Computational Audiovisual Composition Using Lua

Wesley Smith and Graham Wakefield

Media Arts and Technology, University of California, Santa Barbara
Santa Barbara, CA 93106, USA
[whsmith|wakefield]@mat.ucsb.edu

Abstract. We describe extensions to the Lua programming language constituting a novel platform to support practice and investigation in computational audiovisual composition. Significantly, these extensions enable the tight real-time integration of computation, time, sound and space, and follow a modus operandi of development going back to immanent properties of the domain.

Keywords: Audiovisual, Composition, Real-Time Multimedia, Lua, Scripting Language, Functional Programming, Domain Oriented Languages, Coroutines, Computational Aesthetics.

1 Introduction

In this paper we document extensions to the Lua programming language to support time-based audiovisual composition. Significantly, these extensions enable the tight real-time integration of computation, time, sound and space, and follow a modus operandi of development going back to immanent properties of the domain.

In general terms, we are interested in enabling and encouraging audiovisual composition with an elevated aesthetic role of computation beyond the computer-aided or computer-assisted. In particular, we are focusing on a computer programming language as the primary interface to composition, but also consider roles of computation as aesthetic subject, inspiration and perhaps even collaborator. A number of general benefits can be immediately identified:

- Introducing new expressive potential through formal generality and extensibility.
- Creative freedom and independence due to an increased role of the artist in the specification of the result.
- An appropriate means to work with the aesthetic values of process, data and algorithm.
- Ease of interaction with symbolic elements from other disciplines (examples might include mathematics, physics, systems biology, semiotics...)

Our development is targeted towards two specific goals. Firstly, we aim for a tight integration of time, space, sound and computation, motivated by the artistic possibilities engendered. Secondly, we begin from analyses of the immanent elements in computation, time, space and sound. This is partly to support a sufficiently grounded yet more generic and extensible range of expression, and partly since existing idioms may become inappropriate or even counter-productive as novel aesthetic activities develop.

We are responding to a tendency within the co-evolution of audiovisual art and computational technology that might constitute a new domain or field of discipline. This tendency can be characterized as an increasing focus on computation within aesthetic practice, and a deeper engagement with the medium that may be incompatible with antecedents. This tendency is quite contemporary, responding to the rising cultural infusion of computing and cross-modal concerns. For convenience we denote this tendency – the subject matter of our project – *computational audiovisual composition*[1].

Following an elaboration of these concerns, section 2 discusses our technical implementation in general terms. Sections 3 through 5 expound our efforts to extend Lua to meet the identified immanent demands of this emerging field in respective temporal, spatial and sonic terms, followed by concluding remarks in section 6.

2 Implementation Strategy

2.1 Background

The demand for tightly integrated modalities based upon immanent elements leads to the identification of two key design sites:

- Lower level components.
- Generic 'meta-mechanisms' rather than numerous features.

Lower level components may be necessary to support tight interaction between heterogeneous modalities and also to encourage the exploration of computational audiovisual potential. For example, in the design of a computer game, spatializable sound file playback may be sufficient, however the elevated role of computation in algorithmic composition suggests a lower level approach in terms of signal processing.

The *meta-mechanism* terminology is drawn from Lua itself. Lua's authors avoid a language bloated by numerous specialized features by providing a small set of components with which a user can build desired features in the way he or she wants them. For example, Lua offers no object-oriented class inheritance system, however many variations of object inheritance can be built upon its *metatable* construct. Meta-mechanisms avoid preconceived behavior through generality and thus encourages composers to explicitly specify the behavioral features of their works. Generalizing mechanism may also support cross-modality.

On the Choice of a Programming Language The design of programming environments for audiovisual composition can be broadly divided into two camps: visual Graphical User Interface (GUI) environments and text-based environments utilizing domain specific languages. While generally more approachable for novice users, GUI-based environments such as Max/MSP/Jitter and Pd/GEM (Puckette, 2002) lack many of the formal notions a programming language embodies. Such environments more specifically address task scheduling and message passing rather than the computational

[1] Note that *composition* is intended to be equally applicable to real-time performance.

generality of our endeavor.[2] In contrast, a programming language as primary interface addresses the design sites listed above while making use of notions immanent to computation itself.

Regarding composition, Roads (2001) identifies two key benefits of programming language representations: ”First, the compositional logic is made explicit, creating a system with a degree of formal consistency. Second, rather than abdicating decision-making to the computer, composers can use procedures to extend control over many more processes than they could manage with manual techniques.”

