Real-time Multimedia Composition using Lua

Wesley Smith
Media Arts and Technology Program
University of California Santa Barbara
wesley.hoke@gmail.com

Graham Wakefield
Media Arts and Technology Program
University of California Santa Barbara
wakefield@mat.ucsb.edu

ABSTRACT

In this paper, a new interface for programming multimedia compositions in Max/MSP/Jitter using the Lua scripting language is presented. Lua is extensible and efficient making it an ideal choice for designing a programmatic interface for multimedia compositions. First, we discuss the distinctions of graphical and textual interfaces for composition and the requirements for a productive compositional workflow, and then we describe domain specific implementations of Lua bindings as Max externals for graphics and audio in that order.

Categories and Subject Descriptors

H.5.1 [**Information Interfaces and Presentation**] Multimedia Information Systems - *animations*; H.5.5 [**Information Interfaces and Presentation**] Sound and Music Computing - *methodologies & techniques, modeling, signal synthesis and processing*; I.3.7 [**Computer Graphics**] Three-Dimensional Graphics and Realism - *animation*; J.5 [**Arts and Humanities**] –*arts, fine arts and performing*, *music*.

Keywords

Lua, Max/MSP, Jitter, OpenGL, audiovisual composition.

1. INTRODUCTION

The contemporary digital artist can choose amongst many software tools to express his or her idea, however frequently there is a great challenge in manipulating such tools to express complex time-variant structures. Real-time multimedia software must handle continuous flows of interacting heterogeneous data streams yet provide a flexible interface for describing interactions between media, all the way from low-level data and memory processing up to the more abstract structures of content and form.

1.1 Max/MSP/Jitter

The Max/MSP/Jitter environment is a popular choice for constructing complex interactive digital media works. Amongst its strengths are bindings to many different media types and protocols and the open-ended philosophy: Max carries few preconceptions regarding how or for what it should be used [27].

However as with many multi-media description documents, the Max/MSP/Jitter 'patch' is a mostly static structure: a fixed number of objects connected in a statically defined data-flow graph. This paper presents two extensions to Max/MSP/Jitter as opportunities to bridge between the statically defined data-flow graph of a 'patch', and the flexible programming language Lua.

1.2 Lua

Lua is a powerful, efficient scripting language based on associative tables and functional programming techniques. Lua is ideally suited for real-time multimedia processing because of its flexible semantics and fast performance; it is frequently used for game logic programming (e.g. World of Warcraft [24]) and application extension (e.g. Adobe Lightroom) [11].

1.3 jit.gl.lua and lua~

The jit.gl.lua (for OpenGL graphics) and Lua~ (for audio DSP) extensions allow a new range of expressive control in which arbitrary, dynamic data structures and behaviors can be generated, whilst interfacing with the convenient graphical interface and libraries of Max/MSP/Jitter. The development of these extensions has been driven by the presenters' artistic goals and requirements [16].

The ability to work with Max in a dynamic text-based environment addresses some of the major interaction problems of graphical programming environments. With scripts, the composer can create dynamically changing networks of processes, model objects and behaviors that aren't otherwise available without creating a C external, and implement control flow logic more compactly.

2. MULTIMEDIA COMPOSITION ENVIRONMENTS

2.1 Digital media composition

Designing an environment for multimedia composition in the digital domain requires constructing a model of multimedia representation. The ideal specificity of a model depends upon the problem it is trying to solve, however digital media composition is a highly unconstrained problem. A good model should balance design for efficiency with minimization of preconceptions about its potential usage.

Building a multimedia composition where audio, video, 3D graphics, MIDI, OSC, and other hardware and communication protocols need to be pulled together in real-time to realize the composition requires a good amount of technical knowledge as well as a large, modular code base. To handle the complexity of this task and allow rapid sketching of compositional ideas, a number of general purpose multimedia composition environments have been developed of which the Max/PD family [27, 19] is one of the most widely used. With a basic knowledge of the environment, a user can link together sonic and visual processes and interactively construct and audiovisual composition.

