aa-cell IN PRACTICE: AN APPROACH TO MUSICAL LIVE CODING

Andrew Sorensen
MOSO Corporation
Brisbane, Australia
andrew@moso.com.au

Andrew R. Brown
Computational Arts Research Group
Queensland University of Technology
Brisbane, Australia
a.brown@qut.edu.au

ABSTRACT

Live coding performances provide a context with particular demands and limitations for music making. In this paper we discuss how as the live coding duo aa-cell we have responded to these challenges, and what this experience has revealed about the computational representation of music and approaches to interactive computer music performance. In particular we have identified several effective and efficient processes that underpin our practice including probability, linearity, periodicity, set theory, and recursion and describe how these are applied and combined to build sophisticated musical structures. In addition, we outline aspects of our performance practice that respond to the improvisational, collaborative and communicative requirements of musical live coding.

1. INTRODUCTION

For the past two years we have been performing as aa-cell, a live coding duo, in regular concerts around Australia. Our performances are semi-improvised collaborations using the Impromptu environment, developed by Andrew Sorensen.

Live coding is a practice where software that generates music and/or visuals is written and manipulated as part of the performance. It emphasises the expressive possibilities afforded by programming languages as a means for defining and manipulating music and/or visual processes. Live coding of music is a-stylistic in principle, although as with all practice, a chosen medium leaves its imprint on output. In this paper we will explain how we manage the effect of medium and intent in our musical practice.

Collaborative live coding places particular demands on the creative process because it requires a shared vocabulary to facilitate directed musical exploration. In this paper we will outline the results of our approach to working effectively within the constraints of our practice in the hope that the techniques we have developed (or accumulated) will inform other computer musicians and, perhaps more importantly, may highlight issues regarding, a) parsimonious computational representations of music and, b) practical approaches to interactive computer music.

2. BACKGROUND

Discussions of live coding as a practice have come to the fore in recent years [14][3][10][13]. The practice of live coding, building code structures during performance, is similar to what Wang and Cook [33] call on-the-fly-programming, and McLean [13] refers to as just-in-time programming. Most of these discussions have focused on the dynamic nature of live coding and how various programming environments have been developed to facilitate live coding. They emphasise the need for a highly interactive environment, the ability to dynamically vary processes at runtime, and a strong concern for robust and flexible timing structures. In this paper we will discuss how we exploit these and other features of the Impromptu environment and leave detailed technical discussion about how this is achieved to other previous [27] and forthcoming papers.

With regard to live coding practice some approaches and issues are discussed by Collins [14] and Collins and Olofsson [15]. Collins [14] focuses especially on the performer/audience relationship and aa-cell adopt a similar stance to that suggested by Collins (after McLean [13]) where code is projected and other measures are taken to engage the audience. The article by Collins and Olofsson is particularly concerned with their audio visual live coding and the capture, segmenting and synchronizing of audio and visual material during performance. In aa-cell's practice (so far) there is no coding of visuals, however, synchronisation is important to our collaboration even though it is more directed and less agent-based than that discussed by Collins and Olofsson.

Looking further afield, aa-cell's live coding practice is informed by work in interactive music [25] [32] and hyperimprovisation [17] with which it shares an interest in the role of the computer in live performance. Our approach differs from this work not only in its "on the spot" programming but also with regard to the role of the computer. Our practice is less concerned with instilling the machine with musical "intelligence" to listen and respond, and is more akin to composition in real-time. The literature on algorithmic composition is, therefore, a rich source of inspiration for our work (Hillier and Isaacson [20], Xenakis [34], Berg [1], Dodge and Jerse [19], Cope [16], Taube [30]).

The approach of aa-cell to live coding combines composition and performance practices. This is in contrast to the batch-compile process that Paul Lansky once referred to as "sort of improvising in real-slow time" [9]. Rather, live coding for aa-cell is composition in real-time. However, it is not enough to simply consider the compositional aspects of the process. Live coding is a performance practice and we must also control the algorithmically generated material.

Our approach revolves around setting up generative processes, and the dynamic nature of live coding allows the performer to direct these processes. Live programmers not only write the code used to generate the music, they also constantly change and modify the behavior of that code dynamically throughout the performance. In this way the live programmer controls higher level structure, directing processes like a conductor directs an ensemble.

