LuaAV: Extensibility and Heterogeneity for Audiovisual Computing

Graham WAKEFIELD and **Wesley SMITH** and **Charles ROBERTS**
Media Arts and Technology, University of California Santa Barbara
Santa Barbara, CA 93110,
USA,
{wakefield, whsmith, c.roberts}@mat.ucsb.edu

Abstract

We describe LuaAV, a runtime library and application which extends the Lua programming language to support computational composition of temporal, sound, visual, spatial and other elements. In this paper we document how we have attempted to maintain several core principles of Lua itself - extensibility, meta-mechanisms, efficiency, portability - while providing the flexibility and temporal accuracy demanded by interactive audio-visual media. Code generation is noted as a recurrent strategy for increasingly dynamic and extensible environments.

Keywords

Audio-visual, composition, Lua, scripting language

1 LuaAV

LuaAV is an integrated programming environment based upon extensions to the Lua programming language enabling the tight real-time integration of computation, time, sound and space.

LuaAV has grown from the needs of students and researchers in the Media Arts & Technology program at the University of California Santa Barbara; its origins lie in earlier Lua-based audio and visual tools [20] [15] [22]. More recently it has formed a central component of media software infrastructure for the AlloSphere [1] research space (a 3-storey immersive spherical cave-like environment with stereographic projection and spatial audio).

Various projects built using LuaAV have been performed, exhibited or installed internationally, for scientific visualization [1], data visualization [13], immersive generative art [21], game development[1], live-coding (Figure 1) and audiovisual performance[2].

LuaAV is available under a UC Regents license similar in nature to the BSD license[3].

[1] http://www.charlie-roberts.com/projects/circles/
[2] http://www.mat.ucsb.edu/~whsmith/Synecdoche/
[3] http://lua-av.mat.ucsb.edu/

Figure 1: LuaAV in a live-coding performance (Asterisk: performed by A McLeran and G Wakefield, 2009).

2 LuaAV philosophy

The broad attitude taken in the development of LuaAV draws inspiration from the Lua programming language itself: extensibility, meta-mechanisms, efficiency, and portability[4].

2.1 Extensibility

Lua is described as an 'extensible extension language' [8]: a configuration language to embed within and extend the control of kernel application or library code (which is typically written in a statically compiled language such as C). LuaAV follows this methodology directly: the core engine of LuaAV is a library of code (*libluaav*) intended to be embeddable within applications and application plugins, embedding code to manage instances of Lua interpreters, schedulers, an audio driver, and basic communication protocols (MIDI and OSC). Most aspects of the core library have both C and Lua programming interfaces.

[4] The reasoning behind the choice of Lua has been documented in prior publications, particularly [17].

The LuaAV application embeds *libluaav* and adds to it a GUI for managing active script states, a CodePad for adding code to script states at run-time, and a Windowing/Menu system. Most of these components are also scriptable; indeed much of the application logic of LuaAV itself is written using embedded scripts.

2.2 Meta-mechanisms

A fundamental Lua concept is the provision of low-level meta-mechanisms for implementing features, rather than a built-in fixed set. Lua's meta-mechanisms bring an economy of concepts while allowing the semantics to be extended in unconventional ways. We find this open-ended philosophy appropriate to our research domain of computational composition.

We attempt re-use the mechanisms provided by Lua itself in a consistent and predictable manner. As examples: the addition of temporal scheduling in LuaAV is implemented as an extension of the existing coroutine construct in Lua; new functionality is added to the runtime environment using the existing Lua module system; and many of these capabilities are specified using the existing data description and metamethod features of Lua.

2.3 Efficiency

Lua is widely acknowledged as amongst the most efficient of interpreted or scripting languages, however there is still an order of magnitude of performance cost relative to statically compiled code: the price paid for dynamic flexibility that interpreters offer. For this reason, the core scheduler, drivers and other elements of the *libluaav* runtime library are coded in C/C++. Nevertheless, in order to overcome the usual trade-off between flexibility and efficiency, we have begun to leverage of run-time compilation to machine code within LuaAV.

On i386 platforms, the Lua core of libluaav is replaced with the LuaJIT [11] interpreter and trace compiler, allowing performance approximating C for certain algorithms, and significant performance boost over regular Lua even in interpreted mode.

