Demo: Using Jamama's MVC features to design an audio effect interface

Trond Lossius
BEK
trond.lossius@bek.no

Nathan Wolek
Stetson University
nwolek@stetson.edu

Theo de la Hogue
GMEA
theod@gmea.net

Pascal Baltazar
L'Arboretum
pascal@baltazars.org

ABSTRACT

The Model-View-Controller (MVC) software architecture pattern separates these three program components, and is well-suited for interactive applications where flexible human-computer interfaces are required. Separating data presentation from the underlying process enables multiple views of the same model, customised views, synchronisation between views, and views that can be dynamically loaded, repurposed, and disposed.

The use of MVC is widespread in web applications, but is far less common in interactive computer music programming environments. Jamoma 0.6 enables MVC separation in Cycling'74 Max, as presented in [1]. This demonstration will examine the development of a multi-band equaliser using these recent additions to Jamoma. This review of the design process will serve to highlight many of the benefits of MVC separation.

1. INTRODUCTION

Model-View-Controller (MVC) is an architecture pattern for developing interactive computer applications that breaks the application's design into three distinct elements [2]. A *model* represents a collection of data together with the methods necessary to process these data. The *view* provides an interface to the model for monitoring and interaction. The *controller* is the link between the model and view, and negotiates information between them. MVC enforces a clear separation between processes and their states, and how these are being represented to the user. This separation results in each concept being expressed in just one place, which in turn makes the code easier to write and maintain. The architecture also makes it possible to have multiple views for the same model. In this way, views can be customised and adapted dynamically based on the needs of the user at any one time, without these changes affecting the model itself.

Jamoma began as a system for developing high-level modules in the Cycling'74 Max environment [1]. The motivation was to address concerns about sharing and exchanging Max patchers in a modular system, and to leverage this structured environment for effective, efficient, and powerful means of automating and controlling Max patchers [3].

The upcoming version 0.6 of Jamoma enables MVC separation in Cycling'74 Max through custom externals and patching guidelines for developers [1]. The examples in [1] have been kept simple on order to focus on the core principles introduced. The following discussion expands on this by demonstrating how a more complex model and a set of views can be implemented. This will highlight many of the benefits of MVC separation when building performance systems in Max.

Key terminology will be introduced using *italics*, the name of Max externals will be **boldface**, and object arguments and attributes, as well as messages communicated to and from objects, will be denoted using `monospace`.

2. AN EQUALISER WITH MVC SEPARATION

A stereo multi-band equaliser combines individual filter bands, each with their own set of filter characteristics. In Max this is typically done using **filtergraph~** and a pair of **cascade~** objects. **filtergraph~** is a relatively complex external, and provides several parallel functions. It is a graphical widget for user interaction. It also maintains its state, as the number of bands and their filter characteristics can be saved with the patcher and output when the patcher loads. State handling can be further complicated by binding **filtergraph~** to **pattr** and **pattrstorage** for centralised management of presets, or by using dictionaries to set up an association with the content of a **filterdesign~** object. Finally **filtergraph~** also functions as a mapper between the higher-level representation of filter characteristics as frequency, gain and resonance (or slope) and low-level filter coefficients.

MVC separation will untwine these distinct functionalities, and make it easier to provide alternative user interfaces for the equaliser supplementing the standard interface provided by **filtergraph~**.

We direct the reader to [1] for details on the specific objects that enable MVC separation in Jamoma, and a detailed discussion of how to set up a model and design a view.

[1] http://www.cycling74.com. All URLs in this article were last accessed July 15th 2014.

