USING LUA FOR AUDIOVISUAL COMPOSTION

Graham Wakefield *Wesley Smith*

University of California Santa Barbara
Media Arts and Technology Program
Santa Barbara, California, USA

ABSTRACT

In this paper, we present new opportunities to overcome some of the inherent limitations of a visual data-flow environment such as Max/MSP/Jitter, by using domain specific (audio and graphical) extensions of the Lua programming language as libraries (externals). Lua is flexible, extensible and efficient, making it an ideal choice for designing a programmatic interface for multimedia composition.

Keywords

Lua, Max/MSP, Jitter, scheduling, microsound, OpenGL, composition, functional programming.

1. INTRODUCTION

The contemporary digital artist can choose amongst many software tools to express his or her idea. The Max family of applications (Max/MSP/Jitter [27], PureData [19], etc.) are popular choices for composing interactive digital media works because of the approachable graphical interface, many bindings to media processes and protocols, and an open-ended philosophy. The Max family implements a visual Data Flow Architecture [15] through a patching interface with both event flow and stream flow. While this type of interface is powerful and flexible, the Max/MSP/Jitter environment also carries some inherent limitations (Table 1). Driven by artistic goals [27], the authors desired a dynamic interface to overcome such issues, supporting both low-level data processing and high-level control. As Puckette notes, an ideal solution is to embed an interpreted language:

> "Rather than a programming environment, Max is fundamentally a system for scheduling real-time tasks and managing communication among them. Programming as such is better undertaken within the tasks than by building networks of existing tasks. This can be done by writing "externs" in C, or by importing entire interpreters…" [21]

Many extensions (externals) for Max exist which embed interpreted languages, whether generic languages (Javascript in *js* [29], LISP in *maxlisp* [5], Python in *py/pyext* [8]), or domain specific languages (such as *csound~* [23] and *rtcmix~* [6] for audio DSP and Ruby bindings for PD's GridFlow graphics [3]). Each address particular issues of multimedia composition but none of them support both audio and graphics while satisfying the authors' real-time performance requirements.

The authors therefore chose to embed the Lua scripting language [12] into the Max environment. Lua's authors describe Lua as an extensible extension language [11] specifically designed to be embedded within host programs and extended by domain-specific APIs. Lua is an efficient scripting language based on associative tables and functional programming capabilities [1], coroutines, and meta-mechanisms for building higher level programming structures. For an interpreted language, Lua fares extremely well in terms of speed and memory usage [1]. It is frequently used for game logic programming (e.g. World of Warcraft [25]) and application extension (e.g. Adobe Lightroom [13]).

The *lua~* and *jit.gl.lua* extensions to Max presented in this paper facilitate arbitrarily complex dynamic behaviour for audio DSP and OpenGL graphics respectively while interfacing with the convenient graphical interface and libraries of Max/MSP/Jitter. The decisions made in the design of *lua~* and *jit.gl.lua* vary according to the demands of the application domain, but both aim to balance flexibility and efficiency with a minimization of preconceptions about potential usage.

2. LUA~

The *lua~* external incorporates domain specific extensions to the Lua language for digital audio. Surprisingly few such extensions exist (notably Geiger's

Table 1. Constraints inherent in the Max visual data-flow environment

Focus on visual representation of process interconnection	Hinders expressive data structures
	Dynamic graphs and data-structures problematic
	Large numbers of processors unwieldy
	Procedural control flow difficult
	Minimal variable scoping
Most processor nodes are black boxes	Limits process control granularity
	Writing processors requires offline C development
Block-rate quantization of audio controls	Separate scheduler & semantics for control and audio
	Sample-accurate granularity and events only within black box

research project ALUA [7] and recent high-level bindings for the CSound API) and none that provide the rich degree of control to satisfy the authors' needs. Following Lua's philosophy, the audio domain extensions in *lua~* were designed to provide meta-mechanisms for digital music composition rather than a variety of preconceived musical structures (as appropriate for a domain so fraught with complexity and ambiguity [4]). Computer music compositions may involve serial and parallel processes and structures in a complex web of relationships, eventually producing samples of digital audio. Crucially, such processes and structures are dynamic and possibly actively determined in time. The *lua~* external therefore extends Lua's excellent data-description and functional programming capabilities for digital audio domain in two principal areas, both evaluated under the control of a sample-accurate scheduler:

- Concurrent functional control (via coroutines)
- Signal processing (via unit generator graphs)

A *lua~* object embedded in a Max patch can load and interpret Lua scripts that make use of these extended capabilities in order to receive, transform and produce MSP signals and Max messages accordingly. A code sample of a typical script is given in Figure 1.