LuaJIT grants efficiency for pure Lua source code, however it cannot optimize over the C/Lua boundary. For those aspects of an application that talk to existing C code, we have been investigating the potential of LLVM/Clang [9]. We have developed a binding to a large proportion of the LLVM/Clang API from within Lua[5], such that abstract syntax trees as Lua data-structures, or pure C code strings, can be JIT-compiled and linked back into the application as it proceeds, under the direction of an executing Lua script.

2.4 Portability

It is near impossible to portably support the complex demands of multimedia applications because of the diversity of platforms and their dependencies. LuaAV currently targets recent Linux and OSX platforms, however we strive to use established, stable cross-platform libraries where possible (such as PortAudio/JACK, the Apache Portable Runtime, etc), or else provide abstraction layers for platform-specific code as appropriate (for example, the LuaAV application Windowing and GUI is written using a common abstraction layer over Qt for Linux and Cocoa for OSX).[6]

3 Related work

LuaAV is one of a family of audio/visual applications in which the primary interface is an embedded programming language, including SuperCollider [10], Impromptu [2], Fluxus [7], ChucK [23], and many more. All of these applications can all be used within a performative context, such as live-coding [3].

Impromptu, based on the Scheme programming language, is an OS X only environment that was originally created for audio manipulation; it has been extended to also include visual programming. One element of Impromptu that is of particular interest is its multi-user runtime; a single networked Impromptu environment can be accessed and manipulated by multiple users concurrently. We are actively developing a similar capability in LuaAV to satisfy multi-user demands in the AlloSphere.

Fluxus is another Scheme based platform, however it is primarily geared towards visual composition and is cross-platform. The Fluxa add-on module adds basic audio synthesis and playback capabilities to the Fluxus environment; however it is limited in terms of the number and breadth of unit generators that it provides.

[5] http://code.google.com/p/luaclang

[6] Nevertheless, the broad scope of the LuaAV application implies many non-trivial dependencies. We currently include a Lua-based build tool and command-line scripts to try to make installation on Linux more fluid.

ChucK is a live coding language geared towards audio with fine scheduling between synthesis and control ('strongly timed'). LuaAV's scheduling system offers similar control, which will be described in detail below. ChucK's visual capabilities only extend to one of its development environments, the Audicle, and primarily revolve around visualizing currently running audio processes.

SuperCollider is also predominantly a system for audio synthesis and scheduling. Its language is strongly inspired by Smalltalk, whose dynamism provides many possibilities for modifying running programs in real time. SuperCollider can be extended with downloadable modules ('quarks'), some including graphical capabilities[7].

4 LuaAV Implementation

4.1 Script states

Opening a script in the LuaAV application creates a *luaav_state* object, a *libluaav* wrapper around a Lua interpreter state with additional components:

- a logical clock
- a scheduler queue with pending events and coroutines
- a graph of audio processes (*Synths*)
- a bidirectional message queue (for communication with the audio graph)
- a memory pool for C-allocated objects associated with the lifetime of the *luaav_state*

An opened script can be closed or reloaded from within the LuaAV application, and its source can be viewed by opening the default external editor. The file modification date of the script file is monitored by LuaAV and the script automatically reloaded if changed; thus users can edit scripts in their preferred editor and see the results updated in LuaAV immediately.

Scripts can also be edited via the integrated CodePad (Figure 2) which was added to incorporate support for live coding practices into LuaAV. Code entered into the CodePad does not reload the script; rather it is injected into the *luaav_state* without interruption. Important features of the CodePad include:

- syntax highlighting via the Leg[8] parsing expression grammar
- the ability to edit multiple scripts concurrently in tabs or multiple windows
- the ability to execute a selected portion of a script
- basic visual error reporting

[7] e.g. http://sourceforge.net/projects/scgraph/

[8] http://leg.luaforge.net/

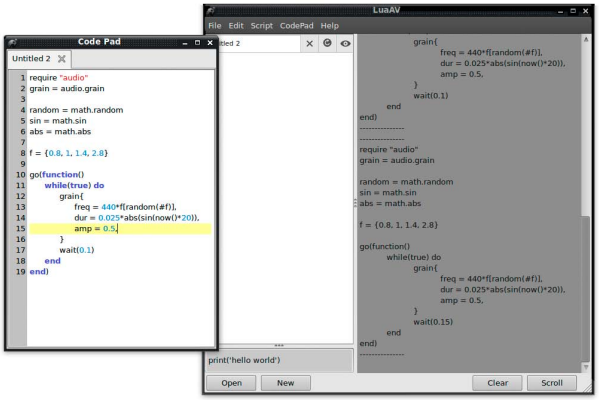

Figure 2: The LuaAV application running on Ubuntu 9.10, with the CodePad view open.