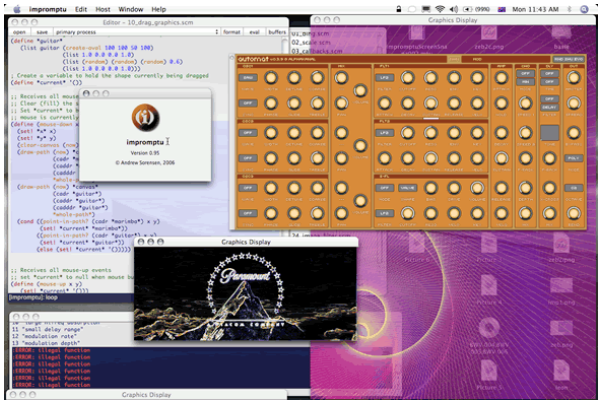

Figure 1. The Impromptu development environment

This is, for us, one of the most interesting aspects of our performance practice. The programmer is intimately involved in not only defining the process (as in standard computer music programming practice) but also in controlling its execution and evolution.

This focus on directing a real-time musical outcome through code has required us to seek out flexible computational control structures, and simple processes that can be combined to yield rich musical results. In particular we have settled on probability, linearity, periodicity, set theory, and recursion as useful techniques. While these techniques are not unique, we have settled on them after exploration of numerous approaches [26][4][5][6][7][8]. We have found that by re-engaging with these techniques in a dynamic performance context we achieve new levels of intimacy with these processes and with code as a medium of musical expression.

3. MUSICAL COMPUTATIONS

The search for pattern and structure in music has produced a large body of research from both the computer music community and the music theory community at large. Generally this research has been focused on analysis, an identification of structure after the fact. While this is an important approach (much maligned by undergraduate music students) the structural descriptions can often be quite detailed, and live coding requires succinct description. As well, many of the results of music analysis are contextually limited, whereas for live coding we need the power of fewer generic methods wherever possible.

Overall, succinct representation has become a central tenant of our live coding practice. The limitations of how much typing can be done during a live performance mandate parsimonious solutions for both musical and systems design considerations. As well the improvisational nature of the practice demands that techniques are memorised, so the utility of a limited set of processes across a variety of circumstances is important.

Also of importance is modularity. In aa-cell's live coding practice performances are constructed by building up complexity over time. In order to facilitate this process, it has been necessary to have a set of techniques that can be combined in a variety of ways to form musical patterns at multiple levels of hierarchy. In a sense, these form the atomic elements of our musical construction.

Live coding practice requires that we become fluent with these tools, so that we are free to concentrate on musical, rather than technical, expression. In a sense these base processes have become the symbols of a new dynamic scoring environment based in code, to be manipulated symbolically as composers may manipulate structures in more traditional common practice notation.

Far from feeling restricted by simpler techniques, we have been energised by this re-engagement with elementary processes. These primary functions have re-introduced an intimacy of process and a connection with medium; often lost in the higher level abstractions of more complex processes. This effect maybe due to a forced return to a more restricted compositional tool-set, a release from the tyranny of choice, or possibly due to our perception of a closer fundamental relationship to structure. Whatever the deeper reasoning, this parsimonious approach has resonated with us and has resulted in a live-coding vocabulary centred around elementary notions of probability, linearity, periodicity, set theory, and recursion. We will briefly explore some of these "atomic" processes and provide some practical examples of their use in aa-cell's practice.

3.1. Probability

It is 50 years since Leonard Meyer wrote "Emotion and Meaning in Music", a seminal work in the field of music perception. In this work, Meyer [23] contends that musical style is a system of sound relationships commonly understood and used by a group of people. He describes these sound relationships as "complex systems of probability relationships in which the meaning of any term or series of terms depends upon it's relationships with all other terms possible within the style system" [23]. Computer music composers and theorists have made extensive use of probabilistic techniques from the birth of computer music in the 1950's [20] through to the present [31][21]. we draw on this heritage to make extensive use of probabilistic techniques in our live coding practice.

One revealing approach that we have found is the "random" test. If we replace *algorithm X* with a random number generator, do we achieve any degradation in output and, if so, what scale of degradation? What is the musicality of *algorithm X* compared to noise? In our explorations to date we have found this to be an extremely revealing test. Our experience suggests that many algorithms, especially those not derived from music analysis, do poorly in this test. Often we have found that the mapping of data to parameters is more musically significant than the data being consumed.

