THE DEVELOPMENT OF A MULTIFACETED PARAMETER MODEL IN JAMOMA

First author
School
Department

Second author
Company
Address

Third author
Company
Address

ABSTRACT

A prototype implementation in Jamoma is presented and discussed.

1. INTRODUCTION

Fundamental to the development of musical or artistic material is the ability to transform raw materials. This ability implies the facility to master many facets of the material, and to manipulate it with plasticity. Computer music environments typically provide points of control to manipulate material by providing parameters with controllable values. However, we find that this capability to control the values of parameters is inadequate for many of our artistic endeavors, and does not reflect the analogous tools and methods of artists working with physical materials. For this reason we have started to explore alternative ways of working with dynamic structures.

A possible partial solution is to treat a parameter not as a single-value representing entity, but to treat a parameter as a multi-dimensional tool or object. Thus the parameter, as a tool or object, has many facets itself in addition to the value it renders. These many *properties* of the parameter define its behavior. This paper presents a prototype of these ideas, implemented in Jamoma, a modular framework for Max/MSP [3]. In addition to defining behaviors, the behaviors are themselves interdependent upon each other requiring a flexible and dynamically bound code base for the implementation.

2. MVC

Jamoma is a Model-View-Controller (MVC) framework that supplies a modular structure for the Max/MSP environment. "MVC programming is the application of [a] three-way factoring, whereby objects of different classes take over the operations related to the application domain (the model), the display of the application's state (the view), and the user interaction with the model and the view (the controller)." [1] In Jamoma, the model is represented as a Max patcher or external object called the *algorithm*. The user interface, embedded in a patcher or bpatcher is the view. The controller is composed of a collection of interacting and dynamically-bound objects, which we refer to as the *Jamoma Core*.

The messaging model in Jamoma addresses modules using *parameters* (i.e. nodes with state) and *messages* (i.e. nodes which are stateless). For the duration of this paper we will often refer only to parameters, but the concepts generally apply to both parameters and messages.

3. DYNAMIC BINDING

For our purposes we will assume the *materials* we are working with to audiomusical information, such as audio samples or midi representations of a performance. The plasticity we refer to means we want to easily shape and mold our materials into a final sonic or visual output.

Current practice for many users of computer music software is predicated on static relationships and the use of static presets. This is certainly true for many DAW systems whose overall structure is fixed. It is, however, also true of open-ended systems, such as PureData or Max/MSP. In a graphical environment, the relationships between objects and their interconnections form the algorithm that determines a tools behavior. Within this algorithm there is typically some freedom to modify its behavior by e.g. changing coefficients. However, the objects and connections generally do not change on-the-fly as a performance is executed.

Many of these systems, Max/MSP in particular, have provisions for breaking out of sets of static relationships through scripting. For the vast majority of users, however, mastering this task is onerous at best. By keeping these relationships fixed, the expressivity available to the user is inherently limited.

Jamoma aims to address this through the use of *dynamic binding*. One prevalent example of dynamic binding is the Objective-C language. Static binding "constraints are limiting because they force issues to be decided from information found in the programmers source code, rather than from information obtained from the user as the program runs." [1]

Dynamic binding in Jamoma takes place on several levels. At the highest level, a module's parameters are bound to its hub dynamically to form the Controller layer of the MVC paradigm. Then dynamic binding takes place inside of the parameters as its properties are changed.

[1] `http://developer.apple.com/documentation/Cocoa/Conceptual/OOP_ObjC/Articles/chapter_5_section_6.html`

4. THE PARAMETER

The parameter is the primary interface for a user manipulating the state of a module. In most systems, the parameter has a single task: to set a variable or coefficient. While it is straightforward to understand such a simple 1-dimensional control, it does not offer the degree of nuance that, say, a sculptor has when working with clay.

In Jamoma, the parameter is made multi-dimensional through the use of *properties* and *methods* in addition to maintaining the state of a value [5]. These properties define the behavior for parameters by setting a value range, repetition filtering, the type of units used to express values, and how automation is applied. The following is list of available properties and methods for any given parameter.