4.2 Scripting in real-time

A central problem of interactive computing applications is the translation from the abstract temporality of programming to the concrete and often unpredictable behavior of real-time behavior and interaction. The articulation of structure goes beyond matters of efficiency to demand:

- capacity to maintain required state until the appropriate moment (dynamic memory management)
- flexibility to re-activate maintained state at unpredictable moments (re-entrancy)
- ability to delay activity until a chosen or appropriate time (event ordering, scheduling)
- ability to specify events in measures of real-time (clocks, event spacing) as well as logical/causal relations (event handling)

Lua already offers excellent re-entrancy and dynamic memory management: user-driven calls and library callbacks can be made into an interpreter instance after a script has executed, and variables will remain alive so long as they are accessible. This re-entrancy extends to the

implementation of coroutines[9] in Lua in which lexically scoped local variables will remain 'alive' for as long as the coroutine block runs or awaits to be resumed.

4.2.1 Metrics, scheduling and concurrency

While Lua ensures deterministic sequencing of instructions, it lacks is a sense of temporal metric. Adding this metric is one of the roles of *libluaav*. Within a script in LuaAV, we can ask for the current time using the *now()* function. The time returned is logical time for the *luaav_state* scheduler, which is anchored in real-time by reference to the audio sample counter.

To grant explicit scriptable control over the scheduling capabilities of a *luaav_state*, we have extended the coroutine mechanism to allow yielding control to the scheduler by means of a *wait()* function. The arguments to *wait* can be a duration (after which the coroutine will resume)[10] or an arbitrary string (the name of an event trigger to wait for). A scheduled coroutine can be directly created using the *go* function, whose arguments can specify a duration or event to wait for before starting the coroutine, and a function and arguments to form the body of the coroutine. Of course, any valid Lua code can be placed within a coroutine; in fact the entire script itself is also a coroutine and can *wait()* and check *now()* as needed.

A code example may speak a thousand words:

```
-- define a function to print a message
-- repeatedly, every 1 second
function printer(message)
  while true do
    print(message)
    wait(1) -- wait 1 second
  end
end
-- start ticking:
go(printer, "tick")
-- start tocking after 0.5 seconds
go(0.5, printer, "tock")
```

This relatively simple interface is a low-level meta-mechanism from which more complex temporal patterns and semantics can be constructed. For example, a coroutine which returns a function will continue execution in that function's body (as a tail call in Lua), and a coroutine which returns a call to its own function will implement temporal recursion [19].

Furthermore, it is possible to create new schedulers whose metrics are driven by events within the script itself; this can be used to create a tempo clock for example.

4.3 Multi-threading and audio

Given the power of coroutines to deterministically model concurrent activities the decision by the Lua authors to shun multi-threading is easier to understand[11]. Our own approach is to maintain this single-threaded nature for the Lua interpreter instances: it is consistent with the recommended manner to interact with OpenGL contexts and GPU resources, and its deterministic assurances greatly simplify the code within *libluaav*.

Unfortunately however, audio processing is better placed in a dedicated independent high-priority thread, in which unbounded calls (such as memory allocations, garbage collections and so on) are avoided [4]. The natural result is two threads: one for the interpreter and graphics, one for the audio processing, and the problem of synchronization between them.

Our solution is to mirror state between the interpreter thread and the audio thread by means of time-stamped synchronization messages along a pair of single-reader/single-writer FIFO (first-in, first-out) message queues (built upon the JACK ringbuffer[12]). Memory allocation/disposal and initialization of audio objects occurs in the main thread, but subsequent state changes triggered from Lua code are serialized and dispatched to audio thread via the message queue. The audio thread can then retrieve these messages (up to the appropriate timestamp) and apply the state changes in the context of signal processing directly.

[9] Coroutines are subroutines that act as master programs (Conway, 1963). A coroutine in Lua is a concurrent asynchronous state with its own instruction pointer, stack and local variables, but with access to shared globals. A coroutine is constructed from a Lua function, which can explicitly yield execution and be resumed later.