Using a programming language as the primary artistic interface leads to a notion of composition-as-script. This notion is not new: a number of contemporary tools emphasizing the aesthetic role of computation such as SuperCollider (McCartney, 2002) and Fluxus (Griffiths, 2007) exemplify the same concerns for independently musical or spatiotemporal media.

Furthermore we note that many of these tools build upon existing languages rather than invent new ones. This leads to a number of clear benefits:

- Documentation easing learning curves
- Potentially numerous extension libraries adding additional capability for free
- Code re-use: repositories and portability
- Revision, debugging and profiling minimizing error and improving user experience
- Facilities formally verified in the community, supporting future scalability

Given these advantages, writing a new language only seems appropriate if no existing language can satisfy the demands of a domain. A preferable solution is to use a flexible language that is clearly designed for extension to specific domains: a *domain oriented* language (Thomas and Barry, 2003).

Lua The Lua programming language has been designed throughout as an ‘extensible extension language.’ *Extensible* refers to the ease by which new semantic and functional features can be added within Lua or through underlying C functions, while *extension* refers to the ease by which Lua can be embedded within a host application as a higher-level configuration and control language (*Adobe Lightroom* and *World of Warcraft* are notable examples (Ierusalimschy et al., 2007)). The author of SuperCollider states that an extensible language allows the programmer to “focus only on the problem and less on satisfying the constraints and limitations of the language’s abstractions and the computer’s hardware” (McCartney, 2002).

Typical use of Lua as an extension languages combines libraries of performance sensitive code and key data structures modeled in C/C++ with the dynamic aspects of the application such as control flow and interface configuration written in Lua. Together with a number of generic benefits summarized in Table 1, these features make Lua ideal for the development of the domain-specific language in our project.

[2] For a more detailed analysis see (Wakefield and Smith, 2007).

Data-description language	Suitable as a readable and enactive document format
High-level programming features	Compact descriptions of complex algorithms and relationships, support for both imperative and functional styles
Incremental garbage collector	Driven by game-programming needs, suitable for real-time performance
Well-regarded performance	Amongst the fastest languages of its class
Ease of development	ANSI C, small library, well defined API
Flexible license	Compatible with both GPL and commercial use

Table 1: Noteworthy benefits of the Lua language.

2.2 Contributions

We have extended the Lua programming language into the domains of temporal, spatial, and sonic composition. These extensions, and the associated host application, are together called *LuaAV*. This is the focal point for our explorations in computational audiovisual composition as a medium for compositional, technical, and philosophical inquiry within the audiovisual arts.

The core application of LuaAV is a simple platform upon which users can define custom application environments (such as window arrangement and menu structure) and execute Lua scripts in order to actualize compositions into performances. These scripts make use of the grammar and vocabulary added by our domain-specific extensions. The extensions themselves are embodied as dynamically loadable Lua libraries (*modules*), making them independent of the core application. Consequently, any other application embedding Lua that allows a user to load scripts and modules will be able to execute scripts written using LuaAV. The subsequent sections of this paper detail the development of these modules.

LuaAV is intended to be cross-platform. OS dependent support resources such as windowing and file management are currently implemented on the OSX native Cocoa (Apple Computer – Cocoa, 2008) framework as well as the cross-platform GLUT (Kilgard, 2008) library, but will be mirrored with native resources for Linux and Windows in the near future. A public website of software resources, documentation and community pages related to LuaAV, and computational audiovisual composition using Lua in general, can be found at http://www.mat.ucsb.edu/lua-av, with a related community mailing list at lua-av@mat.ucsb.edu.

3 Temporal Structure

Sharing temporal mechanisms between sonic and spatial structures clearly supports generality, thus we begin with temporal support in the form of the Lua module *time*. This module must provide a low-level basis for the gradual determination of audiovisual output, and present a high-level interface for the gradual articulation of temporal expression, making use of notions immanent to computational audiovisual composition.

3.1 Computational Time and Composition Time

The fundamental force of time in computer architecture is the inexorable movement from one discrete instruction to the next. Computational ideas must be encoded into an executing state space via a series of such instructions. Any desired temporal structure in the executing performance process must emerge from this state space. From the standpoint of the developer, the problem can be restated as a low-level, generic mapping of composition time, i.e. the flow of articulation in performance, to computing time, i.e. the determined series of instructions.

We also note that the computational instruction flow is topological, in that the temporal distance between events is unstated. In practice individual instructions execute at a rate beyond the threshold of perception. To produce a real-time performance we must slow this process to a desired temporal metric.