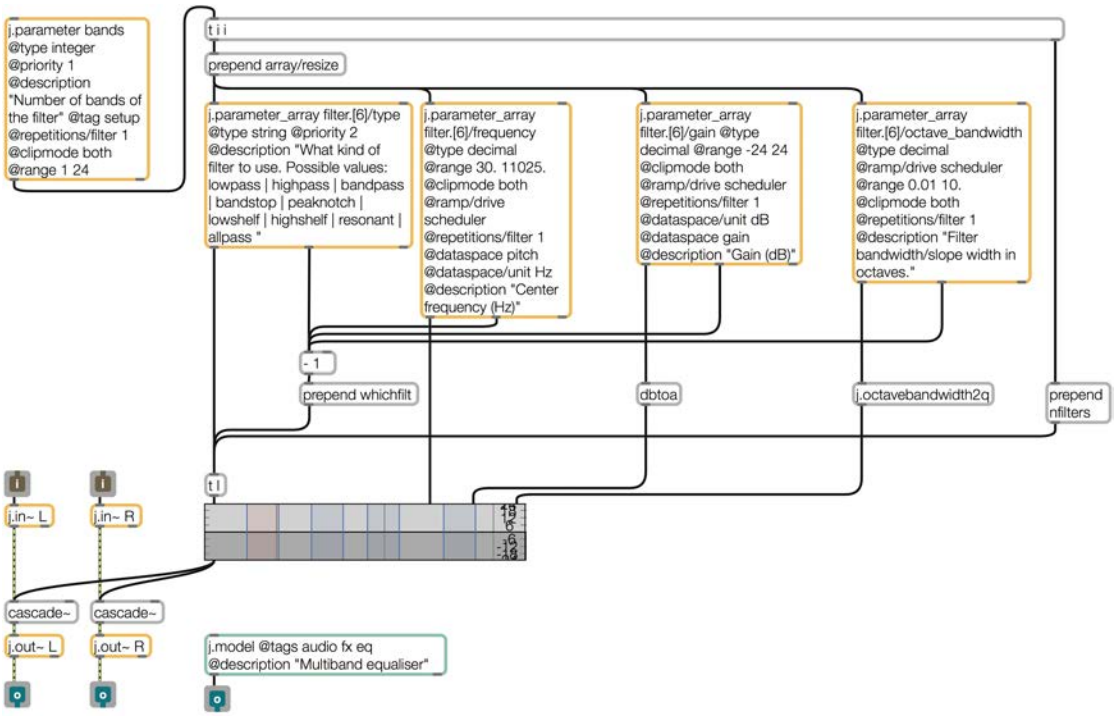

Figure 1. The equaliser model.

2.1 Setting up the equaliser model

The equaliser model can be seen in Figure 1. First, because we want it to be variable, the number of bands needs its own parameter. We then have four characteristics that define each filter band: Filter type, cutoff or center frequency, gain and bandwidth. Frequency and gain make use of the JamomaCore dataspace library [4], and this adds flexibility in terms of what data unit views can use when representing the values of these parameters. Filter bandwidth is expressed in octaves. **j.q2octave_bandwidth** and **j.octave_bandwidth2q** can be used to map between octave bandwidth and resonance or slope.

The filter characteristics are implemented as arrays using **j.parameter_array** as discussed in section 3.3 of [1], with arguments for parameter name such as `filter.[5]/gain`. This way the filters are represented as node instances `filter.1`, `filter.2`, etc., and the filter characteristics becomes subnodes of the respective instances. This results in a clear namespace for parameters, as illustrated in Figure 2.

```
bands 2
filter.1/type lowshelf
filter.1/frequency 75
filter.1/gain 0.000000
filter.1/octave_bandwidth 1.000000
filter.2/type peaknotch
filter.2/frequency 300
filter.2/gain 0.000000
filter.2/octave_bandwidth 1.000000
```

Figure 2. Excerpt of the parameter namespace for the equaliser model, with current values for the various parameters. The equaliser currently has two filter bands.

Whenever the number of bands changes, the `array/resize` message is sent to the various parameter arrays, causing them to dynamically create or dispose of array instances as needed. If a filter characteristic is changed, **j.parameter_array** will output what instance has been affected, as well as the new value. This is used to address the appropriate filter band in the **filtergraph~** below. This object is only being used here for mapping higher-level filter characteristics to low-level filter coefficients in preparation for **cascade~**. The instance of **filtergraph~** in the model should not be exposed to or used by the user for controlling the equaliser, as changes done in this object will not be reflected as updates to the current state of the model. Neither will they be reflected in any of the views.

When storing or recalling presets for the equaliser model, it is important to set the number of bands prior to the filter characteristics of each of the bands. For this reason, we have used the `@priority` attribute of **j.parameter**, as discussed in section 3.2 of [1]. In this model the number of bands is given first priority. Then, for each of the filter bands, `type` is given higher priority than the rest of the filter parameters, and the priorities are reflected in the ordering of the namespace in Figure 2.

2.2 Designing views for the equaliser

In the following subsections, several views will be designed for interacting with all or part of the equaliser model.

2.2.1 Displaying frequency response

The first view presented here displays the frequency response of the equaliser, but does not provide any means for changing equaliser settings. The view patcher can be seen in Edit Mode in Figure 3, and shares many patching solutions with the model in Figure 1. A **j.receive** object sub-

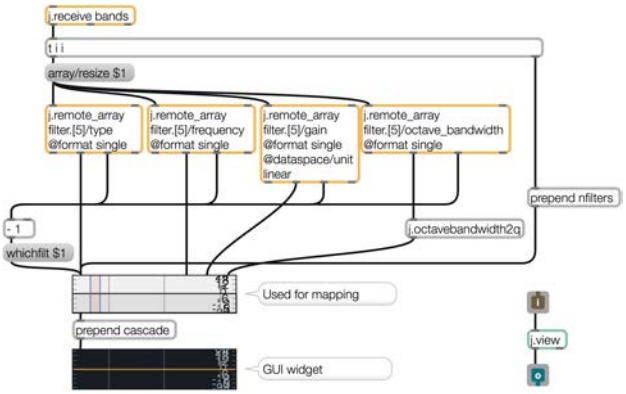

Figure 3. A view displaying equaliser frequency response.