2.1. Concurrent functional control

Concurrent functional control is based upon an extension of Lua coroutines. A coroutine represents an independent thread of execution for deterministic scheduling (also known as collaborative multi-tasking). In *lua~,* such coroutines are extended to be aware of the sample-clock, with a small number of additional functions to interact with the scheduler. The scheduling of control flow using coroutines in lua~ is a variation of the continuation-based enactment design pattern [13].

Coroutines are launched with the *go()* function, which takes a delay time as its first argument, and a function as its second argument[1]. Effectively a copy of this function is inserted into the scheduler's list of parallel activities, to be activated after the delay time has elapsed. All statements in a coroutine occur instantaneously with respect to the sample clock, with the exception of *wait()* and *play()*. The *wait(dur)* call will yield execution of the function body for *dur* seconds, in which time other coroutines may execute or signal processing occur, and the *now()* call returns the number of seconds since the coroutine was launched. All specifications of time are sample-accurate.

2.2. Signal Processing

Because the Max/MSP SDK API does not allow dynamic instantiation of any MSP object, a different set of signal processing unit generators has been provided, based on the efficient C++ library Synz [22]. An SDK to extend the DSP vocabulary is planned as future work.

Signal processing primitives (unit generators) are created by calling library constructor functions, such as *Sine(), Env(), Biquad()* etc. The constructor functions may themselves take numeric or unit generator inputs as their arguments, such that for example the statement `Sine(Sine(0.1) * 400 + 500)` will create a basic FM synthesis graph modulating between 100 and 900Hz ten times per second. Note that basic operators (+, *, -, /, %, ^) are overloaded for unit generators to aid legibility.

The *play(bus, dur, unit)* call adds the unit generator *unit* as an input to *bus* for a duration of *dur* seconds (yielding the coroutine in between). This bus may be the global *Out* bus, which represents the *lua~* outlets in Max/MSP, or another bus created by the programmer using *Bus()*.

```lua
-- play a panned single-cycle waveform:
function grain(dur, amp)
	local s = Sine(1/dur, 0) * amp
	local graph = Pan(s, s * (math.random() - 0.5))
	play(Out, dur, graph)
end

-- an algorithmic stream of pulses:
function pulsetrain()
	local duration = 0.005 / math.random(50) -- 0.1 to 5ms
	local pulsewidth = 0.02 / math.random(20) -- 1 to 20ms
	local fade = 0.8 + (math.random() * 0.199) -- 70 to 99%
	-- periodic counter:
	local step = 1
	local limit = math.random(12)
	-- keep playing grains until very quiet:
	local amp = 1
	while amp > 0.01 do
		go(grain, duration / step, amp) -- schedule pulse
		wait(pulsewidth * step)      -- pause for pulse width
		amp *= fade                  -- decrease amplitude
		step = (step % limit) + 1    -- interate counter
	end
end

-- schedule pulsetrains
while true do
	-- launch a new train evern 10-100ms:
	go(pulsetrain)
	wait(0.01 + (math.random() * 0.09))
end
```

Figure 1. Code sample layering multiple pulse-trains with distinct algorithmic control of pulse duration & pulse width in each train.

2.3. Avoiding block-rate

The scheduler algorithm at the heart of the lua~ external manages the coroutines and the signal processing graphs, avoiding block-rate control limitations. The scheduler lazily evaluates graph sections only when deterministically necessary, maximizing vector-processing potential where possible. Latency between inputs and outputs is only incurred for graph sections with cycles (feedback), and can be minimized to arbitrary control rates. *Lua~* thus permits a sample accurate articulation of the composition that may be dynamically deterministic. State changes that involve interpreted code to generate new signal graphs may

[1] Additional arguments are passed on to this function.

occur sub-millisecond rates, ideal for generative microsound [24].

2.4. Dynamic graphs & multiplicity

In the Max visual interface, the audio graph cannot be recompiled without audible discontinuities, limiting dynamic audio processing to static, pre-allocated structures. Similarly, the maximum number of parallel voices must also be pre-allocated (e.g. poly~ arguments). In contrast, *lua~* supports generative, dynamic signal graphs without discontinuities.

2.5. Optimization for real-time processing

The majority of scheduling and signal processing code is written in C++ for efficiency. To achieve sample accuracy, the lua~ interpreter necessarily runs in the high propriety audio OS thread, but the cost of interpreted code is minimized by only calling into Lua for the scheduled state change actions. The Lua memory allocator and garbage collector is optimized for real-time[1], and free-list memory pools are used for audio buffers and coroutines to avoid unbounded memory allocation calls.