As a result we have found that simple probabilistic functions, in particular linear and gaussian distributions, can provide effective variety and interest. It is worth noting that our use of probability is often subtle and usually highly constrained. Randomness provides two useful functions in our practice, 1) the function of non-determinism for the provision of structural novelty and variation and, 2) the less commonly discussed role of abstraction, where it provides the most practical means for abstracting away many of the details of performance nuance and human inaccuracy. We do not mean to suggest that a simple random selection is the best computational mechanism for handling performance parameters; only that when used subtly it can often

provide adequate variation, making it a useful tool for live programming.

```
;; A Simple diatonic progression - 1st Order Markov
(define chords
   (lambda (degree)
      (play-chord 60 80 3
                  (pc:diatonic 0 '^ degree)
                  *second*)
      (callback (+ (now) *second*) 'chords
                (random (cdr (assoc degree
                                    '((i v7 iv ii)
                                      (ii v7)
                                      (iv ii v7 i)
                                      (v7 i))))))))
```

The example above demonstrates how simple probabilistic techniques can be used to succinctly realise musical goals. A simple first order Markov model from diatonic musical theory provides a simple process for harmonic progression. Most of the examples that follow will also make use of probability.

3.2. Linear & Higher Order Polynomials

In his book "Structural Functions in Music", Wallace Berry [2] outlines some of the relationships between Linear function and musical structure. He discusses these relationships in pitched, rhythmic and timbral material at multiple perceptual levels [2]. Linear functions are generally applied to abstract representations of features that are often based on perceptual scales, even if the underlying physical properties are non-linear. For example, chromatic or diatonic pitch organisation can be linear even though pitch frequency relationships are not. aa-cell use linear functions extensively across all musical elements; pitch, duration, amplitude, timbre and so on. We also make use of higher order polynomials and splines.

Linear functions are often applied via break point envelopes to provide temporal structure at many levels in an aa-cell performance, ranging in duration from milliseconds (microstructure) to an entire work (macrostructure) [34]. The code below demonstrates a simple looped pitch cell generated from an envelope. Although this is a trivial example it does outline two important practical benefits that envelopes provide for outlining pitched material; (a) they naturally coordinate rhythmic and pitch variation and (b) by changing the modulus values it is possible to loop subsections of the envelope as a method of motivic development.

```
;; simple pitch quantized envelope
;; with randomized rhythmic performance
(define melody
   (lambda (env pos)
      (play-note (now) zeb1
           (pc:quantize (floor (env (fmod pos 8)))
                        '(0 2 4 5 7 9 11))
           80 3000)
      (callback (+ (now) (random '(7500 5000)))
                'melody env
                (+ pos (random '(.25 .5))))))

(melody (make-envelope (vector 0 60 3 72 5 79 8 60))
        0)
```

Often aa-cell use pitch envelopes in conjunction with a gaussian-random whose mean follows the envelope and whose standard-deviation is either fixed or changed over time by an auxiliary envelope, with the final result often quantised to a pitch class set.

Some of our other common applications of polynomial functions include using two curves for tendency masks, specifying upper and lower boundary conditions [1] and for controlling synthesis parameters.

3.3. Periodic Functions & Modular Arithmetic

Another family of functions common to musicians are the periodic functions. We exploit periodic functions as a means for generating structure in all aspects of musical form - pitched, rhythmic, structural and timbrel. These functions have a range of applications for the live coder, including metric pulsation and pitch cell extraction as well as more common usages such as low frequency oscillation for timbrel (synthesis) variation. Like polynomial functions, periodic functions can provide subtle contours through to grandiose gestures and when combined with various probabilistic tricks can produce engaging performance results.

One aspect of periodic structure that aa-cell regularly exploit is the "composers pulse" [11]. By mapping a cosine function to amplitude it is possible to provide a metric pulse to a regular pattern. Multiple levels of metric information can be easily provided by mapping multiple periodic functions simultaneously. Further, Clynes [11] showed that these same functions can be applied to other rhythmic elements such as note duration and tempo. In fact our experience suggests that it is only through interaction at multiple musical dimensions that an engaging musical performance is obtained.

One common aa-cell application is to use a cosine function on note amplitude to provide a metric pulse to a regular pattern, such as a hi-hat part. Modifying the period of the oscillation probabilistically produces subtle yet engaging syncopation in the rhythmic pattern.