scribes to the `bands` parameter in the model while a series of **j.remote_array** objects subscribe to the filter characteristics parameters for the array of filter bands. The view contains two instances of **filtergraph~**. The upper one is only used to map filter characteristics to low-level coefficients in a manner similar to what was found in the model, while the lower one provides the user interface of the view, and has its colour-scheme changed to resemble the look of Max for Live. When the patcher is in Presentation Mode, the lower **filtergraph~** object will be the only object in this view that is visible.

2.2.2 *Views interacting with one filter band only*

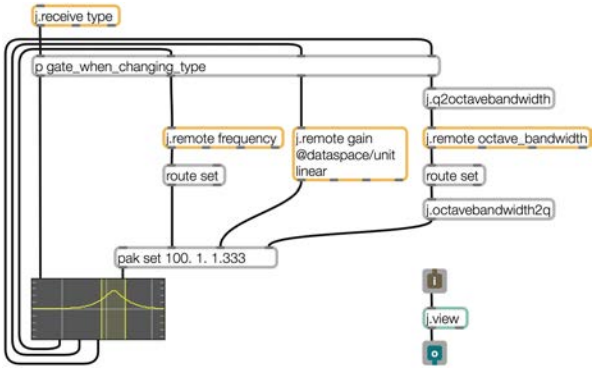

Figure 4. A view of one filter band only, with **filtergraph~** as GUI widget.

Views do not necessarily have to encompass all of the model. Figure 4 presents a view for the state of a single filter band. The view subscribes to the four parameters of the filter band, and connects them to a **filtergraph~** object. This **filtergraph~** object displays the current setting of the filter, and can also be used to change and update the filter parameters. The idiosyncrasies of the **filtergraph~** external make it impossible to change the filter type without causing it to output all filter characteristics whenever the filter type is changed. Changes to the filter type are likely to happen when changing which filter band is viewed, and introduce the risk of accidentally updating parameters for the wrong filter band during this transition. In order to avoid this a gate has been added (embedded in a subpatcher) that temporarily closes to prevent frequency, gain and bandwidth from being updated and changed in response to a change of filter type.

This patch also illustrates the benefits of relative addresses, as discussed in section 3.4 of [1]. The addresses provided for each of the **j.receive** and **j.remote** objects are relative to the address of the **j.view** object in the patch. If the user change what address within the node tree **j.view** subscribes to, this will propagate to all other controller objects in the patch. This way the user can dynamically change which filter band she wants to view and interact with.

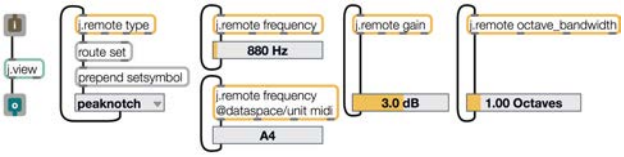

Figure 5. A view of one filter band only, displayed as numerical values.

Figure 5 presents a more basic alternative view for the same filter band, and uses number boxes to display the filter parameters. The filter type can be updated using a pop-up menu widget.

It is convenient to represent gain as linear amplitude in the two views presented in Figures 3 and 4. In Figure 5 frequency is expressed as *Hz* as well as midi note value. The Dataspace Library [4] enables **j.remote** to perform the necessary conversions. By default the `dataspace/unit` attribute of **j.remote** will be inherited from the associated **j.parameter** in the model.

2.2.3 *Combining and nesting views*

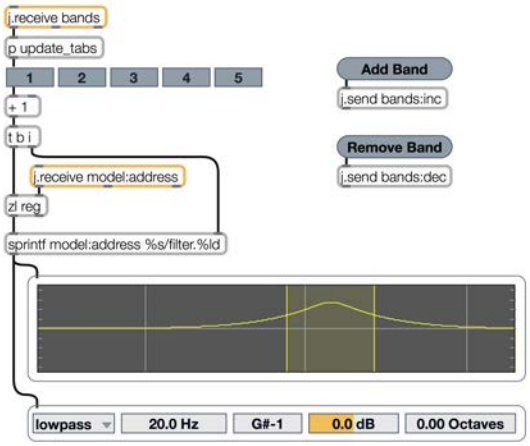

Figure 6. The single filter band views can co-exist, and can be dynamically repurposed to access all filter bands.

The lower part of the patcher in Figure 6 embeds both single filter band views from the previous subsection in separate **bpatcher** objects. The embedded views are now set to Presentation Mode, and hence only display the widgets intended for user interaction. Both views subscribe

to the same filter band of the model, and whenever a parameter value is changed in the model, both views will be notified. These views are dependent on the model, but the model does not depend on the views [5]. Because of this, the views can co-exist without any potential conflicts or synchronisation issues.