Following is a list of the properties and methods of a parameter.

`:/value`	The value of the parameter.
`:/value/stepsize`	The size of the step taken by the inc and dec messages.
`:/value/inc`	Increase the value of the parameter by the stepsize.
`:/value/dec`	Decrease the value of the parameter by the stepsize.
`:/value/default`	The initial value of a parameter when the module is first created.
`:/type`	The type of data represented by the parameter.
`:/priority`	The value of the parameter.
`:/ui/freeze`	The value of the parameter.
`:/ui/refresh`	The value of the parameter.
`:/ramp/drive`	The value of the parameter.
`:/ramp/function`	The value of the parameter.
`:/repetitions`	The value of the parameter.
`:/range/bounds`	The value of the parameter.
`:/range/clip`	The value of the parameter.
`:/description`	The value of the parameter.
`:/node/type`	The value of the parameter.
`:/node/name`	The value of the parameter.
`:/dataspace`	The value of the parameter.
`:/dataspace/unit/active`	The value of the parameter.
`:/dataspace/unit/native`	The value of the parameter.

A parameter's behavior is ultimately determined by a compendium of these properties. Some properties are simple values, such as the `:/repetitions` property. Other properties may be a dynamic object which itself possesses a number of properties. Such properties include the `:/ramp/drive`, `:/ramp/function`, and various `:/dataspace` properties.

4.1. Implementation

The parameter is implemented as a Max external called *jcom.parameter*. Within jcom.parameter, the ramp and dataspace properties are implemented internally as dynamically bound objects using the TTBlue framework [2]. The TTBlue framework is a C++ library that implements a dynamic messaging layer rather than using statically-linked C/C++ function calls [4].

By using dynamically linked components inside of the parameter, it is possible to switch between many different options for a give parameter property, while each option may implement an entirely different set of methods. This will be demonstrated in upcoming sections.

[2] `http://www.electrotap.com/ttblue/`

5. THE JAMOMA LIBRARIES

The functionality we refer to within a parameter is implemented in a shared library. Within the shared library the different functionalities are grouped in a series of "Libs". These libraries provide the key functionalities for developing multidimensional tools. To date, the libs are composed of:

- FunctionLib: a library of FunctionUnits which map an input value to an output value
- RampLib: a library of driving mechanisms (RampUnits) which use the FunctionLib to automate value transformations over time.
- DataspaceLib: a library by which a parameter or message can be given a class that describes the type of data it represents. Values may then be set by any of a number of DataspaceUnits to allow control of a parameter in any of a number of ways.

Why do we need to work with Functions and Ramps and Dataspaces: - we have ramps everywhere -¿ dynamic (reconfigurable) non-static - different functions - much more flexible than pattr

Things in Common among the 3: So that we have a library - shared so that everywhere we need it we can access it - easily extendable - queryable

5.1. The Function Library

The Jamoma FunctionLib API provides normalized mappings of values $x \in [0, 1]$ to $y \in [0, 1]$ according to functions $y = f(x)$. The FunctionLib can easily be expanded by introducing new functions in the form of two C++ files: a source file and a header file that provides an interface for the source file.

Currently five functions are implemented:

- Linear: $y = x$.
- Cosine: $y = -\frac{1}{2} \cdot cos(x \cdot \pi) + \frac{1}{2}$.
- Lowpass series: $y[n] = y[n-1] \cdot k + x[n] \cdot (1-k)$, where k is a feedback coefficient.
- Power function: $y = x^k$, parameter k can be set.
- Hyperbolic tangent: $y = c \cdot (\tanh(a \cdot (x-b)) - d)$, where coefficients a, b, c, d depends on the width and offset of the curve.

