Impromptu: An interactive programming environment for composition and performance.

Andrew Sorensen
MOSO Corporation
15/925 Brunswick St
New Farm, 5004
Australia
andrew@moso.com.au

Abstract

Interactive development environments are making a resurgence. The traditional batch style of programming, edit -> compile -> run, is slowly being reevaluated by the development community at large. Languages such as Perl, Python and Ruby are at the heart of a new programming culture commonly described as extreme, agile or dynamic. Musicians are also beginning to embrace these environments and to investigate the opportunity to use dynamic programming tools in live performance. This paper provides an introduction to Impromptu, a new interactive development environment for musicians and sound artists.

Introduction

Impromptu is a new programming environment being developed to explore the manipulation of musical structure in live performance. Live, real-time or on-the-fly programming is a performance paradigm stemming from laptop performance, but with an emphasis on the crafting of algorithms in real-time. Impromptu is designed to provide a dynamic, real-time, interpreted, multi-user runtime capable of supporting the creation, modification, distribution and evaluation of source code in live performance. Impromptu consists of a synthesis/DSP engine, a real-time scheduling engine, a Scheme interpreter and an Integrated Development Environment.

Live Programming

> The use of computers in the creative process mandates that we think of communicative and creative processes in terms of abstract structures and the manipulation of such structures. (Holtzman 1994)

Live Programming is a performance practice that emphasizes the creation and manipulation of algorithms in real-time. Live Programming emphasizes the cerebral nature of algorithmic development and mandates that this process should be transparent and accessible. "Give us access to the performer's mind, to the whole human instrument." (www.toplap.org)

Live Programming offers a novel opportunity for audience members to participate in the creation of a work as it unfolds; an exciting medium allowing audience members to intellectually grasp the structures and abstractions of a work even before it has been rendered sonically. And perhaps even more important is the opportunity for performers to connect with each other, freely exchanging ideas of both micro and macro structure. Additionally, there is a performance practice, a level of virtuosity evident in Live Programming that will hopefully provide a stage presence that has often been missing in live laptop performance. Above all, the author hopes that this interactive medium for music making will provide an environment for the novel exploration of structure through abstraction and combination, the building blocks of digital art.

A Brief History

The first documented Live Programming performance was at a concert performed by Ron Kuivila at STEIM in 1985. Ron used his Formula programming language (A language based on Forth) for this half hour performance. Forth and Lisp systems were used for various real-time performances during the late 80s and early 90s. The League of Automatic Composers (later becoming the Hub) were a performance group that made use of real-time programming methods in live performance, encouraging audience members to walk between performers in order to see the code being created and executed during the performance. (Gresham 1998).

With the exception of small pockets of activity live coding seems to have been largely absent during the 90s. Live Programming during this period appears to have been restricted to the runtime manipulation of signal paths in Max patches.

A resurgence in the use of text based languages for Live Programming can be dated to 2000 and Julian Rohrhuber's real-time Supercollider experiments. Other important work at this time was Alex McCleans work with SLUB and Adrian Ward's REALbasic environment "Pure Events" (Collins et al 2003).

Recently Live Programming has been given a broader treatment due to the attentions of computer music luminary Perry Cook. Ge Wang and Perry Cook have developed a new programming language specifically tailored for on-the-fly creation of DSP algorithms. The language ChucK provides novel concepts such as the ability to embed time primitives directly into the program flow (Wang and Cook 2003).