In summary, computational articulation of temporal structure involves specification of the ordering and spacing of events, in turn demanding support for scheduled activity and measured durations respectively.

Event Spacing: Measured Durations In order to allow specific temporal intervals to elapse between selected instructions, we need to provide a metric of time to measure against. The computational audiovisual domain suggests at least three sources for this metric: system time (based upon CPU hardware), frame-time (counting image buffer swapping events in the display system), and audio sample-time (based upon the sample-rate clock in the audio driver). These clocks are independent: audio clocks may drift slightly compared to system timers (Bencina, 2003), and frame-rate may vary adaptively with windowing system resources. In addition we may need user-space metrics within a composition, such as rhythmic tempo, as abstract temporal structures within and against which aesthetic form may occur. A user-space metric may be independent or synchronize with an existing clock.

The *time* module answers these demands by providing a generalized accumulator as a *clock* type, making no assumptions as to the semantics of the clock units (they could be nanoseconds, seconds, frames, samples, beats, mouse-clicks etc.) These accumulators can be polled for current value using the *now()* method, and manually incremented using the *advance(n)* method. To support synchronous relationships, clocks can be nested such that one becomes the driving parent of another: whenever the parent advances, so does the child. A relative offset and dynamically variable rate characterizes the child-parent relation.

In addition we implement a special clock available through *time.system* using nanosecond units and offering high-precision access to the CPU clock. This system clock supports two alternatives to manual increment: the *update()* method polls the system timer and advances all child clocks accordingly (suitable for event-driven hosts), while the *execute()* method hands over control of the application to the system timer which makes low-level system sleep calls to control execution in real-time.

Operation	Significance	Lua constructs
Memory	Duration, extended present.	Lexically scoped variables and garbage collection.
Control flow	Breaks homeomorphism between instruction series initiated and state space executed: conditions and branches.	If/then, do/while etc., but also higher-order functions and run-time compilation.
Concurrency	Simultaneity, parallelism, semi-independence.	Coroutines.
Events	Indeterminacy, event-driven execution.	Global callbacks from a host application or C library and file/stream handling.

Table 2: General classes of the computational articulation of temporal structure.

Event Ordering: Scheduled Activities Explicit articulation of temporal structure in computation must be built upon operations that divert the implicit, ephemeral causality between instructions. We identify four general classes of such operations, summarized in table 2.

Based upon this analysis, we chose a scheduling model that makes particular use of Lua's inherent support for concurrency and events. Specifically, we extend Lua coroutines[3] by inserting conditional scheduling within the yield-resume mechanism, based upon either the clocks described above or arbitrary event notifications. The body of a coroutine can encapsulate any valid Lua code and thus any combination of memory and control flow instructions, completing coverage of the operation classes identified.

3.2 Implementation: Coroutine Scheduling

A coroutine can be created under control of the *time* module using the *go* function, which takes a function and an optional list of arguments. Any active coroutine can yield its execution using the *wait* function, whose argument is either a numeric duration (relative to a specified or default clock), or an arbitrary Lua value as an event token. In either case, this call suspends the coroutine and places it under the ownership of a condition within the *time* module scheduler.

An event notification may be triggered using the *event* function, whose first argument is the Lua value event token in question (see script 1). Event tokens can be tables, functions, strings or any other valid Lua type except numbers. Any coroutines pending

[3] Coroutines, originally introduced by Conway in the early 1960s, are subroutines that act as master programs (Conway, 1963). A coroutine in Lua is a deterministically schedulable parallel virtual machine, constructed from a function defined in Lua code. A coroutine has its own stack, locally scoped variables and instruction pointer (which resumes from the same code point at which it last yielded) but shares global variables with other coroutines. Lua coroutines use a flexible asynchronous yield/resmue interface and are first-class objects within Lua code.

```
function clickwatcher()
   while true do        -- infinite loop
      local result = wait('click')
      print('a click occurred:', result)
   end
end

go(clickwatcher) -- launch a coroutine

event('click', 'left')
event('click', 'right')
```

Script 1: Using *go*, *wait* and *event* for event-based conditions. The *clickwatcher* function becomes the body of a coroutine when launched using *go*, resuming upon a ‘click’ event token notification via *event*. The additional arguments ‘left’ and ‘right’ are passed as return values to the *wait* call, and populate the *result* variable, for event-handling purposes.

```
function clockprinter(name, period)
   while true do
      print(name)
      wait(period)
   end
end

go(clockprinter, 'tock', 4)
go(clockprinter, 'tick', 1)
```

Script 2: Example using the default clock to create synchronous yet polymetric messages: printing four ‘ticks’ for every ‘tock’.

upon the specified event token will be resumed in first-in, first-out order. Event-driven coroutines are means to specify procedural behavior in response to indeterminate asynchronous input. A public C function is also provided to support notified events from a host application.