Another, pitch based trick, is to use an oscillator for selecting drum samples, occasionally changing the oscillation rate in order to modify the drum pitch pattern, while retaining a constant rhythmic pattern. As with most of aa-cell's live coding techniques the interest in these simple structures comes from a combination of constant localized change and larger scale regularity. The example below uses an oscillator to choose drum samples and amplitudes.

```
;; a trivial drum machine
(define drum-machine
   (lambda (time p)
      ;; period drum pattern changes each 4 beats
      (play-note (now) kit
                 (+ 50 (* 6 (* cos (* time p pi))))
                 (+ 60 (* 20 (* cos (* time p pi))))
                 2000)
      (case (fmod time 4.0)
            ((0 2.5) ;; kick
             (play-note (now) kit *kick* 80 5000))
            ((1 3) ;; snare
             (play-note (now) kit *snare* 80 5000)))
      (callback (+ (now) 11025) 'drum-machine
                (+ time .25)
                (if (= 0 (fmod time 4.0))
                    (random '(2 3 4 5))
                    p))))
```

Figure 2. *aa-cell* in performance.

Modular arithmetic is another tool which aa-cell use to control the regular cycles common to musical structure. The example above demonstrates the use of modulus for locating the beat position within a 4/4 bar.

3.4. Set Theory (Pitch Class Sets)

Set theory has become a standard tool in 20th Century music composition [29]. aa-cell make extensive use of set theory for manipulating pitch space (tonal or otherwise) [22]. Pitch Class Sets (PCS) provide a simple yet powerful tool for manipulating musical patterns.

By applying common musical devices such as inversion, expansion/contraction, retrograde and transposition to a musical cell, or motif, and then filtering the output though a PCS quantisation process it is possible to rapidly develop interesting musical sequences with high level structural control. The simple example below outlines the probabilistic, two octave transposition of a musical *cell* within the bounds of *pcs*.

```
(define *cell* '(60 62 63)s)
(define *pcs* '(0 3 7 8 10))

;; random two octave transposition of *cell*
;; with random choice of mutation to *cell*
(pc:transpose
    (random -7 7)
    (random (list *cell*
                  (invert *cell*)
                  (retrograde *cell*)
                  (expand/contract *cell*
                                   (* 5 (random)))))
    *pcs*)
```

3.5. Recursion & Iteration

Recursion and iteration provide many opportunities for repetition, evolution, pattern programming and grammars, and they are fundamental to notions of computational time.

Like almost anything we can conceive of, it is possible to think of music as a collection of processes arranged in some form of hierarchical structure that unfolds through time. To a large extent it is the arrangement of these processes that defines the organisation of sound that we describe as music. As Wallace Berry notes, "Musical structure may be said to be the punctuated shaping of time and space into lines of growth, decline and stasis hierarchically ordered." [2 pp. 5].

Our live coding makes extensive use of Impromptu's ability to precisely schedule closures (a function and its environment) for future invocation. The ability to schedule functions self referentially was first discussed by D. Collinge in reference to his Moxie system [12]. Using this mechanism functions[1] may continuously re-schedule their own invocation. The ability for a function to call itself self referentially is the basis of recursion. Impromptu supports scheduled recursions which move through time at a governed rate, we refer to these as "temporal recursions." Impromptu uses the `callback`[2] function to provide this functionality.

The asynchronous nature of Impromptu's temporal recursion model results in a natural cooperative multi-tasking whereby multiple temporally recursive processes operate in a quasi-concurrent manner. Musically, this means that we can have numerous independent musical lines running in parallel. By retaining their arguments between invocations, temporal recursions can maintain their state. This provides an intuitive and encapsulated mechanism for maintaining change over time.

```
;; play a one octave chromatic scale
(define scale
   (lambda (pitch)
      (play-note (now) synth pitch 80 8000)
      (if (< pitch 72)
          (callback (+ (now) 10000) 'scale
                    (+ pitch 1)))))

;; start scale on middle C
(scale 60)
```

[1] Impromptu also allows continuations to be scheduled providing functionality similar to a co-routine.

[2] Impromptu's callback is similar to Moxie's cause function

We make extensive use of temporal recursion in our practice, providing us with three primary advantages.

