Proceedings of the 17th Linux Audio Conference (LAC-19), CCRMA, Stanford University, USA, March 23–26, 2019

BIPSCRIPT: A DOMAIN-SPECIFIC SCRIPTING LANGUAGE FOR INTERACTIVE MUSIC

John Hammen
bipscript.org
Berkeley, California, USA
jhammen@j2page.com

ABSTRACT

Bipscript is a domain-specific scripting language designed to make it easier to create interactive music. The base language is the Squirrel scripting language which has been complemented with a standard class library containing audio-specific domain objects. This API provides methods for creating, scheduling and handling events of various types including MIDI, OSC and extracted features of audio streams. A single-threaded programming model with asynchronous event handling is familiar to web developers but atypical in music DSLs. Scripts are executed by a command line interpreter with tight integration to the audio system.

1. INTRODUCTION AND DESIGN GOALS

The Bipscript project began as an attempt to implement a musical "bot" application, with the base functionality of existing auto-accompaniment software augmented with a high degree of interactive functionality. The goal was software that would output appropriate MIDI sequences in real-time based on external inputs, most notably data from human performers. The emphasis was on tempo-driven music with tight integration to a local transport.

Design goals did not include specialization on any particular style of music, nor any specific assumptions on how external inputs would affect generation of MIDI sequences, leaving these decisions to the configuration of a particular piece.

As the options for configuration grew it became clear the easiest way to express the behavior of a particular musical part would be via an imperative language with the abilities to directly receive relevant input from external sources, and use this information algorithmically to sequence MIDI notes.

2. FEATURES

2.1. Squirrel

The scripting language itself is the Squirrel language [1]. From the Squirrel website:

"Squirrel is a high level imperative, object-oriented programming language, designed to be a light-weight scripting language that fits in the size, memory bandwidth, and real-time requirements of applications like video games."

These attributes and the associated predictability in run-time behavior make Squirrel an ideal language also for the real-time demands of audio applications.

Bipscript builds on top of Squirrel by adding a class library API containing audio domain-specific classes and a custom transport-aware interpreter that allows for event handling.

2.2. Class Library

The Bipscript class library API features objects representing plug-ins, mixers, and system inputs and outputs of various types, predominately audio, MIDI and OSC [2]. These objects can be connected programmatically to create complex networks of the different protocols (see Figure 1)

Events of any applicable type can be scheduled to occur on any node in the network, in particular code can generate and output timed MIDI and OSC sequences. Event handlers can be registered to fire on particular events including features extracted from audio streams.

Additional classes allow the use of textual specification of musical score data in the scripts using ABC notation [3], Music Macro Language [4] or a MIDI tablature format based on common drum tablature.

```
local midiInput = Midi.Input("myinput")

local synth = Lv2.Plugin("urn:example:a-synth")
synth.connectMidi(midiInput)

local effect = Lv2.Plugin("urn:ardour:a-reverb")
effect.connect(synth)

local mainOutput = Audio.StereoOutput("main")
mainOutput.connect(effect)
```

Figure 1: Creating Connections.

2.3. Threading and Context Model

Scripts including event handlers run in a single execution thread with a single global context. Instructions in a script will be executed sequentially until the main body of the script completes. At this point any event handlers that were registered by the script will execute as needed in the same thread and scope with direct access to all variables defined in the main body of the script.

This programming model is analogous to traditional JavaScript development in a web browser where the main body of the script registers event handlers that are then executed in the same thread within the same page context. In both cases execution is single-threaded and non-blocking in favor of asynchronous event handling.

Although many objects such as plugins will participate in the audio client's process thread the script thread itself is separate from the process thread and not subject to its programming limitations on e.g. memory allocation and system I/O.

2.4. **Transport-Aware Interpreter**

The command-line interpreter is tightly integrated with the system transport, on Linux provided by the Jack Audio Connect Kit [5]. Scripts can request to be the transport master via the API but are not required to do so when there is an external master present to provide position information.

Whether or not the script itself is the transport master, the interpreter will act as a sequencer for any events scheduled by the script, playing them in time with the transport and reacting to any arbitrary external transport position changes including looping.

3. COMPARISON WITH OTHER PROJECTS

3.1. **Synthesis Languages**

There exist several DSLs for music creation with large and active communities, for instance Pure Data [6] and SuperCollider [7] among others. These environments differ from each other, for instance Supercollider is a traditional text-based programming language while Pure Data is a visual language.

There is a however a common emphasis on sound design with code libraries of elements representing oscillators, filters and other signal generation capabilities. Bipscript currently offers no such objects in its standard class library instead offering hosting capabilities for 3rd party sound generation and effects plugins.

The emphasis instead is on timed music which has led to an API built around handling events and just-in-time sequencing using data structures representing e.g. MIDI notes and mutable patterns (groups of notes). In contrast most music DSLs produce timed music at a lower level by alternating between immediate sound-generation instructions and some variation of a system "sleep" command.

3.2. **Other Open Source Projects**

Other comparisons can be drawn to some of the many projects arising from the community of open source audio software on Linux and elsewhere:

One of the most feature-rich open source audio applications is the Ardour DAW [8], which in recent versions has a large number of the C++ implementation classes exposed as Lua objects [9] giving a scripting environment incorporating much of Ardour's MIDI and DAW functionality. This differs from a more traditional script language-plus-interpreter environment in that scripts are executed as callbacks in application-specific contexts, each with their own scope and applicable model objects.