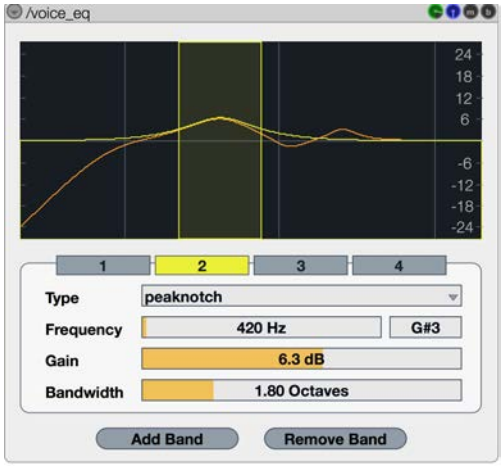

Figure 7. A combined view for the equaliser.

Finally, the view in Figure 7 pulls together all of the views presented so far to provide a complete interface to the multi-band equaliser model from section 2.1. The frequency response display view of section 2.2.1 forms the basis of this view. It has been extended with the two nested views accessing a single filter band that were introduced in section 2.2.2. Some refactoring has been done to gather the GUI widgets of the subviews in the top-level patcher window, while the controller objects (**j.view**, **j.remote**, **j.remote_array**, etc.) remain located in subpatchers. The **filtergraph~** object from the view in Figure 4 is now transparently overlaid on top of the **filtergraph~** object from Figure 3. **j.ui** provides a background for the view, while also offering access to the audio and preset amenities of the model, as discussed in sections 2.1 and 3.1 of [1].

The user can dynamically choose which filter band to view and control in detail, a feature that is achieved by the upper part of the patcher in Figure 6. When this section is added to the main view in Figure 7, the two **j.receieve** objects will be notified which equaliser model instance we are currently viewing, and how many filter bands it currently has. Whenever the value of the **tab** widget is changed, a `model:address` message is sent to the **j.view** objects in the two nested subviews, causing them to change their subscription to the proper filter band. Filter bands can be added or removed by addressing the `inc` and `dec` methods of the `bands` parameter, as described in [4].

3. DISCUSSION

Jamoma 0.6 enables MVC in Max as detailed in [1], and the presentation in section 2 illustrates how the dissociation of models and views makes it easier to design user interfaces. With the implementation of the observer pattern, each view gets updated when the model change, so that several views can co-exist [5]. In a similar way to how section 3.1 of [1] demonstrated the use of nested models, the view in Figure 7 illustrates the usefulness of nested views. Views can bind to models dynamically, and the use of relative addresses for controller objects within the view makes this straight-forward, as it is simply a matter of changing what model address **j.view** subscribes to. The single filter band views developed in section 2.2.2 illustrates how dynamic binding of views can be particularly useful when addressing an array of instances.

The need to refactor Jamoma for MVC separation has emerged out of the needs in the developers own artistic and research practises. During the alpha testing of Jamoma 0.6, it has been used in a number of large artistic projects in France and Norway in recent years, including works developed at GMEA, BEK, by The Baltazars and a new stage production currently in development by Verdensteatret. Jamoma 0.6 is scheduled for release during the summer of 2014, and requires OSX and Max 6.1 or newer [2] . It is licensed according to the "New BSD license", enabling it to be used in open-source and closed, commercial applications alike.

Acknowledgments

The development of MVC separation in Jamoma 0.6 has benefitted from a number of developer workshops hosted by iMAL, GMEA, BEK and fourMs lab, University of Oslo. Development has been supported by the French National Research Agency via the research projects Virage 2008-2010 and OSSIA 2012-2015, Hordaland County Council, Arts Council Norway and l'Arboretum. We would like to express our gratitude towards fellow Jamoma developers and all other artists, developers and researchers that we have consulted with in the process.

4. REFERENCES

[1] T. Lossius, T. de la Hogue, P. Baltazar, T. Place, N. Wolek, and J. Rabin, "Model-View-Controller separation in Max using Jamoma," in *Proc. of the joint ICMC SMC 2014 Conference*, Athens, Greece, 2014.

[2] T. Reenskaug, "Models - views - controllers," Technical Note, Xerox Parc, Tech. Rep., 1979. [Online]. Available: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[3] T. Place and T. 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.

[4] T. Place, T. Lossius, A. R. Jensenius, N. Peters, and P. Baltazar, "Addressing Classes by Differentiating Values and Properties in OSC," in *Proc. of the 2008 International Conference on New Interfaces for Musical Expression*, Genova, Italy, 2008, pp. 181–184.

[5] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[2] http://www.jamoma.org/download/