3.5.1. On-the-fly modification of code

The first advantage is the ability to redefine a temporally recursive process on-the-fly. Because the scheduler will always invoke the most recent definition of any given function, aa-cell can modify the behavior of a temporal recursion by simply redefining the behavior of it's target function. This is a simple and intuitive behavior inherent in Impromptu's temporal recursion model and allows real-time programmers to build, extend and modify code on-the-fly.

3.5.2. Constantly adjustable control rate

A second major advantage of temporal recursion is the ability to change the rate of recursion. When a function schedules itself for future invocation it specifies a delay time. This delay time can be adjusted at any stage, either through random variation, or some other deterministic process and, through this change, modify the rate of recursion. The most trivial application of this ability is to playback a series notes where arguments to the function provide new pitch and duration information. The pitch and duration are used to render a note, and then the duration is again used to specify the time of the next invocation of the function. Suitably modified arguments are retained for the next invocation.

```
;; a chromatic one octave random walk
;; crotchet and quaver rhythm in samples
(define melody
   (lambda (pitch duration)
      (play-note (now) piano pitch 60 duration)
      (callback (+ (now) duration)
                'melody
                (range-limit (+ pitch (random -1 2))
                             60 72)
                (random '(22050 44100)))))

;; start the temporal recursion
(melody 60 44100)
```

What makes this such a valuable technique is its inherent just-in-time behavior, vital in live coding practice as it defers computation and facilitates complex interplay between concurrent processes.

3.5.3.Temporal graphs

The third major advantage of temporal recursion is the ability to modify a temporal recursion's target closure on-the-fly. This is a powerful technique that aa-cell use to control higher level structure in live performance. By altering the "course" of a temporal recursion—by modifying the target function of the callback—it is possible to change a processes direction in a trivial manner. At it's simplest one can think of an example whereby two functions have an equal chance of calling themselves, or their opposite, setting up a temporal recursion which moves, with a probabilistic weighting between two functions. There is, however, no reason to limit the available paths to only two choices and more sophisticated decision mechanisms can be used. This is somewhat analogous to timed Petri Net's and can be used to implement Markov processes, augmented transition networks or other graph-like structures where functions operate as nodes with transitions to arcs defined by callback functions.

```
;; func-a always repeats 10 times
;; then calls back to func-b
(define func-a
   (lambda (cnt)
      (print "in func a" cnt)
      (callback (+ (now) 1000)
                (if (> cnt 9) 'func-b 'func-a)
                (if (> cnt 9) 0 (+ cnt 1)))))

;; func-b has a 50/50 chance
;; of calling into func-a or func-b
(define func-b
   (lambda (cnt)
      (print "in func b" cnt)
      (callback (+ (now) 1000)
                (random '(func-a func-b))
                cnt)))

;; start temporal recursion
;; note that we can call func-a
;; multiple times to start
;; multiple concurrent recursions
(func-a 0)
```

The code example above demonstrates a simple temporally recursive transition network. The network contains two nodes, the functions part-a and part-b, each maintaining it's transition conditions specified within the callback function. In this example, func-a is called and must recall itself ten times before passing control (calling) to part-b, which then has a 50/50 chance of calling itself or calling back into part-a. This simple technique provides a useful mechanism for building higher level musical structure on-the-fly.

4. PERFORMANCE PRACTICES

In addition to the computational approaches inherent in aa-cell's practice, there are a number of performance considerations that do not impact so much on the musical result but on the effectiveness and presentation of performances.

4.1. Code Expansion

One problem that all live programmers must deal with is how to physically type the required code within a reasonable period of time; reasonable for both the audience but, probably more importantly, to assist the performer to more rapidly realise ideas.

In addition to the parsimonious and efficient algorithmic descriptions discussed in section 3, an obvious way to deal with this issue is to abstract away as much detail as possible into pre-built functions and libraries. This "preparation" is an important aspect of live-coding and aa-cell regularly spend time working on library code. However, the downside with this approach is that abstracting away the ideas restricts our ability to change, modify and re-evaluate the code during the performance. We spend a good deal of any given performance modifying and extending code structures, in fact a performance may well be based around the constant manipulation of a single temporal recursion.

Given that code is our medium, and that abstracting away code reduces our range of expression we have been using code expansion as a complimentary

technique to functional abstraction. Code expansion allows aa-cell to program the essential elements of an expression and abstract away the remaining details to a code template. The template generates code based on the arguments supplied to the template. Once the generated code is pasted into the environment we can interact with it as normal; executing, extending and running it as we would any other code. Our code expansions cover a basic set of regular usage patterns including melodic, chordal and rhythmic structures. Code expansion provides aa-cell with an efficient mechanism for customising common usage patterns and has proven itself to be a huge benefit in performance as it allows us to concentrate on the essential details without having to worry about writing boiler plate code.