[10] Durations are measured in seconds by default, however LuaAV supports the creation of arbitrary user clocks and schedulers, with which concepts of tempo and beats can be easily constructed.

[11] Introducing threading into standard Lua can be done, however the granularity is so high as to make this feature nearly useless and execution effectively single-threaded anyway.

There is necessarily a latency between the Lua thread time and the audio thread time, which is bounded by the update period and jitter of both. So long as actual latency remains below a (user-specifiable) ideal limit, fully sample-accurate temporal determinism can be achieved[12]. If ideal latency cannot be kept, events will fire late, but the order of events remains determinate.

The main drawback of this approach is that audio state cannot be immediately retrieved in the Lua script: method calls on audio objects are asynchronous and cannot return concrete values. Similarly any messages sent from the audio thread to the main thread are also latent, preventing temporally accurate triggering of Lua code in response to audio analysis for example.

4.4 Audio engine

The audio engine within LuaAV acts upon time-stamped messages received on the message queue from the Lua thread, and triggers any process calls in active audio objects (from here on denoted *Synths*[13]). Synths have notions of signal input and output ports of various kinds which can be connected to each other[14]. The connections and disconnections of ports are specified by messages from the main thread.

4.4.1 Dynamic graphs with sample accuracy

For efficiency (and to achieve real-time guarantees) signal processing graph nodes are typically computed over blocks of N sample frames with buffered input and output signal streams. The audio engine has the responsibility to ensure that buffers of data for a Synth's input ports are properly filled and the output ports properly prepared before the Synth's signal processing function is executed.

Since the block-rate is purely an implementation detail and carries no musical or aesthetic significance, we aim to hide it completely from the Lua interface; users should be able to code with state changes at any arbitrary time ('strongly timed'). In order to achieve sample-accuracy in graph dynamics while maintaining determinism in the signals, LuaAV traverses sub-sections of the graph for sample-accurate state changes. For CPU efficiency, only the upstream dependencies of the changing node(s) must be computed, and only up to the sub-block timestamp of the scheduled change.

For memory efficiency, it is better to minimize the number of buffers allocated, and re-use existing memory when it can be safely done. A good strategy would maintain a memory pool of re-usable buffers with a lazy allocation, eager recycling policy, under the control of a coloring algorithm akin to register allocation. However this strategy becomes quite complex with the combination of multiple references (buffers with multiple readers and/or multiple writers), feedback connections and sub-block size traversals. We currently only optimize for single-use non-feedback connections and defer recycling until the end of the block, but are researching more optimal algorithms.

4.5 Signal Processing

Toward efficiency and extensibility, LuaAV's Synths are built according to a low-level C API. The API provides as much functionality as possible (such as automatic bindings to Lua) without compromising flexibility. Synth code can be written in C or C++, does not need to inherit or compose any pre-existing objects, nor conform to a particular data layout.

4.5.1 Standard signal processing units

For the purposes of rapid testing and minimal dependency, a concise set of standard Synths are provided within the *libluaav* library. These units wrap low-level synthesis routines (using code from the Gamma [14] library) within abstract definitions of ports, methods and process routines. Static code-generation in the *libluaav* build process uses these definitions to automatically create bindings to the LuaAV audio API, Lua bindings, and documentation.

The following example code plays a series of sine bleeps of random frequency, whose durations progressively shorten from 1 second to 1 millisecond:

```
local outs = Outs() -- stereo output bus
for i = 1, 1000 do
  local dur = 1/i
  local f = 100 * math.random(10)
```

[12] Clock drift is not an issue per se, since we derive our source of real time from the audio sample clock itself.

[13] The term 'synth' is used rather than 'unit generator' to indicate a coarser granularity in the graph. Finer granularities are better handled by JIT compilation of synths from sub-components which better deserve to be named unit generators.

[14] Feedback between Synths necessarily incurs a block of delay. Feedback within synth implementations incurs a single-sample delay.

```
  local synth = Sine{ freq = f }
  outs:play(synth, dur)
end
```

4.5.2 Embedding CSound

Audio synthesis specification is a complex domain with a long history. We considered it practical to re-use existing interfaces and frameworks if possible. CSound in particular has a long heritage and a huge collection of signal processing primitives ('opcodes').