Coroutines that wait upon a clock duration are placed in a schedule list belonging to the clock, indexed by its current value plus the specified duration. When the clock advances to this target time, the coroutine is removed from the schedule and resumed as normal (see script 2). As a convenience, the *go* function also takes an optional first argument prior to the function body to specify a duration to wait for before launching the coroutine.

Coroutines are continuations: they are objects that model everything that remains to be done at certain points in a functional structure. The execution of a composer’s script becomes a real-time variation of the *continuation-based enactment* design pattern (Manolescu, 2002). The composer can use other module functions within coroutines to articulate many different audiovisual structures across time, whether synchronous or asynchronous, determinate or indeterminate.

3.3 Concurrency: Future Directions

Readers might note the absence of multithreading in the discussion of concurrency above. A Lua interpreter itself cannot be safely used across multiple operating system threads (a design decision of the Lua authors). [4] In contrast to coroutines, the scheduling of operating system threads is predominantly outside the scope of the executing program, introducing indeterminism in the timing of instructions between threads that prevents micro-temporal interdependencies. Safely sharing data structures between threads demands nontrivial solutions that are not conducive to a transparent programming interface (Lee, 2006). The call for tightly interleaved interaction between sonic and visual processes is very difficult to maintain with indeterminately scheduled threading. The authors therefore follow Lua by avoiding multithreading within the LuaAV interpreter.

This is not to say that threading is undesirable, since it may offer significant performance gains (particularly for multi-core machines). Finding a suitable means to make use of multithreading, while keeping the interface free of implementation details, is a pressing issue for LuaAV. Spreading computation across multiple threads is a problem of determining dependencies in processing graphs and dispatching batches of parallel computation. For example, parallelizing intensive sonic, spatial and geometric calculations in C/C++ functional tasks is viable if the return values are not time-critical.

A related need, in order to avoid temporal glitches in real-time output, is the graceful management of slow, blocking or indeterminately timed calls (such as file IO) within the main process. Libraries for Lua already exist that can place indeterminate calls into distinct operating system threads and return data once complete (HelperThreads (Guerra, 2005) for C calls, Lanes (Asko Kauppi, 2007) for Lua processes). It may be viable to provide similar support within the existing semantics of the *time* module, such as yielding a coroutine until a sub-thread process has completed.

4 Spatial and Visual Structure

In this section we describe the development of Lua modules for spatial and visual composition, which are founded on the notions put forth in the introduction. The modules deal with fundamental elements of spatial relationships and image composition. Spatial relationships are handled by *vec* and *space* while image composition and manipulation functionality is contained in *opengl* and *glo* modules.

4.1 Foundations

Within current computer architecture, 3D graphics operations are handled by specialized stream processors, known as graphics processing units (GPUs)[5]. The GPU takes

[4] Lua can be modified to be thread-safe by installing locks however there are performance and portability prices to pay.

[5] Throughout this document, the term GPU will be used to refer to the actual hardware in the computer as well as the system-level driver controlling the operation of the hardware

spatial information as input and projects it into image space for rasterization and display. In addition, modern GPUs support the compilation and execution of micro programs called *shaders* in order to support user customization of geometry and data processing routines. Within this system, we can identify four basic notions in the movement from spatial form to visual output:

- Position and orientation
- Distance and proximity
- Projection of space into image
- Compositing of images

Given a set of objects in space, each object will have a position and orientation. These properties describe relationships of the objects to the space itself. Between points and their orientations we can derive notions of distance and relative orientation as well as various types of proximity. Proximity in its most basic form is equivalent in meaning to distance, however the notion of proximity can also encompass spatial distribution. These notions form the basic elements of spatial composition.

Once on the GPU, spatial form is projected into image space through a virtual viewport. This viewport may contain additional fragments from previous frames or alternate views of the same space and potentially even elements from entirely different spaces.