Another project using Lua is the Moony Lv2 plugins [10]. Taking advantage of Lua's real-time performance, scripts are run directly in the process thread and thus allow manipulation of e.g. MIDI messages as they pass through the plugin. but are bound by all the standard real-time limitations of running in the process thread.

LuaJack is a Lua binding library for Jack [11]. Scripts written in LuaJack and Bipscript have a visual similarity due to the similarity between Lua and Squirrel and the fact the LuaJack and Bipscript API are wrapping some of the same objects, e.g. system ports. However as a language binding LuaJack does not include an interpreter nor an object API beyond directly exposing the Jack client API.

4. IMPLEMENTATION

The command line executable that functions as an interpreter to execute scripts was written in C++ and runs on the Linux operating system with the Jack Audio Connection Kit as a run-time dependency.

The standard Squirrel implementation is intended to be embedded and was used as the basis of the command line interpreter.

The interpreter also acts as a standard audio client, opening and connecting system audio and MIDI input/output ports and hosting plugins as specified by the executing script.

Scripts are loaded and run in a dedicated execution thread separate from the application's audio process thread. Any event generated by the script is appended to an applicable lock-free queue that is consumed by the process thread.

Script objects that hold scheduled events will participate in the audio process thread to pull events from the queues and emit them at the appropriate position in a running transport with sample-level accuracy. Synchronization between the script and process threads allows the script execution to properly respond to arbitrary transport location changes.

A set of bindings was created for the class library, generated from a high level API description to interface the C++ implementations of the standard library classes and methods to the Squirrel engine via its stack-based API.

The object implementations make use of reliable third party code where possible via both embedded code and dynamically linked libraries. The current implementation makes uses of popular projects such as abcmidi, liblo and libsndfile.

5. USE CASES

Now existing in a basic implementation, Bipscript can be used for the following use cases:

5.1. **Dynamic Accompaniment**

An example "Robot Jazz Band" was built [12], showing the script implementation of 3 related bots (playing acoustic bass, piano and trap drum samples) that play a dynamic sequence based on rhythmic probabilities in a jazz "swing" pattern coupled with a given input chord progression. All players take an input parameter of "intensity" playing louder and busier vs. softer and sparser on a measure by measure basis. The main script connects an audio onset de-

tector to a system audio input and continuously updates this intensity variable for all players based on the number of onsets received.

This example shows a standard programming model for creating a script with this kind of interactive behavior:

- The main script instantiates plugins and needed audio and MIDI connections (see Figure 1) and schedules any static parts of the score
- Event handlers listen to human performers via e.g MIDI, OSC and/or audio features and update an internal state (see Figure 2)
- Scheduled methods read this state and use as input in calculating a short output sequence (e.g. a beat or bar at a time)

The result is a dynamic and reactive auto-accompaniment system written in a relatively few lines of code compared to general purpose programming or even other music DSLs. The simple example works as expected in practice but leaves open many paths for future development in creating more complex interactive scripts.

```
23 local controlPower = 0
24 controlInput.onControl(function(cc, pos) {
25   if(cc.controller() == 15) {
26     controlPower = cc.value() * 100 / 127
27   }
28 })
```

Figure 2: Sample Code from the Robot Jazz Band Demo.

5.2. Utility Scripts for Live Performers

In many traditional live musical projects there is no need for dynamically generated sequences, especially those where human performers are playing from a static score. In such an environment there is still a use for certain computer-aided functions such as triggering samples in time or adding a “click track” or other timed audio cues not heard by the audience. These functions can be built in custom scripts with relatively few lines of code.

5.3. Live Coding

Another prospective use is that of live coding. Those functions of the command line interpreter that allow for developer convenience may also be useful in a live coding situation, e.g. the ability of the script to be loaded and unloaded dynamically including while the transport is running. This use case is as of yet mostly unexplored

6. FUTURE WORK

With the completion of a basic proof-of-concept interpreter and class library the main focus of the project remains stabilizing the standard API and improving the basic tool implementations especially with an eye to reliability in live settings. To this end recent work has been done in the area of unit and functional testing for testing scripts as well as the interpreter itself.

Much of the API design going forward should be based on feedback from those who use these tools in a live production setting.

7. REFERENCES

[1] Demichelis, A. 2016. Squirrel – The Programming Language. http://www.squirrel-lang.org/

[2] Wright, M. 2002. Open sound control 1.0 specification.

[3] Walshaw, C. 2018. abc notation. http://abcnotation.com/

[4] Nakamura, S. 2015. A tiny MML parser. *Cubeatsystems.com* https://www.cubeatsystems.com/tinymml/

[5] Davis, P. 2003. Jack audio connection kit. http://jackaudio.org/

[6] Puckette, M. 1970. Pure Data: another integrated computer music environment.

[7] McCartney, J. 1996. SuperCollider. *supercollider.github.io*

[8] Davis, P. 2018. Ardour: the Digital Audio Workstation. http://www.ardour.org

[9] Manual.ardour.org 2018. The Ardour Manual. http://manual.ardour.org/lua-scripting/

[10] Portner, H. 2018. Moony - realtime Lua as programmable glue in LV2. http://open-music-kontrollers.ch/lv2/moony/

[11] Trettel, S. 2018. LuaJack Reference Manual. *Stetre.github.io*, https://stetre.github.io/luajack/doc/index.html

[12] Hammen, J. 2016. Robot Jazz Band Example. bipscript.org, http://www.bipscript.org/en/examples/robotjazz