Embedding CSound within LuaAV was a remarkably straightforward process, thanks to the design of the CSound API [5]. CSound instances can be created in a Lua script as LuaAV *synth* objects using CSound's host-implemented audio option bound to the *libluaav* audio API. CSound synths can thus be connected with other LuaAV synths. Typically CSound instances in LuaAV have only minimal score specification, turning over the responsibility of the generation of score events to the Lua script. For example, the following code snippet plays an ascending harmonic scale on instrument 1 from the orchestra defined in "demo.csd":

```
require "csound"
local cs = csound.create("demo.csd")
play(outs, cs, 4)
for h = 1, 16 do
  cs:scoreevent(
    'i',   -- event type
    1,     -- p1 (instrument)
    now(), -- p2 (start)
    1,     -- p3 (duration)
    1.0,   -- p4 (amplitude)
    100*h  -- p5 (frequency)
  )
  wait(0.25)
end
```

CSound instances can also be created from strings of valid CSound code, opening up interesting possibilities of code-generating CSound orchestras and scores at run-time.

4.5.3 Run-time generation of signal processing code

Our primary focus for audio synthesis in LuaAV however is the generation and compilation of synthesis code from definitions specified at run-time, leveraging the the JIT capabilities of LLVM via *luaclang*. It is our view that runtime code-generation best serves the goals of extensibility and efficiency (and to a certain degree also portability [6]).

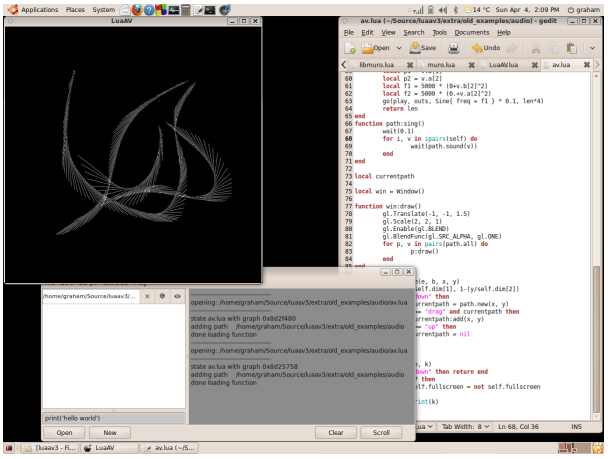

Figure 3: An audio-visual interactive canvas (mouse-paths converted to hyperbolic lines, rendered with OpenGL and sonified as grain chirps). The script itself is being edited in Gedit.

A more detailed description of our investigations can be found in [18]; here we will provide a summary for the reader's convenience. Initial experiments constructing abstract syntax trees (ASTs) of complex expressions from elementary nodes were very promising: expression trees have a natural corollary to data-flow networks typical in signal processing, and also to the static single-assignment (SSA) form of LLVM's intermediate representation (IR).

Expression graphs however are limited because they have no internal state. Extending our model to support stateful objects such as filters and variable oscillators called for the run-time generation of data-structures to maintain state across function calls, and associated routines to allocate and free memory and connect to the LuaAV audio system appropriately. We have successfully implemented such a model, and are continuing to pursue this line of development and hope that a user programming interface will stabilize soon.

The performance of the JIT compiled code is close to that of a native static compiler (GCC). The time to JIT a simple Synth can fit within the acceptable latency window between the Lua thread and the audio thread.

4.6 Beyond Audio

We have concentrated on audio in this paper, but it is important to note that LuaAV has very strong capabilities in the visual (2D and 3D) domain (see Figure 3). A near-complete binding of the OpenGL standard is included as a dy-

namically loadable Lua module, along with the Muro module which uses a generic Matrix data format to connect image, video files/cameras, GPU textures, shaders and slabs, matrix data processing and analysis, 3D mesh drawing, and supporting utilities for vectors, quaternions and other common 3D tasks.

LuaAV has MIDI and OSC built-in, and extension libraries for numerous devices and systems. And of course, anything that can be loaded in standard Lua can also be used in LuaAV, such as existing SQL database bindings, networking code, Cairo 2D drawing, PEG text parsing, and so on.

5 Future directions

It is notable that code generation appears in all three strategies to embed signal processing within LuaAV: static generation of synthesis units within the library, the potential to generate CSound orchestras programmatically at runtime, and to code-generate entire synthesis routines to machine code using LLVM/Clang. We believe it is a natural consequence of increasingly dynamic and extensible programming interfaces and environments, and which will continue to grow.