4.2 Current Implementation

Quaternions and Vectors The *vec* module contains both 3D vector (*vec.Vec3*) and quaternion (*vec.Quat*) classes for composing position, movement, and orientation in 3D space. *Vec3* serves to simplify calculating 3D quantities and covers math operations for moving objects in space while *Quat* provides orientation and local coordinate frame information. In particular, quaternions can be used for calculating orientations without some of the computational drawbacks (Eberly, 2006) and degeneracies (Hanson, 2006) of other orientation representations. In terms of spatial composition, quaternions are eminently suited for designing paths through space as is required for extruding surfaces and calculating complex camera movements.

Euclidean Space and Point Relationships The *space* module supports both the basic notion of proximity as distance and the formulation correlating distance with spatial distribution. The basic notion of proximity is implemented using the Approximate Nearest Neighbors (ANN) (Mount and Arya, 2006) library's kd-tree functionality and can be accessed using the *within* and *nearest* functions. *within* takes a point and a scalar value indicating the squared search radius while *nearest* takes a point and a number of nearest points to return. Both functions return an array of points and their respective squared distances from the query point.

Distribution correlated proximity is implemented using the 3D Delaunay triangulations package from the Computational Geometry and Algorithms Library (CGAL) (Hert et al., 2007) and derived Voronoi diagram classes. O'Rourke (2005) describes Voronoi diagrams – and implicitly Delaunay triangulations due to their duality – as

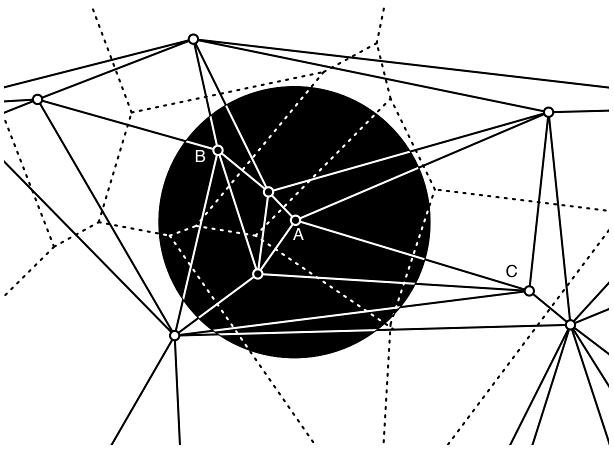

Fig. 1: An illustration of the correlation between distance and spatial distribution in a Delaunay triangulation (solid) with Voronoi diagram (dashed). The black disk is centered on A, which is connected to C but not B due to intermediate points along the direction from A to B despite B being significantly closer to A than C.

"[i]n a sense ... record[ing] everything one would ever want to know about proximity of a set of points (or more general objects)," however with the increase in encoded information comes a parallel increase in upfront computational cost and thus a lower frequency at which such structures can be calculated, especially within the constraints of a real-time system.

In relation to spatial composition, the Delaunay and Voronoi duality describe both a kind of rhizomatic latticework and an adaptive cellular tiling of the space occupied by the latticework (Figure 1). The latticework of the Delaunay triangulation is tetrahedral (in 3D) and also defines a boundary (convex hull) to the point set. The Voronoi diagram in comparison defines a set of adjacent polyhedral cells with each enclosing a single Delaunay point. Consequently, Voronoi diagrams can be used to drive spatial algorithms such as growth processes and aid in quantizing and processing volumetric fields.

Graphics Interface The graphics modules in LuaAV support both low-level control of the GPU through OpenGL[6] as well as higher-level userdata objects that distill complex sections of the OpenGL specification into a more appropriate compositional interface. The low-level OpenGL bindings are based on LuaGL[7] and provide a one-to-one mapping of the OpenGL API into Lua contained in the *opengl* module. The higher level objects are contained in a module called *glo* (GL Objects), which contains the classes *Shader*, *Texture*, and *Slab*.

Shaders and slabs load and compile code written in GLSL (OpenGL Shading Language) from file. The shader files are executable Lua code, taking advantage of Lua

[6] see http://www.opengl.org
[7] see http://luagl.wikidot.com

```
tex = glo.Texture(512, 512)
tex:beginCapture()
gl.Color(0, 1, 0, 1)
gl.Begin('TRIANGLES')
   gl.Vertex(-1, 0, 0)
   gl.Vertex(0, 1, 0)
   gl.Vertex(1, 0, 0)
gl.End()
tex:endCapture()
```

Script 3: Sample script that draws a green triangle to a texture 512 x 512 pixels in size.

as a configuration language, and bear the extension *".shl"*. Since the shaders are valid Lua statements, arbitrary code can be executed in the shader file on load. For example, a depth of field shader using a hard-coded (const float array) Poisson distribution as a sampling function in the fragment shader could generate the sample points procedurally on file load in place of listing a series of numbers.