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

Focus on visual representation of process interconnection	Hinders expressive data structures
	Dynamic graphs 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 messages for audio processors	Separate scheduler & semantics for control and audio
	Sample accurate granularity & event triggers only within black box

2.2 Using visual data-flow programming for multimedia composition

The Data Flow Architecture pattern [14, 20] performs sorted operations on data elements (tokens) through a directed graph of modules (nodes) connected via ports (arcs). Modules read tokens on their input ports, process them according to a module-specific internal algorithm, and then write tokens into their output ports. Tokens can flow through the network in regular rates (stream flow) or irregular rates (event flow).

The Max family implements data flow through a visual patching interface in which audio is processed in stream flow, while other data types are processed in event flow. Users connect processing nodes together via 'patch cords', analogous to patching together a modular synthesizer. While this type of environment facilitates rapid sketching, the Max visual data-flow interface also carries some inherent limitations (Figure 1).

> "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…" [27]

Extending Max by importing interpreters is a very powerful way to add new functionality to the composition environment. Interpreters bring an entirely new interface to Max with new modes of interaction and new methods of composition. Many extensions (externals) for Max exist which embed interpreted languages, whether generic languages (Javascript in *js* [28], LISP in *maxlisp* [4], Python in *py/pyext* [6]), or domain specific languages (such as *csound~* [22] and *rtcmix~* [5] for audio DSP and Ruby bindings for PD's GridFlow graphics [3]). Each of these language bindings address particular issues of multimedia composition within Max but none of them present a unified framework for composition with audio and graphics while remaining sensitive to the performance issues of real-time systems.

2.3 Lua used as a Domain Specific Language for Multimedia

For real-time multimedia composition, interpreted languages must be expressive, efficient to work with, have a low computational cost, and be flexible in terms of how they fit into the compositional workflow. The Lua scripting language has become widely adopted by the game developer community for precisely these reasons. Lua's authors describe Lua as an extension language [9] specifically designed to be embedded within host programs and extended by domain-specific APIs for interaction with the host program. Lua meets the needs of an extension language by providing good data description facilities (associative tables that can also function as arrays), using clear and simple syntax (suitable for non-professional users), offering flexible semantics for extensibility, being small and easy to embed, and without needing to satisfy generic programming constraints such as static type-checking.

Lua has grown from a configuration language to a full-fledged programming language, supporting higher-level features such as first-class functions and coroutines. Higher-level programming structures can be designed for the domain specific API from basic meta-mechanisms provided by Lua (e.g. by using metatables one can support many different types of inheritance). Finally, for an interpreted language, Lua fares very well in terms of speed [1] and memory usage (the virtual machine only has 38 instructions), and incorporates an incremental garbage collector suitable for real-time use.

3. BENEFITS AND IMPLEMENTATION

In this section we discuss some of the general benefits of embedding Lua with the Max environment. Different domains have different constraints and call for different design philosophies in their programming interfaces. In the following sections we present domain specific implementations of Lua bound Max externals for real-time graphics & audio respectively.

3.1 Textual rather than visual data representation

Lua's extensible data description facilities make it very easy to define complex and expressive hierarchical data structures suitable for formal descriptions of multimedia content. Since the data description is textual and programmatic, it can be generative, support large numbers of objects and - most importantly for multimedia composition - it can be dynamic over time.

Furthermore, hierarchical data structures and function calls in Lua support lexical scoping, such that local and (relatively)

global variables can be used (in contrast, Max variable names are almost always global).

3.2 Procedural control structures

Control structures natural to textual programming languages (if, while, for etc.) are often difficult to represent and may be inefficient in a visual data-flow patch, but can be highly expressive tools for multimedia composition.

Textual interfaces generally have much less of an overhead because events are compiled by a virtual machine into byte code rather than message-passing graph iterations.

3.3 Dynamic structure

The primary material of multimedia compositions is time. Using textual interfaces to model dynamic structures can be much more natural than with visual data-flow interfaces. Textual interfaces can describe functional behavior as well as node parameters and data flow connections.

In Max, time is mostly implicit in the flow of events and signals through the graph, which is essentially static. Changing a patch or generating new patches while the system is active through Max scripting is cumbersome and far from straightforward.

3.4 Multiplicity

A composition may require a number of concurrent networks of synthesis and processing, such as multiple voices of a synthesizer. Since duplicating sections of a visual data flow graph is scalable for small numbers only, the Max environment provides support for multiplicity via the poly~ object. This solution however is limited in that the duplicated sections must be identical, and an upper limit on the number of items is fixed.

