THE JAMOMA AUDIO GRAPH LAYER

Timothy Place
74 Objects LLC,
Kansas City, Missouri, USA
tim@74objects.com

Trond Lossius
BEK - Bergen Center for Electronic Arts
Bergen, Norway
trond.lossius@bek.no

Nils Peters
McGill University, CIRMMT
Montreal, Quebec, Canada
nils.peters@mcgill.ca

ABSTRACT

Jamoma Audio Graph is a framework for creating graph structures in which unit generators are connected together to process dynamic multi-channel audio in real-time. These graph structures are particularly well-suited to spatial audio contexts demanding large numbers of audio channels, such as Higher Order Ambisonics, Wave Field Synthesis and microphone arrays for beamforming. This framework forms part of the Jamoma layered architecture for interactive systems, with current implementations of Jamoma Audio Graph targeting the Max/MSP, PureData, Ruby, and AudioUnit environments.

1. INTRODUCTION

Many frameworks, toolkits, and environments for real-time audio fuse the issues of creating unit generators with creating graph structures[1] that process audio through those unit generators. Alternatively, the Jamoma Platform implements a clear separation of concerns, structured in a layered architecture of several frameworks [1]. Six frameworks currently comprise the Jamoma Platform, providing a comprehensive infrastructure for creating computer music systems. These frameworks are: Jamoma Foundation, Jamoma Graphics, Jamoma Modular, Jamoma DSP, Jamoma Graph and Jamoma Audio Graph (see Figure 1).

Jamoma Foundation provides low-level support, base classes, and communication systems; Jamoma Graphics provides screen graphics; Jamoma Modular provides a structured approach to development and control of modules in the graphical media environment Max [2] and Jamoma DSP specializes the Foundation classes to provide a framework for creating a library of unit generators. Jamoma Graph networks Jamoma Foundation based objects into graph structures, providing a basic asynchronous processing model for objects (nodes) in the graph structure. Jamoma Audio Graph, the focus of this paper, is an open source C++ framework that extends and specializes the Jamoma Graph layer. It provides the ability to create and network Jamoma DSP objects into dynamic graph structures for synchronous audio processing[2].

1.1. Requirements

Through years of accumulated collective experience in a myriad of contexts for realtime multi-channel audio work, we found a number of practical needs were still unmet by readily available environments. We believe the following necessities are required not only to ease usability concerns in some environments, but also to open new avenues of creation, exchange, performance, and research in digital audio effects.

- Connections between objects must be capable of delivering multiple channels of audio
- Support for audio signals with multiple sample rates and vector sizes simultaneously within a graph
- Audio signal sample rate, vector size, and number of channels must be dynamic (able to be changed in realtime)
- Dynamic modification of an audio graph at runtime (Live coding/Live patching)
- Ability to transport representations of an audio graph between different programming languages
- Support for multi-threaded parallel audio processing
- Liberal licensing for both open source and commercial use
- Cross-platform

In order to best meet these requirements, Jamoma Audio Graph relies on a *pull* pattern (see Section 2.3) and is designed to be independent from the host environment's DSP scheduler. An additional design decision is the use of a "peer object model" which abstracts the implementation and execution of the graph from any particular environment or platform. This allows for the Jamoma Audio Graph layer to be readily implemented for any number of environments. To date the authors have created implementations for Max, PureData, Audio Units, and Ruby.

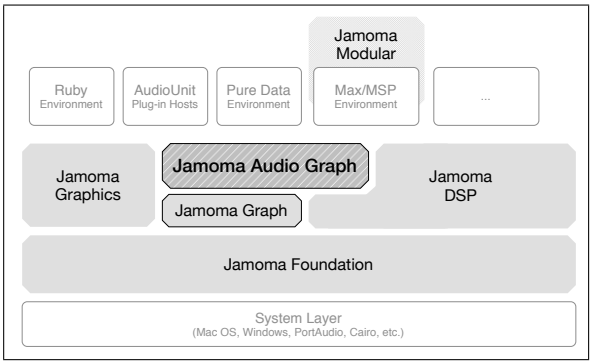


Figure 1: The Jamoma Platform as Layered Architecture.

2. BACKGROUND

2.1. Decoupling Graph and Unit Generator

The Jamoma DSP framework for creating unit generators does not define the way in which one must produce signal processing to-

[1] Wikipedia defines this by saying "a graph is an abstract representation of a set of objects where some pairs of the objects are connected by links". http://en.wikipedia.org/wiki/Graph_(mathematics)

[2] Licensing for all layers of the Jamoma Platform are provided under the terms of the GNU LGPL

pographies. Instead, the process of creating objects and connecting them are envisioned and implemented orthogonally. The graph is created using a separate framework: Jamoma Audio Graph. Due to this decoupling of Jamoma DSP and Jamoma Audio Graph we are able to create and use Jamoma DSP unit generators with a number of different graph structures. Jamoma Audio Graph is one graph structure that a developer may choose to employ.

