LuaAV: Computational audiovisual composition using Lua

Wesley Smith
Graham Wakefield

Media Arts and Technology,
UC Santa Barbara
Computational AV Composition

• Composition as an arrangement of algorithm and information flow
• Tight integration of time, space, sound, and computation
  – Ability to describe both audio and graphics in an interleaved fashion
• Script as composition
  – Both representation and actualization
Media Arts and Technology

Allosphere
• 2 3-story (5m / 16.5ft radius) hemispheres
• Near-anechoic chamber
• Center for research into audiovisualization and audiovisual performance
pre-LuaAV

- Audiovisual extensions for Max/MSP/Jitter
  - jit.gl.lua
    - 3D graphics (OpenGL), Jitter bindings
  - lua~
    - Sample-accurate synthesis and control
- Master’s theses:
  - Smith
    - Abelian: A Visual and Spatial Platform for Computational Audiovisual Performance
  - Wakefield
    - Vessel: A Platform for Computer Music Composition, Interleaving Sample-Accurate Synthesis and Control
LuaAV

• Application design goals
  – Vehicle for computational AV composition research
    • Quick to prototype ideas
    • Adaptable to a wide range of hardware configurations
    • Capable of both real-time and offline processing with minimal changes
  – Composition and performance environment
    • Well-designed workflow for diverse use cases
    • High-performance and reliable
    • High-level interface that doesn’t sacrifice flexibility for abstractness
  – Support multimedia distribution
    • Targeted for major platforms (OSX, Windows, Linux)
    • Simple packaging system for compositions (scripts + data)
Application Design

• Plan to provide most functionality as standard Lua modules
  – Minimizes dependencies on LuaAV platform

• In practice plan was difficult to execute
  – Tricky to separate dependencies
    • Timing shared between windowing, audio, MIDI/OSC
    • Shared data types for time, vector, Matrix, etc.
Application Design

LuaAV

Internal Scripts
- LuaAV.lua
- preferences.lua

Internal Modules
- luaav
- window
- ifs
- audio
- rings

User Environment

External Scripts

External Modules
- opengl
- muro
- vsl
- glv
Application Design

• Application logic
  – Utilizes existing Lua modules for OS services where possible
    • Increases portability and maintainability
    • Currently using LFS and Rings
      – All of the script management logic coded with Rings
      – Rocks for module management in the works
  – Gaps in functionality covered by the LuaAV module
    • File watching, console interaction, …
  – Separate internal modules and userdata for audio, windowing, and high-precision timing
Language Property Mappings

• “Extensible extension”
  – Coroutines for temporal processes driven by C/C++
    clock schedulers
  – Userdata table environments for extensible
    instances
  – Modules natural for application extension
  – Tables used as parameters where possible for
    portability between modules (but not applicable in
    all cases)
    • e.g. OpenGL <-> Vector <-> Color
Language Property Mappings

- Tricky issues for lack of ‘standards’
  - C++ userdata
    - type identification & safety, esp. between modules
  - Standard accelerated (or at least C/C++ fast) math/vector lib where tables aren't fast enough
  - Method of sharing buffers
    - e.g. float[64], etc., really useful for multimedia modules
  - State serialization (save & resume state snapshots would be nice)
  - Threading difficulties
    - yet in multimedia often desirable; in audio, essential
      - Some Lanes / LuaTask-esque solutions
      - Multi-threaded Rings could also be great
LuaAV Bindings

- **Filewatching**
  - Sets up OS notifications for when files are modified
  - Eliminates the need to build a script editor into LuaAV

- **Audio Driver**
  - Abstraction of audio device hardware (based on PortAudio)
  - Creates & opens an audio stream on a chosen device, with configurable sample-rate, channels etc.
LuaAV Bindings

• Timing
  – Global sub-microsecond clock
  – Per-script schedulers (priority-queue of Tasks):
    • Tasks manage temporal processes in LuaAV
      – e.g. Audio.tick(), Window.draw()…
    • Lua interface for coroutines as Tasks:
      – coro = go([delay], func, args…)
      – now()
      – wait(dur)
      – coro:abort()
    • Extension API in LuaAV.h for user-defined Tasks in modules
      – E.g. MIDI.poll(), HID.poll()…
    • Reliable scheduling between script, modules & application
    • Scheduler adapts to load by adjusting slop
    • Simple to convert scheduler to non-real-time mode

function hi(msg)
  while now() < 4 do
    print(“hello”, msg)
    wait(1)
  end
end

go(hi, “world”)
LuaAV Bindings

- **Windowing**
  - Provides highly-customizable OpenGL windowing
    - Prototype constructor
    - Attributes for fps, fullscreen, borderless, active stereo, cursor style, etc.
  - Supports multiple windows, monitors
    - All windows share a root OpenGL context for resource sharing (like Textures, VBOs, displaylists)
  - UI callbacks for mouse and keyboard
  - Tied to the script that creates it
    - Automatically unloads scripts when closed

```lua
local win = Window{
    title = ctx,
    origin = {0, 0},
    dim = {320, 240},
    fps = 33,
}

struct LuaMethod {
    public:
        enum MethodType {
            METHOD = 0,
            GET,
            SET,
            GETSET,
        };

        const char *name;
        lua_CFunction func;
        MethodType type;
    }
```
Muro