'Slab' is the GPGPU[8] term for dispatching a textured quad to the GPU in order to perform calculations with the fragment processor (Harris, 2003). Typically each element in the texture is passed uninterpolated through the fragment processor and the result written into an equal-sized texture. The slab implementation in *glo* uses an FBO (Frame Buffer Object) backend and can automatically adapt output texture size if enabled. Slabs can be chained and used in feedback loops with other slabs or textures.

Textures in *glo* support both image loading from file and render-to-texture. In render-to-texture mode, the texture is bound to an FBO and any successive draw operations are drawn to the texture instead of the screen (see script 3). Textures combined with slabs allow for development of complex image-space effects. Slabs encapsulate image-processing chains on the GPU for operation on textures, which can then be composited together in the image frame by blending together textured geometry. As a simple example, a basic montage effect can be constructed by capturing a scene to texture from two different vantage points and compositing them together according to some function of space and color.

Morphogenetic Space: Future Directions While *space* currently handles purely Euclidean space and geometric primitives, a comprehensive computational spatial system should further include other conceptions of space and spatial entities in order to supply a broader range of compositional tools at various levels of spatial abstraction. The power of such an approach is apparent in software such as TopMod (Akleman et al., 2007) whose manifold preserving topological mesh editing operations enable a user to construct complex structures with holes and handles.

A thorough discussion of spatial abstraction as it relates to generative and emergent form has been given by DeLanda in his treatment of Deleuze's notions of virtuality,

[8] General Purpose computation on Graphics Processing Units (GPGPU), see http://www.gpgpu.org

wherein DeLanda puts forth what could almost be described as a group theoretical framework for spatial composition (DeLanda, 2005). This framework describes a morphogenetic view of space where processes of symmetry-breaking transformations move progressively from intensive to extensive embodiment, resulting in complex spatial and temporal dependencies in the generation of form.

5 Sonic Structure

In this section we describe the development of Lua extension modules to support the articulation of sonic structure according to the strategies outlined in the introduction. The result is the *audio* module for real-time low-level IO, and the *vessel* module providing abstractions of signal processing algorithms under the control of a lazy dynamic scheduler, both of which interface with the *time* module described in section 3.

5.1 Foundations

In considering the immanent notions of the sonic within computational audiovisual composition, we first encounter the low-level representation of acoustic phenomena in terms of series of discrete samples at a fixed sample-rate, as the form by which a program may render audio signals. We also note that the sheer volume of data being handled incurs a hard real-time efficiency constraint.

This representation is agnostic as to how sample-data may be generated or manipulated and thus gives no indicators as to an appropriate higher-level concept for the specification and articulation of sonic structure. We need a paradigm of signal processing which satisfies our low-level design goal, and may map well into the computational context. A number of approaches to model and control signal processing exist, however one of the most familiar is the *unit generator* (UG) paradigm, which encapsulates signal processing functions within stateful modular objects that can be freely interconnected into directed graphs. The homogeneity of the connection type and modularity of interconnection meets our demand for generality, while the discrete nature of graph structure and the symbolic nature of function encapsulation reflects well the computational domain. The authors therefore chose the UG concept as a viable signal processing model for sonic articulation [9].

5.2 Implementation

Real-Time IO Rendering sonic output (and capturing input) in real-time involves interfacing with low-level operating system audio services, such as DirectX for Windows or CoreAudio for OSX. Libraries exist offering cross-platform abstractions of these services. We embed one such library, PortAudio (Bencina and Burk, 2001), within the *audio* module to introduce real-time audio in Lua.

Low-level real-time IO must satisfy a hard constraint: synthesis must precede playback. In practical terms this means that CPU bound synthesis of sample data must run

[9] Nevertheless it is borne in mind that other and possibly superior solutions may yet be found.

in constrained windows of time specified by the buffering placed between synthesis and hardware (usually blocks of between 16 and 2048 samples per channel). Lower IO latencies (reduced buffering) can be achieved using secondary operating system threads for audio processing (Bencina, 2003) however as noted previously, such multithreading is neither supported by standard Lua nor is it conducive to computational audiovisual composition with low-level control. The *audio* module thus uses a blocking interface to read and write samples in the audio hardware. The capability to work with audio sample data alongside graphics calls such as OpenGL within a predictable, determinate shared memory context is considered one of the key features of LuaAV.