Common real-time audio processing environments including Max/MSP, Pd, SuperCollider, Csound, Bidule, AudioMulch, Reaktor and Reason all have unit generators, but the unit generators can only be used within the particular environment. The unit generators have a proprietary format and thus no interchangeability. Likewise, the graph structure is proprietary. While SDKs are available for some of these environments, for others no SDK exists and the systems are closed. Ability to host Audio Units and VST may extend these environments, but with limitations.

2.2. Audio Graph Structures

A graph structure is an abstract structure where paths are created between nodes (objects) in a set. The paths between these nodes then define the data-flow of the graph. Graph structures for audio signal processing employ several patterns and idioms to achieve a balance of efficiency, flexibility, and real-time capability.

Environments for real-time processing typically need to address concerns in both synchronous and asynchronous contexts. Asynchronous communication is required for handling MIDI, mouse-clicks and other user interactions, the receipt of Open Sound Control messages from a network, and often for driving attributes from audio sequencer automations. Conversely, realtime processing of audio streams must be handled synchronously. Most environments, such as CLAM [3], deal with these two types of graphs as separate from each other. In the Jamoma Platform this is also the case, where Jamoma Audio Graph comprises the synchronous audio graph concerns, and Jamoma Graph implements asynchronous message passing.

To reduce the computational overhead of making synchronous calls through the graph for every sample, most systems employ the notion of *frame processing*. The `TTAudioSignal` class in Jamoma DSP represents audio signals as a collection of audio sample vectors and metadata, containing one vector per audio channel. Jamoma Audio Graph uses the `TTAudioSignal` class to processes these vectors of samples at once rather than a single sample at a time.

2.3. Push vs. Pull

When visualizing a signal processing graph, it is common to represent the flow of audio from top-to-bottom or left-to-right. Under the hood, the processing may be implemented in this top-to-bottom flow as well. Audio at the top of the graph 'pushes' down through each subsequent object in the chain until it reaches the bottom.

Alternatively, audio processing may be driven from the bottom of the chain from a 'terminal object' or 'sink'. This strategy for processing an audio graph, the 'pull' method, is used by several environments including Apple's AUGraph and ChucK [4]. AUGraph and ChucK are subject to certain limitations however: AUGraph does not permit "fanning" connections (many inlets connected to one outlet)[3] while ChucK is not multi-threaded.

[3] http://developer.apple.com/mac/library/documentation/General/Conceptual/SLGlobalGlossary/Glossary/Glossary.html

2.4. Multi-channel Processing

In many real-time audio patching environments, such as Max/MSP, Pd, Bidule or AudioMulch, audio objects are connected using mono signals. For multi-channel spatial processing the patch has to be tailored to the number of sources and speakers. If such programs are considered programming environments and the patch the program, a change in the number of sources or speakers requires a rewrite of the program, not just a change to one or more configuration parameters.

2.4.1. CSound

In Csound multi-channel audio graph possibilities are extended somewhat through the introduction of the "chn" set of opcodes. The "chn" opcodes provide access to a global string-indexed software bus enabling communicating between a host application and the Csound engine, but can also be used for dynamic routing within Csound itself [5]. Below is an example of a simple multi-channel filter implementation using named software busses to iterate through the channels:

```
multi-channel filter, event handler
  instr 2
  iNumC    = p4      ; number of channels
  iCF      = p5      ; filter cutoff
  ichnNum  = 0       ; init the channel number
makeEvent:
  ichnNum  = ichnNum + 1
  event_i  "i", 3, 0, p3, ichnNum, iCF
  if ichnNum < iNumC igoto makeEvent
  endin

multi-channel filter, audio processing
  instr 3
  instance = p4
  iCF      = p5
  Sname    sprintf   "Signal_%i", instance
  a1       chnget    Sname
  a1       butterlp  a1, iCF
           chnset    a1, Sname
  endin
```

2.4.2. SuperCollider

SuperCollider employs an elegant solution by representing signals containing multiple channels of audio as arrays. When an array of audio signals is given as input to a unit generator it causes multi-channel expansion: multiple copies of the unit generator are spawned to create an array of unit generators, each processing a different signal from the array of inputs. In the following example a stereo signal containing white and pink noise is filtered:

```
{
  \\ Create stereo signal as array:
  p = [WhiteNoise.ar, PinkNoise.ar];

  \\ Biquad filter applied to array of channels:
  SOS.ar(p, 1, -1.992, 0.986, 1.992, -0.993, 0.1);
}.play
```

3. DESIGN AND IMPLEMENTATION

3.1. Structure

Jamoma Audio Graph is a framework[4] implementing a synchronous multi-channel audio processing graph driven using a pull methodology. This is accomplished through the creation of a node class for the graph, `TTAudioGraphObject`, which wraps a unit generator with other supporting objects. `TTAudioGraphObject` then manages these instances and the information necessary to pull samples from its sources.

