling system can be used to synchronize these and any drawing or animation functions to musical time independent of Processing's frame rate.

As well as supporting the production of audio-visual media art works, the SoundCipher library was conceived as a tool for learning about algorithmic music. The ease of use of both Processing and SoundCipher make media programming accessible to novice users. There is also a commitment to tutorial support that goes beyond describing how the library works, to include introductions to algorithmic music concepts and techniques.

While Processing, and the Java language more generally, are not completely dynamic (programs require compiling) there is still an emphasis in Processing on real-time processes, such as animation, to create interactive media art. SoundCipher supports real-time generation, interaction, and dynamic change at runtime, by supporting responsive and efficient event processing with the `playXXX()` commands for notes, phrases, chords, and audio files. The SoundCipher scheduler in the `SCScore`

class supports timed callbacks that can be used to control program flow at runtime in quite flexible ways, allowing programs to adapt to real-time changes and events.

2. BACKGROUND

SoundCipher is far from being the first music and sound software library and, therefore, it is useful to reflect for a moment on how it draws upon the tradition of computer and algorithmic music systems; in some cases quite directly and in others less so.

Music programming libraries have existed at least as far back as 1956 with the CSIRAC computer which had a library function to play notes [2]. In 1957 Max Matthews wrote the first of his famous Music N (I-V) family of libraries [3]. These libraries, and many that followed, were largely concerned with signal processing and synthesis as a way of making a sound. The event-orientation found in SoundCipher emerged gradually and was well established by the time of the Csound library [4] which clearly differentiated 'instrument' code from the 'score'; a practice with origins in the Music N libraries. Adherence to naming conventions from conventional music, such as note, phrase or part and score, as used in SoundCipher, were well established by the time of the Music Kit library [5] for the NeXT computer in the late 1980s. These early libraries were generally written in the C family of languages but it seems that today every widely used (and even some obscure) languages have a music and/or sound library written in them. Given the popularity of the Java language it is not surprising that it has many music and sound libraries [6], and even Processing has several; some of which are discussed later in more detail.

There are two music and sound programming libraries with which the author has been closely associated and which have had quite a direct influence on the design of SoundCipher. These are jMusic [7], a Java library for music composition, and Impromptu [8] a real-time media programming environment using the Scheme language.

The two most obvious indications of jMusic heritage in SoundCipher are the use of the Java language and the adherence to a note/phrase/score metaphor. In some places SoundCipher borrows minor elements from jMusic, for instance in the Constants classes that define frequently used variables such as General MIDI instrument values and drum mappings. Less obviously, experience with jMusic highlighted the value of tutorials for usability, especially given that jMusic also had a strong pedagogical intent. In contrast to jMusic the SoundCipher library focuses on a minimal set of core functionality rather than an extensive array of features and utilities.

The Impromptu heritage has even more impact on SoundCipher because it was Impromptu that inspired the architectural priorities in support of real-time interaction and dynamic flexibility. In some sense this is still aspirational on the part of SoundCipher because the Java language and Processing environment are not as dynamic as Scheme in Impromptu, and the real-time scheduling in Java is not as refined as Impromptu's scheduler written in C. SoundCipher's architecture also inherited a more functional programming approach (like Scheme) from Impromptu rather than the highly object oriented approach typical of Java and quite evident in jMusic. In contrast to Impromptu, the SoundCipher library is significantly more compact because it leverages the Java/Processing context and is much less ambitious in its scope.

Other music and sound libraries exist for Processing, so to conclude this section we will examine how SoundCipher fits within this landscape. At the time of writing the Minim library [9] is the most prominent sound library for Processing, largely because it is included with the standard distribution of Processing 1.0. Minim focuses on audio file playback and signal processing, in the main, and provides extensive control over sound manipulation. Minim and SoundCipher are complementary and can be integrated quite straightforwardly. Other libraries oriented toward signal processing include p5_sc [10] that provides Processing bindings for SuperCollider [11], Ess [12] that provides similar audio file and signal processing functionality to Minim, and Sonia [13] that uses the jSyn [14] plugin for audio synthesis rather than JavaSound.

There are a couple of event-based music libraries, more like SoundCipher, available for Processing. Tactu5 [15] is most similar in features to SoundCipher in that it supports note, phrases and chords (albeit using different terms) and provides similar constants and utility methods. However, it does not connect to JavaSound for playback but rather relies on another library to send events to external sound sources, for example via OSC. A prominent differentiator is that Tacu5 lacks the dynamic flexibility of SoundCipher's callback architecture. Lastly, there is the jm-etude library [16] that provides a Processing binding for elements of jMusic. The jm-etude library builds scores with the jMusic data structure and plays them back with the JavaSound synthesizer. It follows a similar note-event metaphor to SoundCipher and Tacu5 and inherits from jMusic a limited ability for dynamic variation at runtime.