4.2. Non-programmable control

While text programming languages can be an effective medium for expressing processes and structures, their textual nature makes them less capable of handling rapid change. The issue of rapid change in live programming is of continuing concern for live programmers. As Fredrik Olofsson states, "I feel I'd have to rehearse a lot more to be able to do abrupt form changes" [24]. In our experience the problem may be more intractable than this. While we have been able to find methods, such as code expansion and functional abstraction, for adequately reducing the time spent defining higher level formal structures, there is a severe physical limit to the amount of immediate change available in a text based environment. To be clear, we are not suggesting that immediate change cannot be "programmed", we do this regularly, but as live programmers our ability to respond immediately is limited by the time required to make and evaluate source code changes.

4.2.1.Editor tools

We ameliorate this situation, in part, by providing functionality in the text editing environment. For example, Impromptu allows programmers to set up to ten mark points. These marks can be set, moved to and have their underlying expressions evaluated with key bindings. This provides for rapid movement around the text editing environment and allows programmers to evaluate text that maybe located outside of the current viewable text region or even in another text buffer.

4.2.2.External controllers

Text editing features only go so far in providing real-time programmers with the ability to affect immediate change. To augment, or provide better gestural control, we can also employ external control surfaces with various dials, faders, buttons and so on. Impromptu can communicate to these via MIDI or OSC as required. We have developed code libraries to facilitate the direct assignment of external controllers to pre-bound global symbols. In practice this allows aa-cell to trivially assign controller values to arbitrary code, providing real-time gestural control at any point in our code.

```
;; play a series of notes with random pitch
;; bounded by the current value of
;; controller 1 and controller 2
(define melody
   (lambda ()
      (play-note (now) piano
                 (random ctrl1 ctrl2)
                 60 22050)
      (callback (+ (now) 22050) 'melody)))
```

In particular we have found that control surfaces with motorised controllers allow two way communication between musician and computer. Manual control can update assigned values, and programatic variations can be reflected in automated movement of the motorised controllers.

It is worth emphasising that the interactive, real-time influence exerted on the generative algorithms at play is, we believe, central to our live coding practice.

4.3. Colaboration & Communication

Like any performance practice, live coding requires coordination between the performers and consideration of how the audience relates to the performance practice.

4.3.1. Collaboration

The major consideration regarding collaboration between performers in aa-cell relates to synchronisation, timing and data sharing. At a global level this often includes tempo, meter, harmonic progressions and other structural features. Impromptu supports a number of mechanisms for communication between remote hosts.

In recent years Open Sound Control (OSC) has become a de-facto standard for musical communication and provides a convenient mechanism for communication between Impromptu hosts and a variety of other computer music tools supporting the OSC protocol.

Impromptu also offers an Inter Process Communication (IPC) mechanism for communication directly between remote processes. This provides a powerful mechanism for sharing and executing code across remote Impromptu hosts. In practice this allows aa-cell to share functions and global variables during a performance. More specifically the IPC mechanism allows us to define functions and variables in each other's Scheme interpreter. Unfortunately this powerful feature is limited without the ability to view and edit the associated source code.

In order to resolve this issue, aa-cell anticipate the future addition of a collaborative text editing environment to the Impromptu IDE[3]. This would provide live programmers with the ability to work collaborative on a single source code file simultaneously. We anticipate this to be a significant addition that would allow live programmers to work together in a more substantially collaborative environment. However, that said, aa-cell also enjoy exploring the separation of environments.

Impromptu's precise timing and interactive environment encourages users to "perform." By this, we mean that aa-cell often work without the safety net of networked time codes, shared harmonic structures etc. and instead concentrate on performing our individual

3 For a good example of collaborative text editing see the SubEthaEdit text editor http://www.codingmonkeys.de/subethaedit/

environments by manually timing the execution of code, manually adjusting tempo and metre, and interacting in a constant dialogue of harmonic and timbrel change within a vocabulary of shared musical queues. No doubt our continued exploration of live coding practice will continue to oscillate between fixed and freer forms of coordination.