Figure 2 shows these classes and their relations. Here, an object is connected to three upstream objects, and to two downstream objects, with all connections being multi-channel. In database parlance, a connection between objects can be considered as a many-to-many association; each outlet can be connected to many inlets, and each inlet can be connected to many outlets. The source objects represented by $s0$, $s1$, etc. can then be considered as join tables representing each connection in the many-to-many relationship individually.

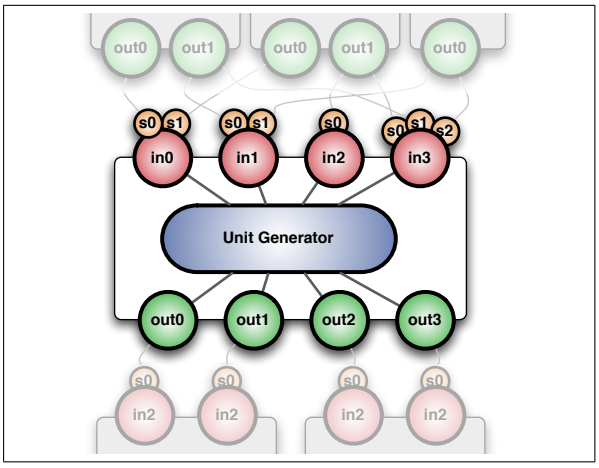


Figure 2: TTAudioGraphObject Class Anatomy

The architecture of `TTAudioGraphObject` is layered: the audio graph object itself only knows about its inlets; the inlets only know about their sources (the so-called 'joins' in database terminology); the sources know from what object and from which outlet they are connected. In the figure this is illustrated by the object having four inlets. Two multi-channel sources $s0$ and $s1$ are connected to the first inlet $in0$, two to the second, and so on. In short:

A graph has many objects.
An object has many inlets.
An inlet has many sources.
A source has many channels.

3.2. Building the Graph

For processing to occur, the connections of the graph must be established. This is accomplished by passing a reference to a source object, as well as the outlet and inlet numbers for the connection, to the downstream object's `connect()` method. Similarly, connections may be cut by passing the same information to the downstream object's `drop()` method. Connections may be created or dropped at any time before, after or during the graph being processed. That is to say that there is no global signal chain compilation; the graph may dynamically change over the course of its operation and performance.

3.3. Processing the Graph

All processing is driven by the object at the end of the processing chain according to a two step process. First, a 'preprocess' method is propagated up the chain from the terminal object. This zeroes buffers and sets flags that indicate each object's processing state. It is of no consequence if an object receives multiple preprocess calls, such as would happen if there are multiple terminal nodes.

Since Jamoma Audio Graph is using a pull-based architecture, an object's *outlets* are passive. They are simply buffers storing the output calculated by the wrapped unit generator. The unit generator is simply an instance of a Jamoma DSP class, specified as an argument when the `TTAudioGraphObject` is instantiated. This unit generator is responsible for actually calculating the audio to be stored by the outlet buffers.

Unlike the outlets, the *inlets* are active. When asked for a vector of audio by the unit generator, the inlets each request audio from each of their sources (other objects' outlets). If an inlet has multiple sources, those sources are summed. When all of the inlets have performed this operation, then the unit generator proceeds to process the audio buffered in the inlets and fills the buffers in the outlets. Sources manage a one-to-one connection between an inlet and an outlet; inlets may have zero or more sources. To summarize:

With the objects in the graph prepared by the `preprocess()` call, the audio can be pulled from the graph by a terminal object using the `process()` call on each of its sources.

3.4. Graph Description

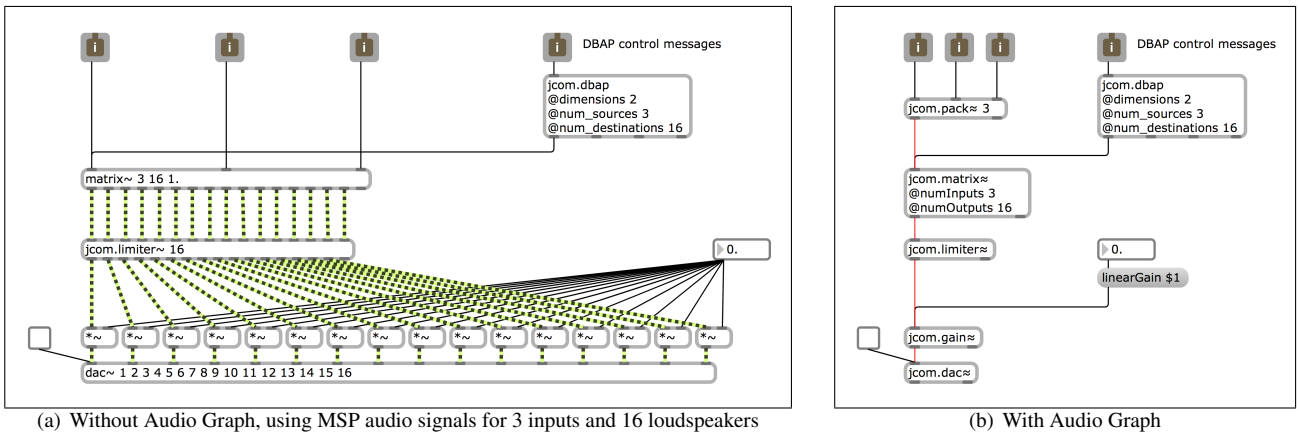
Given an audio graph, the topology can be traversed for purposes other than directly calculating audio. Any node in an audio graph can be queried to create a description. The returned `TTAudioGraphDescription` object will then provide metadata about the entire graph as seen through that object's inlets. There are many applications of this description, including visual representation of the graph, statistical analysis, and cross-coding.