3. ELEMENTS AND USES OF SOUNDCIPHER

The SoundCipher library consists of a small number of Java classes and interfaces that define various functionality and syntax. At its core is a scheduler, built on the JavaSound sequencer class, that controls timing during playback. Events are lodged with the scheduler via methods in the `SCScore` class. There are two types of events, notes (proxies for JavaSound synthesizer messages) or callbacks (proxies for JavaSound meta events). SoundCipher provides 'chord' and 'phrase' methods that correspond respectively to simultaneous or sequential note sets.

The SoundCipher library has been designed to fit into a range of use scenarios from the simple to the complex and the fixed to the flexible. The usage patterns described in this section have been arrived at through years of experience in algorithmic music making and can serve the demands of most situations in which media artists will find themselves, including exhibitions, installations and performance. The remainder of this section will discuss these elements and their uses.

3.1. SoundCipher Class

The `SoundCipher` class provides a quick way to play music and sounds in Processing. It provides methods for playing a single note, a phrase (a sequence of notes), or a chord (a cluster of notes) using JavaSound's built-in synthesizer or via MIDI output to an external (hardware or software) synthesizer. The class also provides methods for simple playback of a specified audio file. These methods handle score construction and event scheduling in a transparent way where required. The `SoundCipher` class assumes that only one event, or set of events in the case of chord, is playing at one time on a single instrument. For polyphonic or multi-part music the user can either create several SoundCipher instances, one for each part, or construct the music using the `SCScore` class directly (see section 3.4).

These SoundCipher methods are particularly useful when sounds need to be triggered by other events in the program, typically visual cues, animation collisions, or user interactions. While this capability is simple to use it does not scale well, and even moderately elaborate musical passages become tiresome to describe with a long list of individual `playNote()` commands. The timing of `playXXX()` methods can be controlled by non-musical events such as Processing's `draw()` method frame rate, visual cues, or user interactions.

3.2. Constants Interfaces

SoundCipher provides a range of constants; text strings that stand for particular numerical values. Most variables in code and values in protocols such as MIDI and OSC are numbers, and the mapping of numbers to attributes can be awkward to remember, and long lists of numbers in method arguments can make code difficult to read for the uninitiated. The use of constants helps to alleviate this. Constants are currently provided for General MIDI program changes, and drum key mapping, and pitch-class sets. Following Java conventions, constants are written in upper case for easy identification.

The `PitchClassSet` constants correspond to arrays of numbers that indicate particular scales or modes based on C as a root. For example, MAJOR = {0, 2, 4, 5, 7, 9, 11} which outlines the scale degrees for that mode. These pitch class sets are used by some of the utility methods and are handy for many algorithmic processes dealing with the western harmonic principles. For musicians preferring to deal with different harmonic systems these constants and utilities can be ignored. Both the `SoundCipher` and `SCScore` classes implement the constants interfaces, resulting in, for example, `sc.TROMBONE` where `sc` is an instance of the `SoundCipher` class.

3.3. Utilities Class

The `SCUtilties` class contains a collection of helper methods for users of the SoundCipher package. In particular the utilities provide support for common musical and mathematical functions. The `SCUtilities` class is not designed to be instantiated directly, but can be accessed via the `SoundCipher` or `CScore` classes which inherit the utilities; for example `sc.gaussian()` where `sc` is an instance of the `SoundCipher` class. At present the utilities consist of functions including a Gaussian random distribution and MIDI to frequency conversion, along with pitch-class set operators such as `pcRandom()`, that selects a constrained random pitch related to a set.

3.4. SCScore Class

The `SCScore` class provides music data structure and scheduling services. A SoundCipher score can contain notes, phrases (note sequences), chords (note clusters), or callbacks that are added with an associated ID value. These objects are added to the score with a playback time specified in beats relative the start of the score. A score can be played back once, repeated or looped. Notes in a score play back using the internal JavaSound soundbank, while callbacks can be used for any arbitrary synchronized purpose (e.g., audio file playback, drawing, sound synthesis parameter control, and OSC or MIDI message sending). Callback messages are parsed by the `handleCallbacks()` method which needs to be added to the Processing program. Users of a `SCScore` instance should always include a 'stop' method in the Processing program that calls the `stop()` method of the `SCScore` instance to halt playback when the program exits. This ensures that scores with future commitments come to an end when the program is exited.

Any number of events can be added to a score and it is typical that algorithmic processes are created to generate and add events to the score. The score's tempo is set in beats per minute and can be modified during playback.

A limitation of the score composition approach is that event commitments are made into the future, which may limit responsiveness and interaction. For fixed works or those without runtime variation or interaction, this will cause no problem, however many media art works created with Processing make a feature of generative processes and interaction.

