THE DEVELOPMENT OF A MULTIFACETED PARAMETER MODEL IN JAMOMA

First author
School
Department

Second author
Company
Address

Third author
Company
Address

ABSTRACT

The authors propose a expanded notion of how we work with parameters in computer-based environments for time-based art. Rather than viewing parameters a single points of control, the authors argue that parameters must become more multifaceted and dynamic in order to serve the needs of artists. A proposed partial solution to this problem is to give parameters additional properties that define their behavior. An example implementation of these ideas is presented in Jamoma, where these properties are themselves dynamic and interdependent upon each other.

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 our purposes we will assume the *materials* we are working with to be media related to time-based art, such as real-time processing of audio, video, midi or other kinds of data as part 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.

1.1. The Struggle to be Dynamic

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 Digital Audio Workstations (DAWs) 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 tool's 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.

To address this need for systems which are more dynamic, a possible partial solution is to treat parameters not as a single-value representing entities, but to treat parameters as a multi-dimensional tools or objects. 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. DESIGN AND CONCEPTS

2.1. Model-View-Controller

Jamoma is a Model-View-Controller (MVC) framework that provides 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]

2.1.1. Model / Algorithm

In Jamoma, the model is represented as a Max patcher or external object called the *algorithm*. The algorithm is a standalone abstraction which performs the task of a module, but has no user interface and does not manage its own state.

2.1.2. View / GUI

The user interface may consist of one or more objects in the module. These are made visible to the user by embedding the module in a bpatcher, or by opening a view of the patcher in Max. Much of the GUI is automated.

Those parts which aren't are tied to the controller through various objects in the JamomaCore.

2.1.3. *Controller / JamomaCore*

The controller layer is responsible for managing all of the relationships in a module, and as such plays the most important role in creating a dynamic system. The controller is composed of a collection of interacting and dynamically-bound objects, which we refer to as *JamomaCore*. At the heart of JamomaCore is the *hub*. The hub acts as a dispatcher, and it coordinates all of the activity in a module.

The messaging model in Jamoma addresses modules using *parameters* (nodes with state) and *messages* (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.

2.2. Dynamic Binding

To implement a controller layer of the MVC framework in a way that makes it able to easily handle continuously changing relationships among the components, it is important to use a method of *dynamic binding*. Dynamic binding can be used on conceptual levels, or as meaning something specific in a variety of contexts. One prevalent example of a dynamically bound environment is the Objective-C language. Objective-C documentation discusses static binding, stating that the "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.

2.3. A Multidimensional 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 [4]. A property is an aspect of the parameter which has a state. A method is simply a mechanism for doing something, such as refreshing the user interface for the parameter. 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.

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

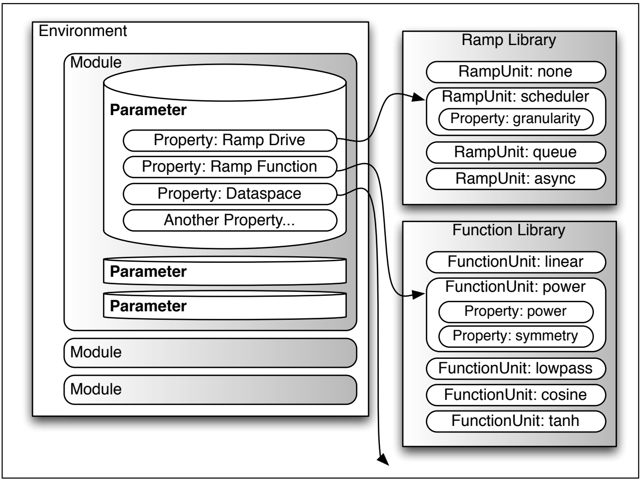

Figure 1. Parameter structure in context.

Figure 1 shows a parameter in context. Within an environment, there may be many modules. Each module may have many parameters, the number of which may change dynamically. Each parameter may have many properties. These properties may address static or dynamic entities. The properties may point to a dynamic entity which itself has properties, and so on.

The example given in Figure 1 shows a number of properties. A common need is for a parameter to interpolate from its existing value to a new value. In Jamoma we call this *ramping*. Ramping is implemented in a parameter using two components: a driving mechanism and a function (shape). The driving mechanism performed by a RampUnit (discussed in Section 4.1) which may have properties and methods of its own. The shape of the ramp is performed by using a FunctionUnit (discussed in Section 4.2) which also may have properties and methods of its own. All of these are reconfigurable on-the-fly during performance.

3. PARAMETER IMPLEMENTATION

At this time, the parameter object in Jamoma implements a variety of both static and dynamic properties. A parameter's behavior is ultimately determined by a compendium of these properties. Table 1 is a brief summary of these.

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 located in a set of shared libraries. These shared libraries are called the FunctionLib, RampLib, and DataspaceLib respectively.

4. THE JAMOMA LIBRARIES

The Jamoma libraries deliver a shared resource to all of the Jamoma framework for applying mathematical functions, converting units, and ramping. For example, the

`:/value`	Value of the parameter
`:/value/stepsize`	Size of step taken `inc` and `dec`
`:/value/inc`	Increase the value
`:/value/dec`	Decrease the value
`:/value/default`	Initial value
`:/type`	Type of data
`:/priority`	Order for recalling values from a preset
`:/ui/freeze`	Stops GUI updates to save CPU
`:/ui/refresh`	Updates the GUI
`:/ramp/drive`	Timing mechanism for ramps
`:/ramp/function`	Interpolation shape for ramps
`:/repetitions`	Filter out repeated values
`:/range/bounds`	Set a low and high range
`:/range/clip`	What to do when the range is exceeded
`:/description`	Documentation
`:/node/type`	"parameter" or "message"
`:/node/name`	Parameter's name
`:/dataspace`	Class of values being controlled.
`:/dataspace/unit/active`	The measurement unit used when values are sent
`:/dataspace/unit/native`	The measurement unit used by the internal algorithm

Table 1. Parameter properties in Jamoma

ramping functionality is not only implemented in parameters, but also in messages, special ramping objects, and in other places. The tools implemented in the libraries are pervasive throughout the environment.

Each of the libraries furnish a clear programming interface so that they are easily extendable. The dynamic binding implementation in the Jamoma framework means that by simply creating one *unit* (object), it is immediately available to the rest of the Jamoma environment with no additional upkeep or maintenance elsewhere in the code.

The libraries can also be queried to find out what functionalities exist. This happens at several levels. For example, a user may wish to find out what functions exist for doing a mathematical mapping. The FunctionLib can provide a list of available FunctionUnits. Having chosen a FunctionUnit, the user can then query to find out what additional properties (if any) the FunctionUnit has published for access. A good user interface will automate all of this querying to simply provide updated selections and options.

Currently there are three libraries in the Jamoma framework:

- 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.

4.1. 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 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 4.2. 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.

4.2. 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	Linear	Lowpass	Power	Tanh
Scheduler		x			
Queue					
None					
Async					

Table 2. The possible ramping configurations in Jamoma

- 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.

4.3. 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

4.3.1. TemperatureDataspace

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

4.4. Interdependencies

One of the largest potentials of the system we have outlined is that these dynamically bound libraries can be used together. One simple example of this is how a ramp drive mechanism can be paired with a function that determines the ramp's shape, and a dataspace determining what kind of unit to use for ramping. This provides for many possibilities.

5. 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.

6. 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, 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.