4. APPLICATION

4.1. Max/MSP

A number of externals for multi-channel audio processing in Max have been developed combining Jamoma's Audio Graph and DSP frameworks. The Max implementation of the Jamoma Audio Graph layer represents multi-channel signals as ordinary patch chords. Interfacing between MSP and a Jamoma Audio Graph is facilitated by the externals *jcom.pack≈* and *jcom.unpack≈*, packing and unpacking multiple mono MSP signals into and out of a Jamoma Audio Graph multi-channel signal. *jcom.adc≈* and *jcom.dac≈* offer additional possibilities for direct audio input and output using PortAudio [6] and thus bypassing MSP altogether. Two or more multi-channel signals might be intertwined into one

[4] We use the term framework in a generic sense, as a dynamically linked library together with supporting source and an API.



(a) Without Audio Graph, using MSP audio signals for 3 inputs and 16 loudspeakers

(b) With Audio Graph

Figure 3: The Jamoma signal processing Max patch for Distance Based Amplitude Panning (DBAP)

joint multi-channel signal and separated again using *jcom.join≈* and *jcom.split≈*. The sample rate and vector size used for audio graph processing, as well as the number of channels in a multi-channel signal, can be inspected using *jcom.info≈*.

A number of multi-channel audio generators and processors comprising common signal processing tasks are available. This set of generators can be readily extended. For example, the various units available in the Jamoma DSP libraries can be easily wrapped into Audio Graph externals for Max using a single line of C++ code.

Spectral processing for multi-channel signals is implemented through the *jcom.fft≈* and *jcom.window≈* objects. Several objects are available for level control and mixing, such as *jcom.gain≈* for controlling gain levels of multi-channel signals, while *jcom.matrix≈* allows $M \times N$ matrix-based mixing of the M channels of the incoming signal onto the N channels of the resulting signal. This can be used for various amplitude-based spatialization algorithms such as VBAP [7], DBAP [8] and ambisonics [9]. Finally *jcom.matrixmixer≈* offers the possibility of mixing M incoming multi-channel signals onto N returned multi-channel signals. This combined mixer/router is still rudimentary, and assumes all incoming and returned signal to have the same number of channels.

Figure 3 illustrates the use of Jamoma Audio Graph objects in a patch providing spatialization of 3 mono sources to 16 speakers, with additional post-spatialization mastering of the signals using a limiter and gain adjustment. The use of Jamoma Audio Graph objects greatly helps simplifying the patch as compared to a standard MSP audio graph.

4.1.1. Jamoma Modular

A strong component of the Jamoma Modular framework is the provision of ready-to-use building blocks for sound spatialization [10].

In the past Jamoma used a simple patching hack that wrapped multiple MSP signals into a multi-channel cable. Within each module, this multi-channel connection was then unwrapped in order that audio signals could be processed. One of several disadvantages of this solution was that whenever the number of sources or speakers was changed, MSP objects and connections had to be destroyed and created. Jamoma modules for spatialization require flexibility to enable users to define the number of incoming sounds (to be spatialized) and the number of outgoing audio feeds (number of loudspeakers), thus further requiring that creation and deletion of MSP objects and connections be done using patcher scripting. This creates stress on Max in time critical situations, and is not sufficiently reliable for real-world performance situations. Furthermore, a change in the DSP structure also requires a rebuild of the DSP chain.

The previous scripting and patching hack approach is currently being replaced by Jamoma Audio Graph objects to simplify the spatialization modules and make them more robust. Figure 3b indicates how the number of sources and speakers can be changed simply by updating attribute values for a few Jamoma Audio Graph objects, thus eliminating the need for patcher scripting in Jamoma Modules. Because the Audio Graph signal network can be dynamically reconfigured without the need to recompile the DSP signal chain, on-the-fly changes are possible while audio is running. In other words, a change to the number of channels requires only a change of parameters rather than change to the program itself.

4.2. PureData (Pd)

The Jamoma Audio Graph layer is also implemented for Pure Data, which is essentially similar to Jamoma Audio Graph for Max/MSP. The most obvious difference between Jamoma Audio Graph for Pure Data and Jamoma Audio Graph for Max may be the slightly different naming convention, the Pure Data version using '=' instead of '≈' appended as an indicator for multi-channel signal streams. Figure 4 demonstrates the use of Jamoma Audio Graph in Pd for passing 4 channels through a chain for processing, taking advantage of the multi-channel patch cord connections.