3.5. Real-time Generative Music

A solution to the issue of undue future commitment is to continually reuse a very short score that minimizes future commitments but still allows SoundCipher to maintain timing control. A short score, just one or two beats in length, can be updated and replayed repeatedly to provide a useful cyclic structure for near-to-real-time responsiveness in the music. With this 'streaming' strategy music is generated by an algorithm in segments of an appropriate length, and a callback is placed at the end of the score and, when parsed, triggers the next cycle of emptying, generation and playback.

3.6. Synchronizing Activities

Using callbacks, SoundCipher's score playback can trigger any arbitrary event (even another score) and this allows scores to become the temporal hub of entire media art works; specifying sound and visual elements with more precision, efficiency and flexibility than Processing's `draw()` thread typically allows. Callbacks can also be used to control MIDI and OSC message-sending, which allows the score to trigger or coordinate activities with other music applications or hardware. Interestingly, and more abstractly, callbacks can be used to execute any Java code at a specified time. Another use of callbacks is to coordinate activities with the Minim library to control sound synthesis and audio signal processing functions using SoundCipher's scheduler.

3.7. Using SoundCipher Outside of Processing

While SoundCipher has been specially created for use within Processing it can also be used as a stand-alone library in any Java application. Thus it can provide the general Java developer with an easy-to-use interface to JavaSound functionality and algorithmic music facilities.

4. CONCLUSION

SoundCipher attempts to make accessible to users of Processing many useful algorithmic music and sound techniques and features. This article has introduced SoundCipher, a music and sound library for the Processing environment, and has discussed how it inherits attributes from a long history of algorithmic computer music systems to arrive at a succinct, but powerful, set of features including a flexible and accurate event scheduling and the representation of music using common musical metaphors.

SoundCipher emphasizes elements that make it fit within the Processing 'ecosystem' and is complementary to existing Processing features and audio libraries. The compact design of SoundCipher reflects a distillation of many decades of event-based music library exploration and its features acknowledge the important role of music and sound for interactive generative media programming.

SoundCipher is available for free and distributed under the GNU GPL license. The library, tutorials and other information are available online at http://soundcipher.com

5. REFERENCES

[1] Reas, C. and Fry, B. (2003). "Processing: a learning environment for creating interactive Web graphics" in Proceedings of the *International Conference on Computer Graphics and Interactive Techniques*, San Diego. ACM SIGGRAPH, pp. 1-1.

[2] Doornbusch, P. (2005). *The Music of CSIRAC: Australia's first computer music*. Melbourne, Common Ground.

[3] Matthews, M. (1997) *Horizons In Computer Music*. http://www.cs.indiana.edu/horizon/970308/

[4] Vercoe, B. (1986). *Csound*. Cambridge, MA, Experimental Sound Studio, MIT.

[5] Jaffe, D. and Boynton, L. (1989). "An Overview of the Sound and Music Kits for the NeXT Computer." *Computer Music Journal*, 13(3): 48-55.

[6] Erbe, T. (1997-2008) "Java Music Projects - Synthesis | Composition | MIDI." Accessed 1 February 2009. http://www.softsynth.com/links/java_music.html

[7] Sorensen, A. and Brown, A. R. (2000). "Introducing jMusic" in Proceedings of *InterFACES: The Australasian Computer Music Conference*, Brisbane. ACMA, pp. 68-76.

[8] Sorensen, A. (2005). "Impromptu: An interactive programming environment for composition and performance" in Proceedings of the *Australasian Computer Music Conference 2005*, Brisbane. ACMA, pp. 149-153.

[9] Di Fede, D. (2007) *Minim*. http://code.compartmental.net/tools/minim/

[10] Jones, D. (2006) *p5_sc*. http://www.erase.net/projects/p5_sc/

[11] McCartney, J. (1996). "SuperCollider: A new real-time sound synthesis language" in Proceedings of the *International Computer Music Conference*, Hong Kong. International Computer Music Association, pp. 257-258.

[12] Olsson, K. (2007) *Ess*. http://www.tree-axis.com/Ess/

[13] Pitaru, A. (2007) *Sonia sound library for java and processing*. http://sonia.pitaru.com/

[14] Burke, P. (1998). "JSyn - A Real-time Synthesis API for Java" in the Proceedings of the *International Computer Music Conference*, Ann Arbor, Michigan. ICMA, pp. 252-255.

[15] Capozzo, A. (2007) Tactu5: A Processing music assistant. http://www.abstract-codex.net/tactu5/

[16] Dihardja, D. (2006). jm-etude: jMusic wrapper for composing music with Processing. http://jmetude.dihardja.de/