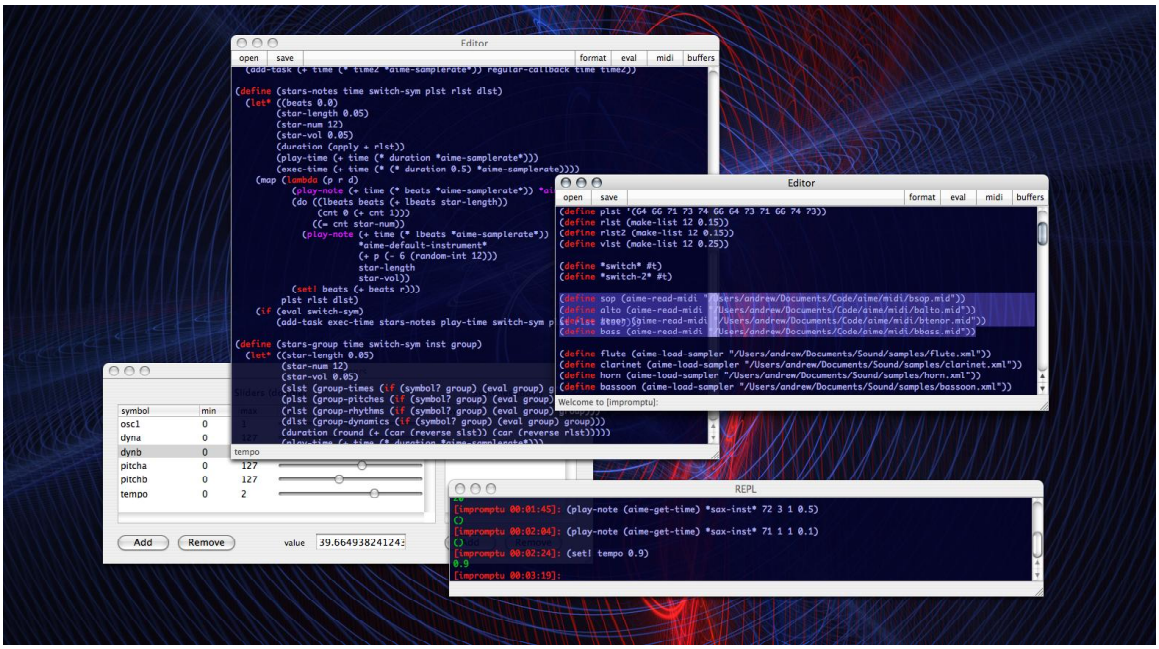

Figure 18: The Impromptu Integrated Development Environment.

Supercollider, having always been at the forefront of this field is playing an ever-increasing role in Live Programming. The success of the now open source SC3 server has seen a proliferation of client architectures making use of the successful OSC communication protocol. There are SC3 clients available for many languages including Java, Scheme, Ruby and the more traditional SCLang. Many of these environments support Live Programming.

Impromptu

Impromptu is an interactive development environment being built by the author. Impromptu has been designed to provide an interactive environment for the exploration of real-time musical algorithm crafting and performance.

Impromptu is designed to support multiple users operating within a shared runtime environment. Impromptu also includes an Integrated Development Environment (IDE) designed for both live and non-live programming.

Impromptu emerged at the juncture of three separate paths: (1) an interest in dynamic languages and interactive development environments (2) a desire to explore programming as a live performance tool and (3) the development of AiME, a new C++ synthesis/scheduling engine. The result of this merger is the close coupling of a high-level language runtime, Scheme, and a synthesis/scheduling engine, AiME.

AiME

AiME has been developed over the past 12 months by the author with support from the Australian CRC for Interaction Design (ACID) dynamic content project. The engine has been designed to provide a vehicle for exploring dynamic content creation and delivery. The AiME engine is based around a simple, flexible real-time scheduling engine providing sample accurate callbacks to C++ methods. AiME does not implement a control-rate, instead allowing scheduled callbacks to effect signal processing units at any frame boundary. Although this approach has inherent performance penalties the relative simplicity of the architecture and an ever-increasing supply of gigahertz make this a musically and architecturally satisfying solution.

Inter-object communication is made through the central scheduling engine promoting a high degree of temporal control across all aspects of the engine. This convention applies equally to both synthesis/DSP units and event/control structures.

AiME includes a growing array of synthesis instruments including a flexible STK wrapper that allows STK generators, filters, effects and instruments to be chained at runtime (Cook & Scavone 1999). AiME provides extensive parametric control of all sound sources through the scheduler.

The AiME library has been designed to integrate directly into 3rd party applications. AiME's primary design consideration was non-linear environments, and more particularly computer game engines. This design goal influenced the expectoration that a scripting language or other real-time mechanism would be used to control AiME. This design made it an obvious candidate for coupling with a high-level language such as Scheme.

AiME is a BSD licensed project hosted at http://sourceforge.net/projects/aime-lib.

Scheme

Scheme is a dialect of Lisp, a dynamic language designed by John McCarthy in 1958. Scheme is a small, elegant yet powerful language developed by Gerald Sussman and Guy Lewis Steel at MIT in the late 1970's (Abelson & Sussman 1996).