Calls to exchange sample data with the audio hardware increment an internal sample-counter, which takes the form of a clock as defined in the *time* module. Lua coroutines may be scheduled to this clock in terms of real-time sample durations, or put another way, articulation of form via Lua code can be sample accurate.

Signal Processing Due to the sheer volume of data being processed and real-time demands, the UG signal processing algorithms should generally be executed in optimized machine code rather than interpreted Lua. A viable solution is to bind existing C/C++ signal processing libraries to Lua so long as such libraries offer atomic access to UGs and make minimal assumptions as to how they may be used. In addition, the domain of computational audiovisual composition strongly suggests evolving dynamic relationships between sonic processes and other objects at arbitrary points of real-time (i.e. not quantized to system-dependent block-rates), adding several further requirements:

- dynamic graph determination
- efficiency in setup/change/removal
- variable block-size processing (for sample-accurate graph changes)

By making use of the Synz C++ library (Putnam, 2007), which meets these requirements, a set of UGs have been added to the *vessel* module, including oscillators, filters, delays, spatializers and more generic math functions. The UGs and their data buffers are implemented using up-front allocated free-list memory pools, to minimize the cost of memory management in a real-time process (Dannenberg and Bencina, 2005). Preallocated memory is recycled as the program executes, and pools grow if needed, utilizing a memory allocator optimized for real-time (Lea, 2000).

The UGs are presented within Lua in the form of mutable *userdata* objects, encapsulating C++ class instances. Userdata are first-class objects within the language and can be manipulated just like any other Lua type. The programming interface offers multiple means to compose UGs into graphs. Constructor functions create new UG instances, and may take arguments to specify input nodes. An instance's inputs are exposed for reference and re-assignment via member functions. Furthermore, all UGs share overloaded operators to construct graphs from elementary mathematical relations (+, -, $*$, /, ^, %). Finally, the outputs of multiple UGs can be merged using a special mixing bus type (see script 4). The combination of UGs as first-class Lua values and multiple modes of graph composition follow the design strategy of computational generality.

```
modulator = Sine()
modulator:frequency(8) -- 8Hz
carrier = Sine(440 + modulator * 10) -- Frequency Modulation
Out:add( carrier )
```

Script 4: Constructing UG graphs. Line 2 directly sets the frequency input of the UG created in line 1. Line 3 defines the frequency of a second UG (via constructor argument) to a graph composed using operator overloading. Line 4 attaches this second UG to the global output bus.

Scheduling: Dynamic and Lazy UG graphs are deterministically directed since a node must have input data before it can produce output. This is a formulation of the producer-consumer problem, and both push (leaf to root) and pull (root to leaf) strategies may be used to traverse the graph. Static scheduling traverses the graph once before executing, while dynamic scheduling constantly re-traverses the graph during execution. Supporting dynamic graph changes at sample-accurate times in the midst of a buffered audio process is nontrivial.

The *vessel* module uses lazy evaluation to achieve this. Any state change to a UG (any Lua calls to modify its graph connections or internal state) triggers a traversal of the sub-graph upstream of the UG in question. Such traversals process each node from its current clock-time to the state change clock-time, resulting in just-in-time sample-accurate graph dynamics. Attempting to read data from the global audio output bus, such as copying to the audio hardware for real-time playback, triggers a graph traversal from the root to determine any remaining samples and advance the associated clock to a new time-stamp.

Since all graph state changes are triggered from within Lua code, the lazy dynamic scheduler fully supports the sample accurate interleaving of synthesis and control specification. [10] The *vessel* module turns out to be ideal for the exploration of algorithmic and generative approaches to the organization of microsound (Roads, 2001).

Signal Processing: Future Directions A general limitation of the UG paradigm has already surfaced in practice: elements of processing become black boxes to the user. An example scenario where this may interrupt creativity is the inability to insert signal processing within a feedback delay path shorter than the graph block size. A possible solution under investigation is the dynamic compiling and loading of arbitrarily complex signal processing routines at runtime. Compiling a short C/C++ function can happen in a matter of milliseconds on current systems, which may be an acceptable latency for real-time code generation of optimized processing routines.

Signal processing in general is not specific to the sonic domain, and although the UG is inherited from electronic music, the concept may be more widely applicable to N-dimensional data streams. Alternative signal processing paradigms may also be applicable to spatial and more abstract domains. Exploring alternative approaches to synthesis specification within Lua is currently being investigated; a strong candidate with clear potential beyond the sonic may be Honing's Generalized Time Functions

[10] Neologised by the authors of ChucK as “strongly timed” (Wang and Cook, 2004)

(N-dimensional manifolds including duration and current time as parameters (Honing, 1995)).