4.3. Ruby

Jamoma Audio Graph is ideally suited for use in many different environments. This includes not only graphical environments, but textual environments as well. The authors have created an implementation of Jamoma Audio Graph for the Ruby language environment. Ruby offers a wealth of application areas include web development using Ruby On Rails [11] and Live Coding [12] using *irb*. The following listing demonstrates a simple *irb* session:

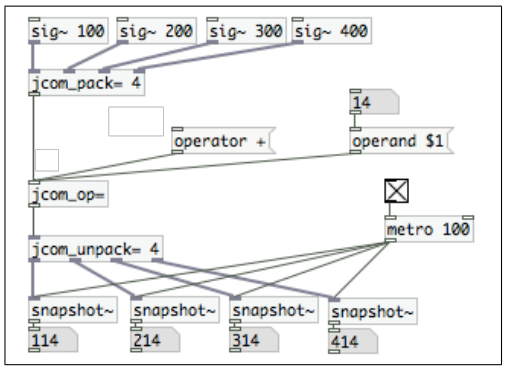


Figure 4: Pd patch, processing 4 channels with Jamoma Audio Graph

```
require 'TTRuby'
dac = TTAudio.new "dac"
osc = TTAudio.new "wavetable"
dac.connect_audio osc
dac.send "start"
osc.set "frequency", 220.0
```

By using Jamoma Audio Graph in the Ruby environment for live coding, users are able to access all of the benefits of using a popular, general purpose language (Ruby) while at the same time gaining access to many of the benefits of a domain specific language for musical performance (Jamoma DSP).

All features of Jamoma Audio Graph can be accessed in real time through a web browser using Ruby on Rails. Through the use of AJAX, interactive widgets can populate a dynamic interface in a web-browser for real time control of parameters, as in the screenshot below.

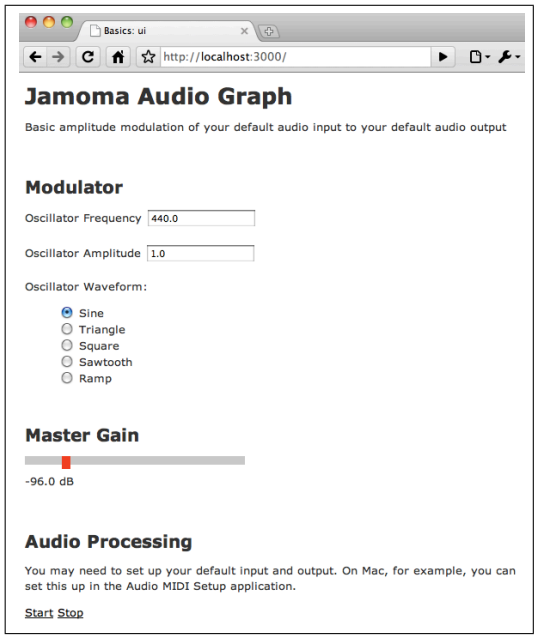


Figure 5: Jamoma Audio Graph graph operating in a Ruby on Rails application through a web browser

4.4. Cross-Coding

Using the capabilities of a graph to produce a description of itself, we are able to port code across environments in a variety of formats. For example, we have implemented functions that export a graph created in Max to a Ruby source file. Likewise we can export from a Ruby source file to Pd and from a Pd canvas to C++ code.

The ability to export code/patcher/canvas/document from any supported environment to any other supported environment offers great flexibility, not only for code generation and prototyping, but for freedom in sharing work across platforms and with colleagues. The currently supported export formats are: Max 5 patcher, C++ source code, Ruby source code and Pure Data.

The additional ability to compile code and export a ready-to-use AudioUnit or VST plug-ins from any of these environments is currently in active development.

5. DISCUSSION AND FUTURE WORK

5.1. In Practice

Audio processing is demanding on several levels. Computational efficiency and performance concerns are just a few of the aspects related to the real-world use and development of any audio graph framework. The Jamoma Audio Graph framework seeks to strike a balance between raw number-crunching, coding and maintaining code, as well as the flexibility and usability of features for end-users in various environments.

5.1.1. Code design

The Jamoma frameworks align with contemporary philosophies for good coding practice, thus facilitating the readability, debugging, maintenance and distribution of code. The frameworks emphasize expressive syntax, idioms, and conventions [13] and adhere to the DRY (Don't Repeat Yourself) principle, which states that "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [14]. Unit testing is so far implemented only in some of the frameworks, mainly for methods processed asynchronously. A general system for developing methods for testing audio signal processes and graphs remains to be developed, but is likely to be much simpler to achieve using the text-based Jamoma Audio Graph implementation for Ruby than it would be within the graphical patching environments Max or Pd.

The emphasis on code design is a long term investment in the creation of frameworks intuitive to use, easy to maintain, well-documented, stable, and supporting an extendable code base, thus fostering a pleasant working environment. We do this to encourage others to make use of the frameworks and join in contributing to future development.