Impromptu uses a modified scheme interpreter based on TinyScheme (Souflis). The interpreter is written in C and is integrated directly into the AiME engine. Communication between the Scheme runtime and the AiME engine happens through a combination of Scheme/C bindings and Scheme opcodes. These integration points provide Scheme with "native" access to the AiME scheduling engine, and therefore to any C++ method available to the scheduling engine. In effect this gives the scheme engine complete access to AiME C++ objects at runtime. This integration is transparent to Impromptu programmers who should be unaware of any distinction between scheme -> scheme calls and scheme -> C++ calls.

The Impromptu scheme interpreter currently supports most of the scheme RSR5 standard (Abelson, H. etal 1998) including macro support. While RSR5 only includes a tiny standard library there is vast array of public domain Scheme code readily available for use in Impromptu. Most of this RSR5 compatible code should evaluate in Impromptu with little or no modification.

Function Scheduling

One of Scheme's most powerful language features (now common in many dynamic languages) is its support for first class functions. First class functions can be sent as arguments to functions and returned from functions but more importantly, support for first class functions means that functions can be created at runtime.

Impromptu enhances this feature by allowing programmers to schedule function evaluation at runtime. The ability to create new functions at runtime and to precisely schedule their execution gives programmer-performers a powerful environment for exploring musical abstractions. To demonstrate this in practice a common technique utilising scheduled Scheme function callbacks is to write quasi-recursive functions that constantly resubmit themselves to the scheduling engine with a time increment.

```
(define (play)
   ; play C4 every 4 seconds
   (play-note (now) *inst* 60)
   (callback (time++ 4.0) 'play))
```

All time synchronisation in Impromptu is handled on the server. In fact there is no client/server synchronization in Impromptu. Impromptu clients do not execute any code, instead passing all expressions to the server for evaluation. This conveniently removes the jitter problems commonly associated with other Live Programming environments. Execution time however, must still be considered when working with Impromptu's Scheme callback architecture. This is generally not a major issue however as it should be relatively intuitive to the real-time programmer that code evaluation is not instantaneous and it is a relatively trivial matter to circumvent the problem by scheduling callbacks ahead of time.

```
(define (long-function T)
   ; schedule callback 4100 frames
   ; ahead of desired sounding time.
   (generate-and-play-100-notes T)
   (callback (+ T 40000)
             'long-function
             (+ T 44100))
```

This code shows an example of calling back ahead of time. The example sets the callback 4100 frames ahead of the desired sounding time. This gives the interpreter 4100 frames in which to evaluate the function (generate-and-play-100-notes <scheduled time>) that internally schedules all 100 notes for evaluation at <scheduled time>. In reality many operations do not need to worry about timing issues this accurately and are free to call Impromptu's `(now)` function, which returns the audio engines current frame. It is important though to remember that precise scheduling is possible if required.

Multi-User Runtime

Another of Scheme's powerful features is the ability to create and bind new symbols at runtime. Symbols are the names given to abstractions. They are the mappings that allow the production of combinations of ever-greater complexity. Scheme allows users to rebind existing symbols to new objects (combinations) at runtime. The ability to remap objects at runtime is an extremely powerful technique.

The simple example below demonstrates a powerful Live Programming technique, the ability to rapidly change mappings at runtime. In this example our looping melody will instantly change as soon as we evaluate the final line.

```
;start by defining a list of pitches
(define melody '(64 62 60 62))

;next make a quasi-recursive callback
;to loop the melody
(define (loop-melody)
   (play-notes (now) *inst* melody)
   (callback (time++ 4.0)
             'loop-melody))

;start the quasi-recursive function
(loop-melody)

;rebind melody to something new
(define melody '(64 64 64))
```

Impromptu significantly enhances this feature by allowing multiple users to share a single networked runtime environment providing any number of collabora-

tors with simultaneous access to a shared symbol pool. The ability for multiple users to access a shared runtime offers them the opportunity to create, modify and extend the ideas of collaborators in a highly interactive environment enhancing both creativity and fun. And users need not be human.