If hundreds or thousands of sub-networks are required (e.g. granular synthesis, particle systems), the visual dataflow environment simply fails, and black box externals are the only solution. But since black boxes are opaque, when multiplicity, dynamic structure and/or low-level control are required together, even the black box solution becomes insufficient.

A textual interface can generate data structures generatively more easily. Avoiding visual representation increases efficiency for large numbers of instances.

3.5 Creating new black boxes

At times during the development of a composition, it can become advantageous or even necessary to create new processing modules within the dataflow environment (using C, C++ or Java). Usually this demand comes when the environment itself does not support a certain required behavior, whether this is due to a limitation in the environment's model (such as those outlined above), whether it is because a third partly library extension is required (e.g. wrapping another C library) or whether it is simply necessary to freeze a section of the patch into more efficient low level code because of hardware processing overhead.

Developing new processing modules ("externals") forces the user into a much more technical and unforgiving way of interacting with the computer. Even if the composer has access to the programming expertise required, ideas are no longer fluid but have to be thoroughly vetted by an incremental development and test process that diverts mental energy from the core ideas of the composition.

Using an interpreted environment, the composer can create new objects (new classes or modules) within the same environment as the compositional structure, during playback. On x86 platforms it may also be possible to JIT compile the new functions to efficient machine code [18]. It may even be possible to create new objects and functions generatively at runtime.

3.6 Mixing high level and low level control

For an audiovisual composition environment to be effective, composers need to be able work with a mix of media at various scales in an interactive and intuitive fashion. A complex set of tradeoffs has to be navigated in order to determine at what scale to expose the programmatic interface where issues of performance, ease of use, and expressivity are determined. The programmatic interface is how the application's underlying structure is exposed to the user.

With Lua, it is natural to work on a number of programmatic interface levels simultaneously, mixing high- and low-level C-based constructs with more intricate script-built structures. Because of its speed and efficiency, it is not uncommon for basic C functions to be bound directly to Lua along with higher-level objects like C++ classes or C structs. This gives the user flexibility in programmatic interface, enabling precise and efficient code while maintaining the dynamicity of the Lua environment.

The decisions made in the design of jit.gl.lua and lua~ vary according to the demands of the application domain. Because audio processing runs in a high-priority thread, lua~ provides higher level abstractions to minimize memory allocation and inefficient calls across language boundary, while the more stateful nature of graphics processing supports a lower level programming interface in jit.gl.lua.

4. The jit.gl.lua external for graphics

jit.gl.lua provides both a high-level programmatic interface for Jitter very similar to the JavaScript Max external as well as a low-level interface into OpenGL and a number of C support libraries. The purpose of jit.gl.lua is to enable the rapid sketching of 3D graphics processes and to provide a smooth transition from initial sketch to final implementation in higher performance compiled languages such as C/C++. Within jit.gl.lua, low-level C functions for manipulating 3D graphics are exposed for direct access to the OpenGL API[1] as well as a host of support libraries available from the Jitter API. These include a full vector math library for dealing with vectors or length 2, 3, and 4 as well as quaternions and 3x3 and 4x4 matrix operations, which are essential for working in

[1] Based on the LuaGL project [7]

Figure 2. Code sample in jit.gl.lua, using tail calls in Lua to turn a set of points into a more complex form

```
local node
--template for node transformations
function transform(angle, axis)
  return
  function(d, x, y, z)
    gl.Vertex(x, y, z)
    node(d, unpack(vec3.transform_axisangle(
               {x, y, z}, angle, axis)))
  end
end
--table of transformation node functions
local templates = {
  transform(20, {1, 0, 0}),
  transform(20, {0, 1, 0}),
  transform(20, {0, 0, 1}),
  transform(10, {0, 1, 1}),
  transform(10, {1, 1, 0}),
  transform(10, {1, 0, 1}),
}
--recursively draw and transform points
local counter = 0
function node(depth, x, y, z)
  if depth > 0 then
    local t = templates[counter]
    counter = (counter % #templates) + 1
    return t(depth - 1, x, y, z)  --tail call
  else
    gl.Vertex(x, y, z)         --tail end
  end
end
--draw and array of points transformed
recursively
function draw()
  for i=1, 200 do
    gl.Color(i/200, i/200, i/200, 0.2)
    local x = math.floor(i/20)/10-1
    local y = (i%20)/10-1
    gl.Begin("LINE_STRIP")
      node(36, x, y, i/200)
    gl.End()
  end
end
```

3D. There are also mid-level OpenGL support function for binding and rendering to textures.