Embedded scripting language bindings to efficient library code still carry a divide between static and dynamic code which remains immutable during runtime. The incorporation of runtime JIT compilation (such as LLVM/Clang in LuaAV) adds the capacity to dynamically generate new bindings into the environment and the augmentation of existing bindings and binaries on the fly.

For example, computer vision video filters in LuaAV are code generated by bringing together functionality from OpenCV and the Muro Matrix specification to generate a new video filter with full Lua bindings which is properly adaptive to the heterogeneous nature of computer vision data. By compiling these filters at runtime, it's possible to dynamically alter how filters mix and combine with further processing elements in ways that would otherwise be preconceived and fixed.

As we have explored this area of software design, it has become apparent that the trend is toward heterogeneous computational environments that freely mix paradigms be they typed versus untyped, dynamically compiled versus statically compiled, and so on. What we are working to achieve is a continuum of paradigms as opposed to simply concatenating them together. Currently in LuaAV it is possible to mix Lua code and C code. As we develop the system further and add intermediate languages to generate new code between the paradigms, the boundary will only become more blurred.

6 Acknowledgements

With thanks for the support of the AlloSphere Research Group, University of California Santa Barbara.

References

[1] X. Amatriain, J. Kuchera-Morin, T. Hollerer, and S. T. Pope, "The allosphere: Immersive multimedia for scientific discovery and artistic exploration," *IEEE MultiMedia*, vol. 16, pp. 64–75, 2009.

[2] A. Brown and A. Sorensen, "Dynamic media arts programming in impromptu," *Proceedings of the 6th ACM SIGCHI conference on Creativity & ...*, Jan 2007.

[3] N. Collins, A. Mclean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organized Sound*, vol. 8, no. 03, pp. 321–330, 2003.

[4] R. B. Dannenberg and R. Bencina, "Design patterns for real-time computer music systems," ICMC 2005 Workshop on Real Time Systems Concepts for Computer Music, 2005.

[5] J. Ffitch, "On the design of csound5," in *Proceedings of the 3rd Linux Audio Developers Conference*, ZKM, Karlsruhe, Germany, 2004.

[6] M. S. O. Franz, "Code–generation on–the–fly: A key to portable software," 1994.

[7] D. Griffiths, "Fluxus," http://www.pawfal.org/Software/fluxus/, 2007.

[8] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua — an extensible extension language," *Software Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.

[9] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," in *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, 2004.

[10] J. McCartney, "Rethinking the computer music language: Supercollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[11] M. Pall, "LuaJIT," http://luajit.org/, 2007.

[12] Paul Davis, "Jack — connecting a world of audio," http://www.jackaudio.org/, 2010.

[13] M. Peljhan, "Common data processing and display unit-tokyo system prototype," http://www.ntticc.or.jp/ Exhibition/2009/Openspace2009/Works/ commondataprocessinganddisplayunit.html, 2009.

[14] L. Putnam, "Gamma - generic synthesis c++ library," http://mat.ucsb.edu/gamma/, 2009.

[15] W. Smith, "Abelian: A visual and spatial platform for computational audiovisual performance," Master's thesis, University of California Santa Barbara, 2007.

[16] W. Smith and G. Wakefield, "Synecdoche," http://www.mat.ucsb.edu/ whsmith/Synecdoche/, 2007.

[17] ——, "Computational audiovisual composition using lua," *Communications in Computer and Information Science*, vol. 7, pp. 213–228, 2008.

[18] ——, "Augmenting computer music with just-in-time compilation," *Proceedings of the International Computer Music Conference*, 2009.

[19] A. Sorensen and A. Brown, "Aa-cell in practice: an approach to musical live coding," in *Proceedings of the 2007 International Computer Music Conference*, 2007.

[20] G. Wakefield, "Vessel: A platform for computer music composition, interleaving sample-accurate synthesis and control," Master's thesis, University of California Santa Barbara, 2007.

[21] G. Wakefield and H. Ji, "Artificial nature: Immersive world making," in *EvoWorkshops*, 2009, pp. 597–602.

[22] G. Wakefield and W. Smith, "Using lua for audiovisual composition," in *Proceedings of the 2007 International Computer Music Conference*. International Computer Music Association, 2007.

[23] G. Wang and P. Cook, "Chuck: A programming language for on-the-fy, real-time audio synthesis and multimedia," in *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*. New York, NY, USA: ACM, 2004, pp. 812–815.