A common operation using symbol binding and a shared runtime is for one user to replace another users running code. A simple example will help to identify why these two features combine to form a compelling new musical performance tool. Jack binds a function that plays a major scale to the symbol scale and sets a callback to itself +1 seconds from now.

```
(define (scale)
  ; Jacks major function
  (play-scale (now) 60 72 'major)
  (callback (time++ 1.0) 'scale))
```

Jack then needs to call his newly bound function in order to begin it's execution. The result is a major scale that begins playing every second.

```
(scale)
```

Jill, who has a connection to the same Impromptu runtime as Jack decides to replace Jack's major scale with a minor scale. Jill now binds a new function to the symbol scale (the same symbol that Jack used) which plays a minor scale and sets a callback to itself +2 seconds from now.

```
(define (scale)
   ; Jills minor function
   (play-scale (now) 60 72 'minor)
   (callback (time++ 2.0) 'scale))
```

The next time the scale function is called from the scheduler Jill's minor scale implementation will be called having replaced Jack's implementation. Notice that Jill never explicitly calls (scale). She does not need to as the scheduling engine is already scheduled to evaluate a function bound to the symbol scale, regardless of what that function might be. The transfer between Jack's function and Jill's function is seamless. Jack is of course perfectly at liberty to change scale back again if he does not agree with Jill's aesthetic choice.

The ability to rebind symbols and the ability to create first class functions are two key Live Programming concepts. These runtime facilities provide real-time programmers with a highly expressive environment for manipulating abstractions and combinations in live performance. The consequences for a multi-performer runtime

Future Work

Impromptu development is currently progressing at a rapid pace. Areas of current development include, an IDE for OSX, additional instrument and ugen development and a growing library of Scheme music and ALife/AI functions.

Three areas of future development that would greatly enhance the project would be a Scheme compiler, for more efficient run-time performance, an integrated threading mechanism for the Scheme interpreter and the addition of intelligent agents.

Intelligent agents offer the opportunity for human users to collaborate with artificial agents in a collaborative partnership. Genetic programming in particular provides interesting and novel opportunities for runtime creation and manipulation of code. In the future Jack and Jill's functions may produce whole new offspring, the combination of their genetic makeup. Intelligent codebots modifying and extending code at runtime is an interesting area of research not just for artistic practice but also for programming at large.

One of the author's primary interests is structure, particularly the ways in which computer programs can be used to provide new insights or techniques into its composition and decomposition. Impromptu was developed primarily to supply the author with a tool that could be used to more readily explore these structures in real-time. Future work will continue to expand Impromptu in directions that help to shed further light on the abstractions and combinations that convolve to produce interesting art.

Conclusion

Live Programming provides an exciting and novel departure from traditional musical performance. The ability to improvise both micro and macro abstractions and to manipulate combinations of any kind provides Live Programmers with an opportunity to explore new avenues of musical collaboration and improvisation. Audiences will benefit not only from new sonic experiences but also through a new level of intellectual involvement that is made possible by the explicit representation of ideas described in source code.

However, Live Programming is in its infancy and there is a much work to be done. Like all great performance art, virtuosic Live Programming will be an immensely difficult skill to attain, requiring time, dedication and talent. I hope that this new performance practice has the ability to survive and flourish to the point where it sees it's first true virtuoso.

References

Abelson, H., Sussman, G., Sussman, J. 1996. *Structure and Interpretation of Computer Programs* Cambridge, MA: MIT Press

Abelson, H etal, 1998 "Revised Report on the Algorithmic Language Scheme"

Collins, N., McLean, A., Rohrhuber, J., Ward, A. 2003 "Live coding in laptop performance", *Organised Sound* 8(3), 321-330

Cook, P., Wang, G, 2003 "ChucK: A Concurrent, On-the-fly , Audio Programming Language", *In proceedins of the 2003 International Computer Music Conference*.

Cook, P., Wang, G. 2004 "On-the-fly Programming: Using Code as an Expressive Musical Instrument", *In Proceedings of the 2004 International Conference on New Interfaces for Musical Expression*.

Gresham, L. 1998 "The Aesthetics and History of the Hub: The Effects of Changing Technology on Network Computer Music"; *Leonardo Music Journal* Vol8, 39-44

Holtzman, S. 1994. *Digital Mantras: The Lanuages of Abstract and Virtual Worlds*, MA: MIT Press

McCartney, J. 2002. "Rethinking the Computer Music Language: SuperCollider", *Computer Music Journal* 26:4, 61-68

Mclean, A. 2004. "Hacking Perl in Nightclubs" http://www.perl.com/pub/a/2004/08/31/livecode.html

Norvig, P. 1992. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp,* San Francisco CA: Morgan Kaufmann.

Souflis, D. "TinyScheme" http://tinyscheme.sourceforge.net/

TOPLAP http://www.toplap.org

Wang, G and Cook, P. 2003. "ChucK: A concurrent, on-the-fly audio programming language." Proceddings of the International Computer Music Conference, Singapore.