4.3.2. Audience Communication

In order to enhance the audience appreciation of aa-cell performances we have adopted a number of performance conventions. As has become standard in live coding practice we project the computer displays so that the audience can see the code being typed. We try to make the code as legible as possible.

However, even with a strongly technical audience, complete comprehension of the generative ramifications of the source code being run during a performance is challenging. Knowledge of the programming language, the environment, the algorithms used, musical, sonic and more general cultural knowledge, even knowledge of individual performers practice are all necessary for a complete mapping of the projected code to the musical outcome. Consequently, most audiences will struggle to fully realise the connections between the code and any generated material.

Critics of live coding have suggested that this makes the projection of the source code an unnecessary and intrusive endeavor. Our experience is that despite people's inability to understand the detail of the code, they appreciate that the work is being built up as it proceeds and seem to enjoy participating in identifying symbolic queues. To this end we make an effort to use function and variable names that people will recognise and that may assist in their interpretation of the code. Symbol names such as "outrageous-kick" and "grunge-it-up" never fail to communicate our intent! Regardless of the audiences' level of understanding, code projection highlights to the audience that structures are coded during the performance. This is particularly important at this early stage in the development of live coding as a musical practice.

To further enhance the appreciation that live coding builds structure on the fly we always start our performances with a blank text editor. We feel that the growing complexity of the typed code paralleled by the increasing complexity of the sonic outcome helps to articulate the building process to the audience. In addition, the Impromptu environment has text hi-lighting features that not only assist the programmer, but the audience to see where the programmers attention (cursor) is positioned and to indicate when functions are evaluated.

5. CONCLUSION

In this paper we have outlined the theoretical and practical aspects that are significant in our live coding practice as aa-cell. The paper has focused particularly on processes related to musical structure and event level considerations and discusses some of the techniques that aa-cell have found useful for defining musical processes in a live coding performance.

We are certainly not the first people to discuss the usefulness of simple mathematical functions for modeling musical behaviors and it is perhaps unsurprising that aa-cell would find these functions useful in our practice. What has been a surprise to us, however, is just how much utility a small set of processes have provided. It is possible that our success with these simple functions is due to our manual control of higher level structure through our manipulation of running process, but it may also point to a more profound revelation about parsimonious computational representations of music and improvisational performance. We hope to expand more on these ideas in the future.

Many of the performance aspects of live coding practice are still being hotly debated and aa-cell are still actively considering the pro's and con's of various performance related issues. However, we feel confident that as the practice matures, and audiences become more familiar with this new practice, these issues will resolve themselves. In the mean time we are busying ourselves with music making.

This article has provided several brief, and necessarily trivial code examples. We have provided a web page http://impromptu.moso.com.au/icmc-examples.html with supplementary material relating to this paper, including expanded code examples and related audio recordings. We hope this will provide more compelling documentation for the interested reader. Screen casts of complete live coding performances can also be found on the Impromptu website [28].

6. REFERENCES

[1] Berg, P. 1996. Abstracting the Future: The search for musical constructs. *Computer Music Journal* 20(3): 24-27.

[2] Berry, W. 1976 "Structural Functions in Music" General Publishing Company Limited Toronto,Ontario. 1987 Publishing by Dover Mineola, NY.

[3] Blackwell, A. and Collins, N. 2005. The Programming Language as a Musical Instrument. In P. Romero, J. Good, E. Acosta Chaparro and S. Bryant (eds.) *Proceedings of the 17th Workshop of the Psychology of Programming Interest Group*. Sussex University, pp. 120-130.

[4] Brown, A. R., Towsey, M., Wright, S. and Diederich, J. 2001. Statistical analysis of the features of diatonic music using jMusic software. *Proceedings of the Computing Arts 2001: Digital Resources for Research in the Humanities*. Sydney, pp.

[5] Brown, A. R. 2002. Opportunities for Evolutionary Music Composition. In P. Doornbusch (ed.) *Proceedings of the Australasian Computer Music Conference*. Melbourne: ACMA, pp. 27-34.

[6] Brown, A. R. 2004. Playing with Dynamic Sound. In E. Edmonds and R. Gibson (eds.) *Proceedings of the Interaction: System, Practice, Theory*. Sydney: Creativity and Cognition Studios Press, pp. 435-450.

[7] Brown, A. R. 2004. An aesthetic comparison of rule-based and genetic algorithms for generating melodies. *Organised Sound* 9(2): 191-198.