3. JIT.GL.LUA

jit.gl.lua is a 3D graphics specific binding of the Lua scripting language for the Max/Jitter environment[2]. *jit.gl.lua* provides a compromise between execution speed and flexibility when developing custom 3D graphics routines that lies between patch objects and Javascript on the one hand and custom C externals on the other. *jit.gl.lua* is also tightly integrated with the Jitter library and in particular the 3D graphics portion of the library, easing some of the burdens of writing 3D graphics routines.

3.1. Integration with Jitter

Objects in Jitter whose name begins with *jit.gl* by convention all receive notifications from a given graphics context they are attached to. Unlike the *js* (Javascript) object, *jit.gl.lua* attaches to a graphics context and provides hooks in the embedded Lua scripting environment for receiving these notifications which can be used to automatically call the script when the graphics context calls it as well as manage context dependent resources such as sets of drawing commands stored in a displaylist. Embedded Lua scripts also have access to the *jit.gl.lua* object they are embedded in through the global *this* variable. By setting the attributes of the embedding object, global OpenGL state can be managed for the entire script and selectively overridden with low-level OpenGL commands during script execution.

In addition to integration with Jitter graphics contexts, *jit.gl.lua* provides bindings to much of the Jitter library C functions normally only accessible when writing custom C externals. The most important ones for manipulating 3D graphics are the *vecmath* and *drawinfo* libraries. The *vecmath* library is a full vector and matrix math library for 3D graphics, handling vectors of length 2, 3, and 4 as well as 3x3 and 4x4 matrices. The *drawinfo* library contains functions for low-level manipulations of the *jit.gl.texture* object for binding textures to arbitrary geometry and rendering arbitrary OpenGL commands to texture.

Within *jit.gl.lua* Jitter objects for matrix processing and higher level OpenGL functionality are made available in similar manner to the *js* object along with a number of extensions. First, named Jitter objects in a patch can be referenced within a script by utilizing Jitter's name lookup service made available through the *jit.findregistered* method. Second, an extended binding of the *jit.submatrix* object allows for the scripting of in-place submatrix processing routines with any Jitter object that processes matrices.

3.2. Support Libraries

The Lua scripting language has a built in module system for dynamically loading module-formatted scripts and binary collections of compiled C/C++ code. The module system works on Windows and all OSX platforms since 10.3. With the module system, single C/C++ functions or entirely libraries can be brought into the Max/Jitter environment without having to write an external. This enables, for example, 3D drawing routines to be prototyped in Lua and then translated into C without having to deal with the extra programming required in developing a full-fledged Jitter object.

Entire libraries can also be brought into Max/Jitter in this manner. Some libraries currently available include the Open Dynamics Engine (ODE) [26], the OpenGL View toolkit (GLV) [18], and a Matrix Operation (MOP) library. The ODE module brings a sophisticated set physical tools to Jitter, which can be used to give physical properties to elements in a 3D scene or describe a physical system for parameter manipulation as in the *pmpd* and *msd* libraries [10][17]. GLV is a graphical user interface (GUI) toolkit for OpenGL. It contains an extensible set of widgets as well as an event management system with customizable drawing routines. The MOP module is specific to Jitter and is intended to speed the development of matrix processing routines. It provides all of the functionality needed to get data from a Jitter matrix and leaves for the user to simply provide the data processing routine.

---

[1] On x86 platforms it may also be possible to JIT compile the new functions to efficient machine code [19].

[2] OpenGL bindings based on LuaGL [9].

4. CONCLUSION AND FUTURE WORK

We have presented two extensions to the Max multimedia composition environment that enable new approaches to composing within Max and overcome some of its limitations. For both extensions, we have developed new multimedia frameworks for the Lua scripting language, providing flexible and efficient interfaces for developing new works.

Though *lua~* and *jit.gl.lua* are separate objects within the Max environment, we are developing an extension to support bidirectional exchange of messages and data structures between Lua instances using 'tubes'. Tubes can be used for drawing graphics according to audio processes and vice versa, leading towards the construction of functional audiovisual entities.

A further objective is to present a standalone platform for multimedia composition that does not rely on Max as a host, merging the functionality of our audio and graphical extensions to Lua.