Having the ability to mix many different levels of function calls within a single programmatic interface from inside the composition environment is very powerful from a productivity perspective. A user can start a sketch with high-level objects and incrementally refine it with low-level calls that more precisely specify both aesthetic goals and desires for computational efficiency by managing the OpenGL state machine in a more optimized manner. If performance is still an issue once a sketch is finalized, it is straightforward to translate the design into C/C++ and load it back into Lua

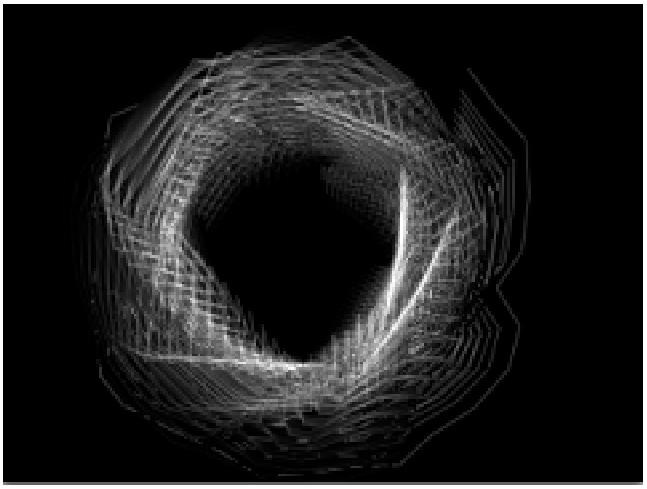

Figure 3. Patch containing jit.gl.lua script and rendering result

through the module system. In addition, external code libraries can be brought into the jit.gl.lua through the module system.

4.1 Support Libraries

Aside from the OpenGL and Jitter bindings, jit.gl.lua is extensible through the Lua module system. Three modules have been developed specifically for jit.gl.lua. They are bindings the Open Dynamics Engine (ODE), a Matrix OPeration (MOP) module for custom Jitter matrix processing, and a flexible OpenGL GUI module called GLV.

4.1.1 ODE Module

The ODE module enables the simulation of rigid body dynamics within a script running in jit.gl.lua. The most obvious use of the ODE module is as a physics solver for driving 3D graphics. ODE handles collision detection, uneven mass distributions, and a number of joint types [30]. The ODE module can also be used for high-level parameter control by using data from physical simulations to drive the parameter values in complex ways as in the techniques developed in such packages for Max/PD like pmpd by Cyrille Henry and msd by Montgermon [8, 16].

4.1.2 MOP Module

The MOP module is slightly different in nature in that its interface is both within C and Lua. The concept of the MOP module is to enable the rapid development of matrix processing algorithms. Normally custom matrix filters have to be written as Jitter objects in C, which while powerful is also time consuming. Writing such filters in scripting languages is too slow for real-time application because the amount of data being processed bogs down virtual machines using dynamic evaluation on each iteration of the for loops.

The MOP module presents a C interface that allows the user to only write the for loops iterating over matrix data instead of having to write the additional boilerplate code required to build a Jitter object. This interface provides an intermediary step between offline prototyping of matrix processing

algorithms and writing a full Jitter object, giving the developer an opportunity to concentrate on algorithm development and evaluate it in a real-time setting. The MOP module works by defining classes of MOP algorithms such as 1in/1out or 2in/1out etc. to which custom processing loops can be attache.

Within the MOP module, each class of MOP algorithm is defined by a generic setup function that takes the high-level matrix objects and hands the raw data to a defined custom processing function. For a given class of algorithm, there is a corresponding signature for the processing function. When the MOP module is loaded, the generic functions are pushed into Lua as closures. Closures allow attach extra data to C functions such that when they are called from Lua, the data can be accessed from the C function. Each custom processing algorithm is thus a generic C function corresponding to its class and a unique value used for looking up the function when the generic class function is invoked.