5.1.2. Audio graph patching

The initial motivation for development of Jamoma Audio Graph was simplification of work on spatialization patches in Jamoma Modular. The use of multi-channel signals greatly reduces tedious and repetitious patching where the same chain of audio processes need to be created over and over again for each channel. More importantly, it greatly improves the workflow in terms of flexibility and inter-operability.

Jamoma modules for spatialization share a common interface. The number of sources and speakers are dynamically reconfigured on-the-fly using simple configuration scripts describing the number and positions of sources and speakers. The usefulness of this can be illustrated by the practical experience of one of the authors while working on a sound installation in Oslo in 2002 as reported in [15]: For the final week preceding the opening, two of the composers moved several times a day between working at the 24 speaker setup of the installation site, a 16 speaker setup at a studio and off-site development using stereo headphones [16]. Each time a composer moved between environments the various parts of the patch dealing with spatialization had to be substituted. Using Jamoma Modular it would instead be possible to trigger simple cue scripts to reconfigure the system from one setup to another.

Within Jamoma Modular, Jamoma Audio Graph is eventually planned to be used for all audio modules. One of the major benefits will be the ability to easily reconfigure each module for work on mono, stereo, 5.1 or other surround sound signals. Thus we can avoid creating and maintaining several versions of modules for the same audio process, in accordance with the DRY principle. For modules that also depend on the use of ordinary MSP externals internally, we can use *jcom.pack≈* and *jcom.pack≈* in combination with Max scripting.

5.2. Computational performance

Initial development of Jamoma Audio Graph has focused on the design of an architecturally sound system. As the project emerges from the early stages of development, minimal time has yet to be devoted directly to issues of optimization, though such issues have informed the architecture itself.

Measurement of computational/processor load can be rudimentarily profiled using the facilities of the host environment (e.g. the CPU readout in Max's DSP Status window), or via alternative methods built into special Jamoma Audio Graph classes. In cases where the audio processing occurs independently of the host environment, such as in Max using the `jcom.dac≈` object, the host environment will not be able to adequately measure the computational load of a Jamoma Audio Graph in the same way as would apply to audio processing using FTM and Gabor [17].

Computational performance, however, is not merely a matter of efficient number crunching in the processes, but also efficient and dynamic management of processes. In live performance situations, such as concerts, works for stage and installations, the active processes often change dynamically over the duration of a work. The ability for dynamic patching without interrupting audio delivery promises that processes are seamlessly loaded, unloaded, connected and disconnected on the fly. This further enables the balancing of processing resources because all employed processes need not be present for the entire duration of a work. It additionally opens possibilities for dynamic reconfiguration of the sequence of processes in Max without requiring the signals to pass through the *send~* and *receive~* MSP combo, thus avoiding the introduction of a potentially undesirable vector-length of delay.

Dynamic patching furthermore enables dynamic loading of polyphonic voices as needed. While the maximum number of voices has to be predefined in Max/MSP using *poly~*, Csound instead dynamically creates as many instances of an instrument as required by the score at any one time. Jamoma Audio Graph offers similar possibilities in Max and Pd for polyphonic synthesis. Polyphonic voices might be represented as dynamically changing channels in a portion of the graph, and mixed at the end using a *matrix≈*.

5.2.1. Multithreading

Parallel CPU architectures are becoming increasingly common in desktop computing platforms. UC Berkeley's Parallel Computing Laboratory identified audio applications as one of the most promising but also most challenging use cases for parallel computing [18]. In addition to prevention of race conditions and deadlocks, multi-channel real-time audio computing also requires low latency and jitter-free audio processing execution.

Jamoma Audio Graph supports operation in multi-threaded environments by running multiple parallel graphs on separate threads. Current research is focused on the expansion of multi-threaded support to include intelligent and adaptive multi-threading within a single graph structure.

The implemented pulling strategy (Section 2.3) offers opportunities to analyze and benchmark an audio graph structure's computational performance in real-time. Involved audio processes can then be distributed and managed amongst different threads or processors, particularly where bifurcations of the graph are present, based on a heuristic determination of how to best balance resources against computational load. The Dynamic Component Cluster Rendering, as found in TANGA [19], presents a promising thread-management strategy for Jamoma Audio Graph.

5.3. Flexible audio graph management

5.3.1. Multiple Simultaneous Graphs

By it's nature, an audio graph is defined as all objects connected together into a graph structure terminating at a sink object that drives the graph. There is no limitation on how many of these graphs may exist independently, and simultaneously, in any environment. As these graphs are fully-independent of each other they may run at different rates, in different threads, and addressing different hardware. The independent graphs may even run side-by-side with one running in real-time while another is operating out of real time.