The externals are available for public download at:

lua~:
http://www.grahamwakefield.net/
jit.gl.lua:
http://cycling74.com/twiki/bin/view/Share/WesleySmith

5. ACKNOWLEDGEMENTS

With thanks to Lance Putnam for the Synz DSP library and the UCSB MAT GLV team for the GUI toolkit. Thanks also to the UCSB MAT InfoVis lab. Partial support provided by NSF IGERT Grant #DGE-0221713.

6. REFERENCES

[1] H. Abelson, G. J., Sussman, "Structure and Interpretation of Computer Programs," MIT Press, Massachusetts, USA, 1996.

[2] Alioth, "Computer Language Benchmarks Game", retrieved April 2007: http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=lua&lang2=javascript.

[3] M. Bouchard, "GridFlow 0.8.4 C++/Ruby Internals", retrieved April 2007; http://gridflow.ca/latest/doc/internals.html.

[4] R. Dannenberg, P. Desain, H. Honing, "Programming Language Design for Music" in *Musical Signal Processing*, Swets & Zeitlinger, Netherlands, 1997.

[5] B. Garton, "Maxlisp v0.8", July 2004; http://www.music.columbia.edu/~brad/maxlisp/

[6] Garton, B. and D. Topper, "RTcmix – using CMIX in real time," In *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1997.

[7] G. Geiger, *Abstraction in Computer Music Software Systems*, doctoral thesis, Department of Technology, Universitat Pompeu Fabra, Barcelona, Spain (2005).

[8] T. Grill, "Py/Pyext", retrieved April 2007; http://grrrr.org/ext/py/.

[9] F. Guerra, "LuaGL", retrieved April 2007; http://luagl.wikidot.com/.

[10] C. Henry, A. Momeni, "Dynamic Independent Mapping Layers for Concurrent Control of Audio and Video Synthesis," Computer Music Journal, 30:1, pp.49–66, Spring 2006.

[11] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, "Lua - an extensible extension language", in *Software: Practice & Experience* 26 #6 (1996) 635-652, 1996.

[12] R. Ierusalimschy, "Programming in Lua" (2nd ed.) PUC-Rio, Rio de Janeiro, 2006.

[13] R. Ierusalimschy, L. H. de Figueirdo, W. Celes, "The Evolution of Lua", to appear in ACM HOPL III, 2007.

[14] D. Manolescu, "Workflow Enactment with Continuation and Future Objects," in OOPLSA'02, Seattle, WA, 2002.

[15] D. Manolescu, "A Dataflow Pattern Language" in Proceedings of the 4th Pattern Languages of Programming Conference, 1997.

[16] J. McCartney, "Rethinking the Computer Music Language: SuperCollider," Computer Music Journal 26, 4 (2002), 61–68.

[17] N. Montgermont, "Modèles Physiques Particulaires en Environement Temps-réel: Application au contrôle des paramètres de synthèse", Masters thesis, Université Pierre et Marie Curie, Paris, France, 2005.

[18] E. Newman, L. Putnam, W. Smith, G. Wakefield, "GLV – OpenGL Application Building Kit," December 2006; http://glv.mat.ucsb.edu/.

[19] M. Pall, "LuaJIT", retrieved April 2007; http://luajit.luaforge.net/.

[20] M. Puckette, "Pure Data," in Proceedings of the 1997 International Music Conference (ICMC '97) (1997), Computer Music Association, pp. 224–227.

[21] M. Puckette, "Max at Seventeen," Computer Music Journal 26, 4 (2002), 31-43.

[22] L. Putnam, "Synz", retrieved April 2007; http://www.uweb.ucsb.edu/~ljputnam/synz.html.

[23] D. Pyon, "Csound~", retrieved April 2007; http://www.davixology.com/csound~.html.

[24] C. Roads, "Microsound," MIT Press, Cambridge, MA, USA, 2001.

[25] Rustak, "WoWWiki, The Warcraft wiki", retrieved April 2007; http://www.wowwiki.com/UI_Beginners_Guide.

[26] R. Smith, "Open Dynamics Engine", retrieved April 2007; http://www.ode.org/.

[27] W. Smith, G. Wakefield, "Synecdoche", January 2007; http://www.mat.ucsb.edu/~whsmith/Synecdoche/.

[28] D. Zicarelli, "How I Learned to Love a Program that Does Nothing" Computer Music Journal 26, 4 (2002), 44-51.

[29] D. Zicarelli, J. K. Clayton, "Javascript in Max," in Max/MSP Complete Documentation, retrieved April 2007; http://www.cycling74.com/download/maxmsp46doc.zip.