4.1.3 GLV Module

The GLV module contains a library for designing hardware accelerated OpenGL GUIs [17]. It consists of a basic set of widgets written in C++ along with an object representing the context of the widgets within a given OpenGL window. GLV only comes with a few widgets but it can still meet a wide variety of aesthetic and behavioral needs by leveraging the semantics of Lua to make each widget instance customizable.

Each widget has both a metatable and an environment table. The metatable contains the class model while the environment holds instance specific data. Custom behavior is defined by assigning Lua functions to predefined fields that are accessed when the widget receives events from the GLV context. Functions like MouseDown, KeyUp, and Draw can be overridden in Lua to create custom widgets in Lua scripts that derive from the base C++ classes.

5. The lua~ external for audio

The lua~ external embeds the Lua programming language inside Max/MSP. When instantiated within a patch, the lua~ external can send and receive discrete messages and receive, transform and produce continuous signal streams. Instance arguments determine the number of signal inlets and outlets, and the name of a Lua script file to load and interpret. A concrete example is given in Figure 4.

5.1 Musical representation in lua~

The domain of musical signal representation itself is fraught with complexity and ambiguity (as noted by Dannenberg et al [2]). A musical structure may be a tangled hierarchy of conceptual strata that interleave with variable dependencies. It can be described at least as both vertical (concurrent) and horizontal (sequential), including layers of containment (notes have pitch, duration etc.). However, the structural organization, containment and dependency may be quite flexible over time. Clearly a language for digital audio composition should offer flexible notions of containment, time, process and concurrency.

Following Lua's philosophy, the design of the *lua~* API aims to provide meta-mechanisms for digital music composition rather than a variety of preconceived musical structures. The Lua language already provides excellent data-description and functional programming facilities, however the lua~ external extends Lua's concurrent capabilities for the digital audio domain in two distinct forms:

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

5.1.1 Concurrent control: go(), wait() and now()

Concurrent functional logic is modeled using an extended form of Lua coroutines. A coroutine in Lua represents an independent thread of execution for deterministic scheduling (also known as collaborative multi-tasking). A coroutine has its own stack, local variables and instruction pointer (it resumes from the same code point at which it last yielded), but shares non-local variables with other coroutines. In *lua~*, these coroutines are extended to be aware of the sample-clock. Coroutines in *lua~* can include the full range of dynamic control structures that the host language offers, along with a small number of additional functions to interact with the scheduler.

The *go(delay, func, args…)* call creates a new coroutine. The coroutine will begin after *delay* seconds (or immediately if not specified), will be based upon the function *func*, which will be passed all values in *args*. The construction of coroutines using *go()* is a variation of the continuation-based enactment design pattern [13].

Within a coroutine body, 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.

5.1.2 Signal processing: unit generators and play()

Because the Max/MSP SDK API does not allow dynamic instantiation of MSP objects within other MSP objects, a different set of signal processing primitives (unit generators) was necessarily embedded. These primitives are based on efficient C++ DSP code from Lance Putnam's Synz library [26].

Unit generators are created by calling 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. Basic operators (+, *, -, /, %, ^) are overloaded for unit generators to aid legibility, and the returned unit generator objects provide methods to access and modify their inputs and state variables. Individual channels of a unit generator can be accessed with the *unit[n]* notation, and *#unit* returns the total number of channels (unit generators increase output channels to match input channels by default).

The *play(bus, dur, unit)* call adds the unit generator *unit* as an input to *bus*, yields the containing coroutine for *dur* seconds (equivalent to *wait(dur)*), then removes the unit generator from *bus*. Typically this bus will be the global *Out* bus, which represents the *lua~* object outlets in Max/MSP (and likewise,

the *In* bus represents the signal inlets), but the programmer can create other busses as required and use them in place of unit generators for complex graphs.

```
-- plays an enveloped impulse, panned by own waveform:
function grain(dur, amp)
    -- bipolar impulse with 8 harmonics:
    local s = Imp(1/dur, 8, 1)
    -- gaussian envelope & sample-rate panning:
    local graph = Pan(Env(dur, s * amp), s)
    -- play it!
    play(Out, dur, graph)
end

-- a gesture to schedule a stream of pulses:
function pulsetrain()
    -- duration from 0.25 to 2 milliseconds:
    local dur = 0.002 / math.random(8)
    -- pulsewidth from 2.5 to 20 milliseconds:
    local width = 0.02 / math.random(8)
    -- decay scalar from 50 to 99%
    local fade = 0.5 + (math.random() * 0.499)
    -- keep playing grains until very quiet:
    local amp = 1
    while amp > 0.01 do
        -- schedule grain:
        go(grain, dur, amp)
        -- increase pulsewidth:
        wait(dur + now() * width)
        -- decrease amplitude:
        amp *= fade
    end
end

-- schedule pulsetrains
while true do
    go(pulsetrain)
    -- wait between 20 and 120 ms between each train:
    wait(0.02 + (math.random() * 0.1))
end
```