5.3.2. Dynamic vector size

A variety of audio processing algorithms require dynamic change of vector size, dynamic variations in onset position of the vectors, overlapping vectors or varying sampling rate for all or some parts of the audio graph. Examples include the Fast Fourier Transform at various vector sizes and with varying amount of overlap, upsampled FFT for improved frequency resolution at low frequencies and granulation with sample accurate grain onset and duration. In particular the Gabor library for Max/MSP offers several interesting audio signal processing methods that can only be achieved by introducing a signal processing paradigm based on the general idea of streams of atomic sound entities processed with arbitrary timing or arbitrary processing rates. Gabor achieves this by scheduling vectors within the Max event (or message) processing model rather than the block-wise signal stream processing engine of MSP. Sound particles are processed at arbitrary rates or moments and thus can be adjusted to the frequency, texture or rhythm of a sound or to the required processing rate of a particular algorithm [17].

Support for this kind of dynamic reconfiguration of vector sizes, sampling rate and vector onsets remains to be implemented. A future implementation is envisaged to rely on a pair of objects

bridging between the regular part of the graph and a subsection with a higher degree of flexibility. Depending on the audio processing algorithm at hand, the bridging objects might be required to communicate back and forth to ensure that the incoming bridge is looking and buffering sufficiently far ahead in the incoming signal so that it is able to provide the vectors requested from the pulling bridge object at the end of the chain. If the stream of vectors or grains depends on analysis of the incoming signal, as is the case for PSOLA (Pitch Synchronous Overlap-add), the internal communication between the two bridging objects might also be used to control the pull mechanism based on the results of the analysis of incoming audio.

5.3.3. Implicit Patching

Currently all connections in an audio graph are managed manually. In a graphical environment such as Max we could say that the connections between objects are explicitly patched together by the user. A higher-level alternative to this "explicit patching" by the user is the "implicit patching" paradigm embraced by Marsyas [20].

Through the use of implicit patching the user is able to interact with groups of objects that are networked together by specifying a *pattern* to use, rather than directly and manually making each connection. One elementary example is the creation of a multi-band EQ, by creating an array of bandpass filters and then specifying them to connect as a parallel group. A group patched according to a 'series' pattern would connect one after the other to apply a transformation throughout the chain of objects in the group.

5.4. Improving support for work on spatial audio

The advantages of multi-channel audio patches in common computer music environments can be clearly seen in Figure 3. The ability to work individually on a single audio channels within the audio graph is currently limited. For instance, it is possible to apply jcom.filter≈ to manipulate all audio channels within a graph connection in the same way. However, many multi-channel audio applications (such as the Near Field Compensation in Higher Order Ambisonics) rely on filtering processes individualized across audio channels. Future work will address this issue. Jamoma Audio Graph can be further extended to support upmix/downmix algorithms that seamlessly mediate between different multi-channel media formats including compressed spatial audio material (e.g. mp3-surround, DirAC [21]). We plan to explore the potential of attaching additional meta-information to the individual audio channels to ensure that multi-channel audio material is correctly processed (e.g. to identify the signal components in an Ambisonics B-format, or to distinguish a binaurally processed audio stream from a conventional stereo track).

Related to IEM's CUBEmixer [22], the authors are currently developing a multi-channel authoring/mixing application which enables the user to work and combine common spatial audio rendering concepts such as VBAP, DBAP, ViMiC [23] and Ambisonics according to compositional aspiration and/or technical possibilities [10]. The Jamoma Audio Graph layer is essential to this development and is already being used in a prototype of this application (Figure 6).

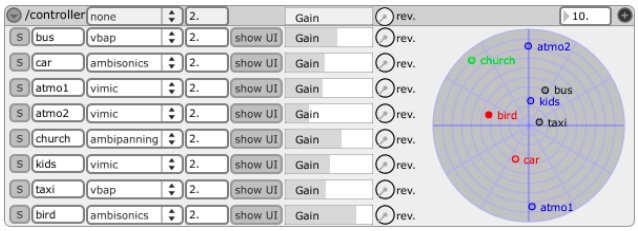


Figure 6: Prototype of an authoring application combining common spatial rendering concepts using Jamoma Audio Graph

6. CONCLUSIONS

The Jamoma Audio Graph framework implements a flexible and dynamic architecture for creating networks of multi-channel unit generators that operate in real time. The connections between objects are not only capable of delivering multiple channels of audio through each individual connection, but the number of audio channels may change dynamically during the operation of the audio graph. Furthermore, the connections themselves are dynamic, meaning that unit generators may be connected or detached during the operation of the audio graph without requiring the signal chain to be re-compiled, and giving rise to live coding and live patching applications. Finally, the vector size of the signal processing graph, as well as the sample rate of the graph is dynamic with parts of the graph that may operated at different rates for spectral, granular, or other areas of interest.

Through implementations for a variety of text-based and graphical environments, we have proven the flexibility of the architecture and the potential for great usability improvements to those environments. We believe the structure to be a good fit for multi-threaded parallel processing, which comprises much of the future development of Jamoma Audio Graph. As an open source project with cross-platform support, we believe that Jamoma Audio Graph provides a relevant and applicable model for developing real time audio processing systems.

