FLEXIBLE CONTROL OF COMPOSITE PARAMETERS IN MAX/MSP

Timothy Place,[a] *Trond Lossius,*[b] *Alexander Refsum Jensenius,*[c] *Nils Peters*[d]
[a] Electrotap, tim@electrotap.com
[b] BEK - Bergen Center for Electronic Arts, lossius@bek.no
[c] University of Oslo, a.r.jensenius@imv.uio.no
[d] CIRMMT, McGill University, Montréal, nils.peters@mcgill.ca

ABSTRACT

Rather than viewing parameters in computer-based systems as single points of control, the authors posit that parameters must become more multifaceted and dynamic in order to serve the needs of artists. The authors propose an expanded notion of how to work with parameters in computer-centric environments for time-based art. 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.

1. INTRODUCTION

Fundamental to the development of musical or artistic content is the ability to transform raw *materials*. This ability implies the facility to master many facets of the material, and to shape it with plasticity. Computer music environments typically provide points of control to manipulate material by providing parameters with controllable values. This capability to control the values of parameters is inadequate for many artistic endeavors, and does not reflect the analogous tools and methods of artists working with physical materials.

Presets and *automation* in computer music systems can be considered archetypes of strategies for dynamic control of material. Presets in their purest form are a vertical-only approach; all values are instantly set to a certain state. Automation, on the other hand, in its purest form is a horizontal-only approach; a fixed stream of time-tagged values progressing over a limited amount of time to control the state of one parameter, often with some sort of interpolation from one value to the next. While presets are widely used in real-time signal processing environments, the use of automation is fundamental to linear time-based media software such as digital audio workstations and video editing software.

Both presents and automation rely on the *mappings* between parameters in the system, which typically need to be both horizontal and vertical to work efficiently [3, 9].

One obvious way of expanding the flexibility of presets is by implementing a cross-fade or gradual transition to the new preset by means of interpolation. Several works have expanded this further by presenting the set of presets as points in a dataspace and develop strategies of traversing that dataspace, creating dynamic interpolations between two or more presets [2, 5, 1]. This has also been extensively used by one of the authors for developing the Hipno audio plugins. [1]

Jamoma [2] is a modular framwork for the Max/MSP environment [6]. In many ways Jamoma may be seen as a *Model-View-Controller* (MVC) framework, where "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)."[4, p26]. All state management, parametric control, and automation for Jamoma is handled within the controller layer of the MVC paradigm. This forms the basis of all relationships both within a *module*, i.e. a high-level unit with a graphical user interfance, and between different modules.

In Jamoma we are currently working towards more complex transitions of parameters in time that integrate both vertical and horizontal qualities. This is achieved through the integration of a cuelist system which (vertical), and ramping with a number of different curves (horizontal). Previously Jamoma offered possibilities of ramping to a new value over a certain amount of time by means of linear interpolation. Recently this has been expanded by re-implementing ramping as a combination of two new libraries.

2. THE COMPOSITE 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 one-dimensional control, it does not offer the degree of nuance that, say, a sculptor has when working with clay.

In Jamoma, the parameter is expanded by adding *properties* and *methods* to the parameter that further refine or change its behavior [7]. These behaviors themselves can be in constant fluid motion together with the value of the parameter. Some examples of parameter properties include setting a value range, filtering out repetitions, determining the type of units used to express values, and how automation is applied. The result is a composite parameter or *node*, made up of many constituent parts rather than

[1] `http://www.cycling74.com/products/hipno`
[2] `http://www.jamoma.org`

Property	Description
`:/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

Table 1. Parameter properties in Jamoma

representing only a single value. As such it is more like a multi-dimensional tool than a single point of control.

2.1. Properties and Methods

We have stated that a parameter may be enhanced by the addition of properties and methods.

A *property* is an aspect of the parameter which itself has a state. For example, filtering of repeating values can be turned on or off. A *method* is simply a mechanism for doing something, such as refreshing the user interface for the parameter, but the method itself does not have any value to maintain. Often when we refer to properties, we are collectively referring to both the properties and the methods of a parameter.

One interesting aspect of properties that does not apply to methods is that properties may themselves have properties. For example, a property that sets a type of filter may then have additional properties that control the coefficients specific to the type of filter selected.