Real-time graphics library for audiovisual composition

- Built as a shared library with Lua bindings
  - Linked against a muro.so module and LuaAV
- Provides a modular set of data processing, GPU programming, and 3D rendering objects
  - Support for imaging and video (cameras, and files)
  - Support for advanced OpenGL capabilities
    - Shaders, slabs, meshes, PBOs, VBOs, …
  - Makes use of:
    - Freeimage
      - Wide range of image formats (load and save)
    - OpenEXR
      - High-dynamic range
    - OpenCV
      - Computer vision
    - Open Frameworks
      - Video files, cameras, OpenCV extensions
Matrix and Texture serve as basic units of CPU and GPU data
  
  - 1-, 2-, and 3-D data
  - uchar, int, uint, float, and double types
  - 1-, 2-, 3- (RGB), and 4-plane (RGBA) formats

Filters for Matrix processing

Slabs for Texture processing
  
  - Abstracts Shader and FBO
  - Allows multiple render targets

Can convert data between Matrix<->Texture
  
  - Fastpath conversion via PBOs, memory mapping

Additionally, Matrix and Texture data can become geometry data via Mesh and VBOs
Muro

- **Shader**
  - Supports GLSL and Cg
  - Shaders loaded via a custom file format (.shl)
    - .shl files are Lua scripts
      - Makes use of Lua as a data description language
    - Consists of shader code and metadata:
      - Shader name, description
      - Parameter names, descriptions, defaults
      - Program code and defaults

```plaintext
Shader{
  name = "cc.scalebias.shl",
  description = "[Scale-bias Color Correction Filter]",
  language = "GLSL",
  parameters = {
    Parameter{
      {1., 1., 1., 1.},
      name = "scale", type = "vec4",
      description = "Scale factor",
    },
  
  Program{
    type = "vertex",
    varying vec2 texcoord0;
    void main() {
      gl_Position = ftransform();
      texcoord0 = vec2(gl_TextureMatrix[0]*gl_MultiTexCoord0);
    }
  }
}
```
VSL

Real-time audio library for scheduling dynamic DSP graphs.

– Problem: DSP in high-priority audio thread for low latency IO
  • avoid unbounded memory allocations, locks, etc.
  • hide these issues in Lua binding

– Solution - two graphs, RPC-like synchronization:
  • in Lua thread: userdata (Vertex::Proxy *)
    – Allocations, user-controlled state changes
  • in audio thread: Vertex & Edge structs
    – Optimized algorithms, memory pools
Lua interface:

- **Audio VM**
  - *Broker* between Lua & audio thread
    - using wait-free FIFOs
  - *Factory* of DSP primitives (‘Ugens’)
    - allocating struct for audio thread & returning *proxy* object for Lua

- **Ugen** *(Proxy)*
  - Methods to connect and form directed graphs
    - sending time-stamped messages to audio thread structs via VM FIFO

```
local mod = vm.Saw(0.1) * 400
local fm = vm.Sine(mod + 500)
vm:add(fm)
```
Audio Thread Loop:
- For each VM:
  - Read FIFO; update graph with incoming Ugen state changes
    - Trailing latency: audio block size + scheduler slop
  - Flatten into adjacency list
  - Execute list
    - Buffer use minimized: lazy allocation + eager recycling.
    - Helps avoid cache misses
  - Expired Ugens returned to main thread to free memory.

ICMC Proceedings, September 2007

Computational Audiovisual Composition Using Lua

Wesley Smith and Graham Wakefield
Media Arts and Technology, University of California, Santa Barbara
Santa Barbara, CA 93106, USA
wsmith@wakefield@mac.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 modular approach of development going back to invariant properties of the domain.

Keywords. Audiovisual, Composition, Real-Time Multimedia, Lua, Scripting Language, Functional Programming, Domain Oriented Languages, Constraints, 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 modular approach of development going back to invariant 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 collaboration. 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 algorithms.
- Ease of interaction with symbolic elements from other disciplines (examples might include mathematics, physics, systems biology, semantics...)

Our development is targeted towards two specific goals. Firstly, we aim for a tight integration of time, space, sound and composition, motivated by the artistic possibilities engendered. Secondly, we begin from analyses of the invariant 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 in novel aesthetic activities develop.
In Action!
Further Info

- http://lua-av.mat.ucsb.edu