7. ACKNOWLEDGMENTS

The authors wish to thank Alexander Refsum Jensenius and the fourMs lab at the University of Oslo for hosting the workshop during which the initial architecture of Jamoma Audio Graph was designed. We additionally thank Pascal Baltazar and everyone at GMEA Centre National de Création Musicale for hosting a spatialization workshop during which the design of Jamoma Audio Graph was reviewed and heavily revised. Initial development was supported by The Municipality of Bergen. Øyvind Brandsegg, Jeff Carey and IOhannes Zmölnig have provided valuable insight into processing of multi-channel signals in Csound, SuperCollider and Pd respectively. Jesse Allison provided assistance with the Ruby on Rails implementation of Jamoma Audio Graph.

8. REFERENCES

[1] Timothy Place, Trond Lossius, and Nils Peters, "A flexible and dynamic C++ framework and library for digital audio signal processing," in *Accepted for ICMC 2010*, 2010.

[2] Timothy Place and Trond Lossius, "Jamoma: A modular standard for structuring patches in Max," in *Proc. of the*

2006 International Computer Music Conference, New Orleans, US, 2006, pp. 143 – 146.

[3] Xavier Amatraian, "A domain-specific metamodel for multimedia processing systems," *IEEE Transactions on Multimedia*, vol. 9, no. 6, pp. 1284–1298, 2007.

[4] Ge Wang, *The ChucK Audio Programming Lanuage: A Strongly-timed and On-the-fly Envon/mentality*, Ph.D. thesis, Princeton University, 2008.

[5] Steven Yi, "Creating encapsulated instruments in Csound5," *Csound Journal*, Winter 2006.

[6] Ross Bencina, "PortAudio and media synchronisation - it's all in the timing," in *In Proc. of the 2003 Australasian Computer Music Association (ACMC'03)*, 2003, pp. 13–20.

[7] Ville Pulkki, "Virtual sound source positioning using vector base amplitude panning," *J. Audio Eng. Soc.*, vol. 45(6), pp. 456–466, 1997.

[8] Trond Lossius, Pascal Baltazar, and Théo de la Hogue, "DBAP - Distance-Based Amplitude Panning," in *Proc. of 2009 International Computer Music Conference*, Montreal, Canada, 2009, pp. 489–492.

[9] Jan C. Schacher and Philippe Kocher, "Ambisonics Spatialization Tools for Max/MSP," in *Proc. of the 2006 International Computer Music Conference*, New Orleans, US, 2006, pp. 274–277.

[10] Nils Peters, Trond Lossius, Jan Schacher, Pascal Baltazar, Charles Bascou, and Timothy Place, "A stratified approach for sound spatialization," in *Proc. of 6th Sound and Music Computing Conference*, Porto, Portugal, 2009, pp. 219–224.

[11] Sam Ruby, Dave Thomas, and David Heinemeier Hansson, *Agile Web Development with Rails. 3rd edition*, The Pragmatic Programmers LLC, 2009.

[12] Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward, "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 3, pp. 321–330, December 2003.

[13] Robert C. Martin, *Clean code. A handbook of Agile software craftsmanship*, Prentice Hall, 2009.

[14] Andrew Hunt and David Thomas, *The Pragmatic Programmer*, Addison-Wesley, 1999.

[15] Jøran Rudi, "Norge et lydrike - norway remixed: a sound installation," *Organised Sound*, vol. 8, no. 2, pp. 151–155, August 2003.

[16] Trond Lossius, *Sound Space Body: Reflections on Artistic Practice*, Ph.D. thesis, Bergen National Academy of the Arts, 2007.

[17] Norbert Schnell and Diemo Schwarz, "Gabor, multi-representation real-time analysis/synthesis," in *Proc. of the 8th International Conference on Digital Audio Effects (DAFx'05)*, Madrid, Spain, 2005.

[18] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J.D. Kubiatowicz, E.A. Lee, N. Morgan, G. Necula, D.A. Patterson, et al., "The parallel computing laboratory at UC Berkeley: A research agenda based on the berkeley view," Tech. Rep., UC Berkley, Tech. Re. UCB/EECS-2008, 2008.

[19] Andreas Partzsch and Ulrich Reiter, "Multi core / multi thread processing in object based real time audio rendering: Approaches and solutions for an optimization problem," in *122th AES Convention, Preprint 7159*, 2007.

[20] Stuart Bray and George Tzanetakis, "Implicit patching for dataflow-based audio analysis and synthesis," in *Proc. of the 2005 International Computer Music Conference*, 2005.

[21] Ville Pulkki, "Spatial sound reproduction with directional audio coding," *J. Audio Eng. Soc.*, vol. 55, no. 6, pp. 503–516, June 2007.

[22] Thomas Musil, Winfried Ritsch, and Johannes M. Zmölnig, "The CUBEmixer a performance-, mixing- and mastering tool," in *Proc. of the 2008 Linux Audio Conference*, 2008.

[23] Nils Peters, Tristan Matthews, Jonas Braasch, and Stephan McAdams, "Spatial sound rendering in Max/MSP with ViMiC," in *Proc. of the 2008 International Computer Music Conference*, Belfast, UK, 2008, pp. 755–758.