2.2. Parameter Properties in Jamoma

Jamoma's parameter object is an implementation of the idea that properties and methods can meaningfully extend parametric control. When communicating to and from modules using the Open Sound Control protocol [10] we use the colon separator to access the properties of the parameter as proposed in [7]:

```
/path/to/parameter <value>
/path/to/parameter:/property <value>
```

Table 1 lists the currently implemented properties of the parameter object, with the path to the parameter omitted. The underlying details of how this is implemented are detailed in Section 3.1.

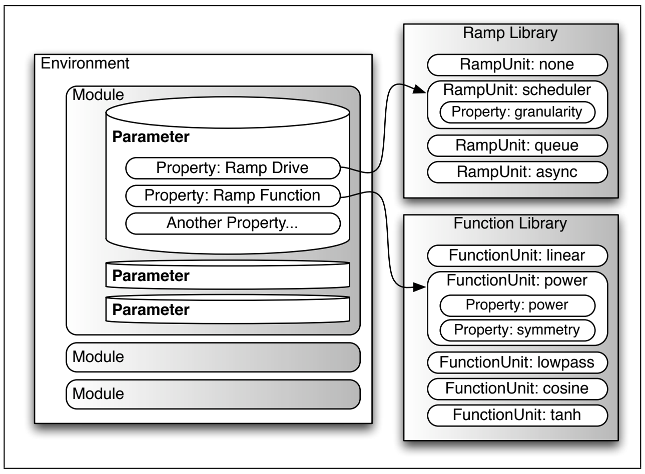

Figure 1. Parameter structure in context.

3. 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.

The parameter is implemented as a Max external called *jcom.parameter*. Within jcom.parameter, the ramping properties are implemented internally as dynamically bound objects located in two shared libraries, called the *RampLib* and *FunctionLib*.

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 or shape. The driving mechanism is performed by a *RampUnit* (discussed in Section 3.1) which may have properties and methods of its own. The shape of the ramp is performed by using a *FunctionUnit* (discussed in Section 3.2) which also may have properties and methods of its own. All of these are reconfigurable on-the-fly during performance.

3.1. The Ramp Library

Depending on the circumstance it might be desirable to generate new interpolated values in different ways during the ramp: Several real-time signal processing environments distinguish between audio rate and control rate signals. Depending on the transients of the ramping signal it might sometimes be desirable to perform interpolation at audio rate, at other time it might be sufficient to update at control rate. If the parameter is controlling a video processing algorithm it might be sufficient to update the value once per video frame processed.

The Jamoma RampLib, implemented as a C++ 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. 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.

The RampLib can easily be extended with more ramp units, and one planned extension is the implementation of audio rate ramping.

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 3.2. 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.

3.2. The Function Library

The Jamoma FunctionLib API provides normalized mappings of values $x \in [0, 1]$ to $y \in [0, 1]$ according to functions $y = f(x)$. 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.

The FunctionLib can easily be expanded by introducing new functions as C++ files. There are immediate plans for introducing additional exponential functions.

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

	Cosine	Linear	Lowpass	Power	Tanh
Scheduler					
Queue					
None					
Async					

Table 2. The possible ramping configurations in Jamoma

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.

3.3. 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.

4. 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.

4 ramp units times 5 function units = 20 ramping modes (see Figure 2).

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.

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.

4.1. DataspaceLib

When sending values to a parameter, it is important to know what kind of value the parameter expects.

5. REFERENCES

[1] R. Bencina. The metasurface – applying natural neighbour interpolation to two-to-many mapping. In *Proceedings of The 2005 International Conference on New Interfaces for Musical Expression (NIME05)*, 2005.

[2] P. Dahlstedt. Creating and exploring huge parameter spaces: Interactive evolution as a tool for sound generation. In *Proceedings of the 2001 International Computer Music Conference*, pages 235–242, Habana, Cuba, 2001. San Francisco: ICMA.

[3] A. Hunt, M. M. Wanderley, and M. Paradis. The importance of parameter mapping in electronic instrument design. *Journal of New Music Research*, 32(4):429–440, 2003.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] T. Place, N. Wolek, and J. Allison. *Hipno: Getting Started.* Cycling'74 and Electrotap, 2005.

[9] D. van Nort and M. M. Wanderley. Exploring the effect of mapping trajectories on musical performance. In *Proceedings of Sound and Music Computing*, Marseille, France, 2006.

[10] M. Wright, A. Freed, and A. Momeni. Opensound control: State of the art 2003. In *Proceedings of NIME-03*, Montreal, 2003.