Figure 4. *lua~* code sample generating pulse-trains with stochastic properties, with increasing pulse-widths and decreasing amplitude patterns within each scheduled train.

5.1.3 The scheduler

At its heart, *lua~* incorporates a scheduler which manages both the queue of active coroutines and the synthesis graph of unit generators.

In summary, the scheduler 'wakes up' each coroutine timeline as its internal timestamp is due, and the coroutine proceeds through its virtual machine instructions until it completes or it yields to reschedule itself at a future sample-clock time. Any state changes or dependencies in the synthesis graph that are triggered by the coroutine execution will trigger the appropriate nodes in the synthesis graph to be processed up to the current timestamp. Once all timelines are complete for the current audio block, the synthesis graph is traversed from the root node, to calculate any remaining indeterminate samples.

Graph traversal thus occurs dynamically to ensure that both unit generator input dependencies and deterministic control flow are maintained. The scheduler lazily evaluates graph sections only when deterministically necessary, maximizing vector-processing potential where possible.

5.1.4 Optimization for real-time processing

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. In addition, free-list memory pools are used for buffers, unit generators and coroutines to avoid unbounded memory allocation calls.

5.2 Overcoming limitations in Max/MSP:

In addition to the general advantages of an interpreted textual environment as outlined in section 2, lua~ introduces two special advantages to Max/MSP for digital audio composition:

5.2.1 Dynamic graphs and multiplicity

If in the Max visual interface audio signal processing units are added or removed at runtime, the audio graph is recompiled causing audible discontinuities, effectively preventing dynamic signal graphs in performance. Pre-allocated graph sections can be enabled and disabled, but not structurally modified. Similarly, the maximum number of parallel voices must also be pre-allocated (e.g. *poly~* arguments). In contrast, *lua~* supports dynamic signal graphs without discontinuities through the design of the scheduling and memory allocation algorithms, with the limit on multiplicity being only CPU and memory bound.

5.2.2 Avoiding block-rate state change

Using the sample-accurate scheduler, there is no longer a notion of block-rate or control-rate. As a result, the *lua~* external can schedule state changes that involve interpreted code to generate new signal graphs at sub-millisecond rates, ideal for generative microsound [28]. The *lua~* external was designed to support a potential use-case of hundreds of simultaneous sonic grains, in which each grain has a different run-time determined sonic graph.

6. Conclusions

We have presented two extensions to the Max multimedia composition environment that enable new approaches to composing within Max. For both extensions, we have developed new multimedia frameworks for the Lua scripting language, providing flexible and efficient interfaces for developing new works. These frameworks will continue to be extended in the future.

Using an interpreted environment, the composer can create new functionality within the same environment as the compositional structure during playback. The composer can create dynamically changing graphs of processes, model complex objects and behaviours unavailable to the basic Max patch, and implement expressive control logic more compactly. Since the data description is textual and programmatic, it can be generative, handle large and

unpredictable numbers of objects, support lexical scoping (local variables) and be highly dynamic over time.

A key future 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:

jit.gl.lua:
http://cycling74.com/twiki/bin/view/Share/WesleySmith

lua~: http://www.grahamwakefield.net/

7. ACKNOWLEDGMENTS

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

8. REFERENCES

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

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

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

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

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

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

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

[8] C. Henry, A. Momeni. Computer Music Journal, 30:1, pp.49–66, Spring 2006

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

[10] R. Ierusalimschy, "Programming in Lua" 2nd ed. Rio de Janeiro, 2006.

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

[12] A. Kauppi, "Lua Lanes – multithreading in Lua", April 2007; http://kotisivu.dnainternet.net/askok/bin/lanes.html.

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

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

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

[16] 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.

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

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

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

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

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

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

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

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

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

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

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

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