6 Conclusion

Our enthusiasm for computational audiovisual composition stems from our belief in the expressive capabilities of highly dynamic, functional and computational explorations and articulations of audiovisual experience, and from our excitement regarding the opportunity to pursue research investigations in this domain as an artistic practice in itself.

A noteworthy observation emerging from our work is the good fit between the coroutine construct and the computational articulation of time. Similar yet purely musical approaches to concurrent composition structure are evident in related languages: *Tasks* and *Routines* in SuperCollider (McCartney, 2002) and *Shreds* in ChucK (Wang and Cook, 2004). Rather than introducing new types however, we leverage an existing programming construct, and by avoiding preconceived notions such as keyframe, beat, measure etc. we provide more generic and interoperable functionality.

In summary, we have described the development of LuaAV, a framework to support a wide range of audiovisual composition practices, taking advantage of the language's inherent flexibility and extending it with modules responding to the immanent conditions of the computational audiovisual domain. Due to the design strategies employed, we hope it provides solid grounds for future research.

References

Ergun Akleman, Jianer Chen, and Vinod Srinivasan. *Topological Mesh Modeling*. Texas A & M University, 2007.

Apple Computer – Cocoa. http://developer.apple.com/cocoa/, 2008.

Asko Kauppi. Lua lanes. http://luaforge.net/projects/lanes/, 2007.

Ross Bencina. PortAudio and media synchronization. *Proceedings of the Australasian Computer Music Conference*, pages 13–20, 2003.

Ross Bencina and Phil Burk. PortAudio - an open source cross platform audio API. *Proceedings of the International Computer Music Conference, Havana*, pages 263–266, 2001.

Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963. ISSN 0001-0782.

Roger B. Dannenberg and Ross Bencina. Design patterns for real-time computer music systems. http://www.cs.cmu.edu/ rbd/doc/icmc2005workshop/real-time-systems-concepts-design-patterns.pdf, 2005.

Manuel DeLanda. *Intensive Science and Virtual Philosophy*. Continuum International Publishing Group, New York, NY, USA, 2005.

David Eberly. Rotation representations and performance issues. Technical report, Geometric Tools Inc., 2006.

Dave Griffiths. Fluxus. http://www.pawfal.org/Software/fluxus/, 2007.

Javier Guerra. Helper threads: Building blocks for non-blocking libraries. http://helper-threads.luaforge.net/, 2005.

Andrew J. Hanson. *Visualizing Quaternions*. Morgan Kaufmann, 2006.

Mark Harris. *Real-Time Cloud Simulation and Rendering*. PhD thesis, University of North Carolina at Chapel Hill, 2003.

Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Michael Seel. An adaptable and extensible geometry kernel. *Comput. Geom. Theory Appl.*, 38(1-2): 16–36, 2007. ISSN 0925-7721.

Henkjan Honing. The vibrato problem: Comparing two solutions. *Computer Music Journal*, 19(3):32–49, 1995.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X.

Mark Kilgard. GLUT - The OpenGL Utility Toolkit. http://www.opengl.org/resources/libraries/glut/, 2008.

Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 2000.

Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

Dragos A. Manolescu. Workflow enactment with continuation and future objects. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 40–51, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1.

James McCartney. Rethinking the computer music language: Supercollider. *Comput. Music J.*, 26(4):61–68, 2002. ISSN 0148-9267.

David M. Mount and Sunil Arya. Ann: A library for approximate nearest neighbor searching. http://www.cs.umd.edu/ mount/ANN/, 2006.

Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2005.

Miller Puckette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

Lance Putnam. Synz. http://www.uweb.ucsb.edu/ ljputnam/synz.html, 2007.

Curtis Roads. *Microsound*. MIT Press, Cambridge, MA, USA, 2001.

Dave Thomas and Brian M. Barry. Model driven development: the case for domain oriented programming. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 2–7, New York, NY, USA, 2003. ACM. ISBN 1-58113-751-6.

Graham Wakefield and Wesley Smith. Using lua for audiovisual composition. In *Proceedings of the 2007 International Computer Music Conference*. International Computer Music Association, 2007.

Ge Wang and Perry 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*, pages 812–815, New York, NY, USA, 2004. ACM. ISBN 1-58113-893-8.