5.2. The Ramp Library

Jamoma offers vastly extended possibilities in how ramping can be done as compared to Max. In Jamoma the process of ramping is made up from the combination of two components: A driving mechanism cause calculations of new values at desired intervals during the ramp, while a set of functions offers a set of curves for the ramping. Both components are implemented as C++ APIs, and can easily

	Cosine	Linear	Lowpass	Power	Tanh
Scheduler		x			
Queue					
None					
Async					

Table 1. The possible ramping configurations in Jamoma

be extended with new ramp or function *units*, expanding the range of possible ramping modes.

The Jamoma RampLib API provides a means by which to create and use *ramp units* in Jamoma. A ramp unit is a self-contained algorithm that can slide from an existing value to a new value over a specified amount of time according to different timing mechanisms. Each ramp unit is implemented in the form of two C++ files: a source file and a header file that provides an interface for the source file. Currently four such ramp units are implemented:

- *none* - jumps immediately to the new value. Typically used for values where ramping do not make sense.
- *scheduler* - use the Max internal clock to generate new values at fixed time intervals.
- *queue* - ramping using the Max queue, updating values whenever the processor has free capacity to do so.
- *async* - only calculate new values when requested to do so. This might be used in video processing modules to calculate fresh values immediately before processing the next video image or matrix.

When a new ramp is started, the ramp unit internally use a normalized ramping value, increasing linearly from 0.0 to 1.0 over the duration of the ramp. Whenever the ramp unit is to provide a new value, it updates the normalized ramping value, and pass it to a Function Unit as described in Section **??**. The normalized value returned is then scaled to the range defined by the start and end values for the ramp, and passed on to the module.

5.3. The Dataspace Library

The DataspaceLib is important because... using different unit types and moving toward more perceptual/semantic representations instead of being chained to technical terms.

It will be good for us to try and explain all the interactions here ;-)

- AngleDataspace
- NoneDataspace
- ColorDataspace
- PitchDataspace
- PositionDataspace
- DistanceDataspace
- TemperatureDataspace
- GainDataspace
- TimeDataspace

5.4. TemperatureDataspace

La de da...right now 1C in Montreal

6. INTERDEPENDENCIES

This paper discusses the structure and development of the interrelated libraries that are used to implement this system.

7. DISCUSSION AND FURTHER WORK

The problem: It has been too static - too difficult to create dynamic setups - for example, dynamic setups in Max using scripting: ick!

Trying to create something that is more flexible, but also easier to work with at the same time.

Splitting up and identify the way we handle structures in Max and put them together. It has been very static and preset-based.

As we've been working on this for along, we hit a wall and needed to move forward, and did so by creating these libs to extend Max.

No one wants to have to manually do all of the housekeeping to get this functionality, so we've made it largely encapsulated behind the scenes.

[2]

4 ramp units times 5 function units = 20 ramping modes

ramp units can be used for other scheduled processes as well

Possibility of expanding ramp units as low frequency oscillators

function units can be used elsewhere, e.g. for mapping

Audio rate ramp unit.

DataspaceLib

Querying - we propose a different system to the Lemur OSC2 draft

Ramp Lib and Function Lib can be used outside the context of jcom.parameter and jcom.value: jcom.map and jcom.ramp

Plans to extend the FunctionLib by introducing exponential functions are under discussion.

* ramping with our own scheduler to take the burden off of the Max scheduler.

8. REFERENCES

[1] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[2] A. Momeni and D. Wessel. Characterizing and controlling musical material intuitively with geometric models. In *Proceedings of the 2003 Conference on*

New Interfaces for Musical Expression, pages 54–62, Montreal, Quebec, Canada, May 2003.

[3] T. Place and T. Lossius. Jamoma: A modular standard for structuring patches in max. In *Proceedings of the International Computer Music Conference*, pages 143–146, New Orleans, LA, 2006.

[4] T. Place and T. Lossius? Ttblue: A dynamically-bound dsp library in c++. In *Submitted to DAFX 2008*, Helsinki, Finland, 2008.

[5] T. Place, T. Lossius, A. R. Jensenius, N. Peters, and P. Baltazar. Addressing classes by differentiating values and properties in osc. In *Submitted to NIME 2008*, Genova, IT, 2008.