[8] Brown, A. R. 2005. Exploring Rhythmic Automata. In F. e. a. Rothlauf (ed.) *Proceedings of the Applications of Evolutionary Computing: EvoWorkshops 2005*. Lausanne, Switzerland: Springer, pp. 551-556.

[9] Brown, A. R. 2003. Music Composition and the Computer: An examination of the work practices of five experienced composers. PhD dissertation. Brisbane: The University of Queensland.

[10] Brown, A. R. 2006. Code Jamming. *M/C Journal* 9(6). http://journal.media-culture.org.au/0612/03-brown.php

[11] Clynes, M. 1984 The secret life of music. *In Proceedings of the 1984 International Computer Music Conference*. San Francisco: ICMA

[12] Collinge, D. J. 1984 MOXIE: A Language for Computer Music Performance. *In Proceedings of the 1984 International Computer Music Conference.* San Francisco: Computer Music Association

[13] Collins, N., McLean, A., Rohrhuber, J. and Ward, A. 2003. Live Coding in Laptop Performance. *Organised Sound* 8(3): 321-330.

[14] Collins, N. 2003. Generative Music and Laptop Performance. *Contemporary Music Review.* 22 (4): 67-79.

[15] Collins, N. and Olofsson, F. 2006. klipp av: Live Algorithmic Splicing and Audiovisual Event Capture. *Computer Music Journal*. 30(2): 8-18.

[16] Cope, D. 2000. *The Algorithmic Composer*. Madison: A-R Editions.

[17] Dean, R. 2003. *Hyperimprovisation: Computer-Interactive Sound Improvisation*. Middleton: A-R Editions.

[18] Desain, P. & Honing, H. 1993. Tempo curves considered harmful*. *In Time in contemporary musical thought* J.D.Kramer(ed.), Contemporary Music Review. 7(2). 123-138. Pre-printed in: Desain, P. & Honing, H. (1992). Music, Mind and Machine. Studies in Computer Music, Music Cognition and Artificial Intelligence.Amsterdam: Thesis Publishers.

[19] Dodge, C. and Jerse, T. A. 1997. *Computer Music*. New York: Schirmer Books.

[20] Hiller, L. and Isaacson, L. 1992. Musical composition with a high-speed digital computer. In *Machine Models of Music*, S. M. Schwanauer and D. A. Levitt, Eds. MIT Press, Cambridge, MA, 9-21.

[21] Huron, D. 2006. *Sweet Anticipation: Music and the Psychology of Expectation.* Cambridge MA: MIT Press.

[22] Lerdahl, F. 2001. *Tonal Pitch Space*. NY, NY: Oxford University Press.

[23] Meyer, L. B. 1956. *Emotion and Meaning in Music*: Chicago: University of Chicago Press.

[24] Nilson, Click. 2007 Live Coding Practice. *Proceedings of NIME New York.*

[25] Rowe, R. 1993. *Interactive Music Systems: Machine listening and composing*. Cambridge, MA: MIT Press.

[26] Sorensen, A. and Brown, A. R. 2000. Introducing jMusic. In A. R. Brown and R. Wilding (eds.) *Proceedings of the InterFACES: The Australasian Computer Music Conference*. Brisbane: ACMA, pp. 68-76.

[27] Sorensen, A. 2005. Impromptu: an interactive programming environment for composition and performance. In *Proceedings of the Australiasian Computer Music Conference,* 2005.

[28] Sorensen, A. The Impromptu programming environment: http://impromptu.moso.com.au

[29] Straus, J. N. 1989. Introduction to Post Tonal Theory. Prentice Hall College Div.

[30] Taube, H. 2004. *Notes from the Matalevel: Introduction to Algorithmic Music Composition*. London: Taylor & Francis.

[31] Temperley, D. 2007. *Music and Probability:* Cambridge, MA: MIT Press.

[32] Winkler, T. 1998. *Composing Interactive Music*. Cambridge, MA: MIT Press.

[33] Wang, G. and Cook, P. R. 2003. ChucK: A Concurrent, On-the-fly, Audio Programming Language. *Proceedings of the International Computer Music Conference*. ICMA, pp. 219-226.

[34] Xenakis, I. 1992. *Formalized Music: Thought and Mathematics in Music*. Stuyvesant NY: Pendragon Press.