# Paradiddle: a code-free meta-GUI for musical performance with Pure Data

Adam T. Lindsay, Alan P. Parkes

Computing Department, Lancaster University
*email:* atl@comp.lancs.ac.uk

## Abstract

This paper describes a framework that utilizes existing tools to allow the simple creation of GUIs for musical performance, initially for Miller Puckette's Pd (Pure Data). Paradiddle leverages the Cocoa frameworks on Mac OS X to allow interface developers to create native GUIs with Apple's own Interface Builder tool, and without writing any computer code. We examine the requirements for such a system, touch upon the techniques for constraining developers in a way that they feel as if they are unconstrained, and examine the qualities of Cocoa and Pd that make this so easy to accomplish.

## 1 Introduction

Miller Puckette's Pd [1] is a powerful, open-source, cross-platform system for multimedia programming. It belongs to the same class of visual programming languages as Max/MSP and jMax, in which a program (called a "patch") is built up by visually connecting objects that perform functions to one another.

Pd's cross-platform nature, however, has led to a lowest-common-denominator graphical user interface. Programmed in Tcl/Tk, the GUI provides some interface elements, such as sliders and number-boxes, but as a whole, it does not integrate well with the computing environment. Its interface is ample for programming patches, but can be somewhat lacking when it comes to performance. The sliders and number boxes are somewhat alien to the native GUI of the host computer, and do not always include the same affordances as those offered by native UI widgets.

Joseph Sarlo [2] addressed the issue of creating a performance-oriented interface, in the form of GriPD, but as it also uses a cross-platform GUI, it does not fully take advantage of the native operating system's GUI, which can be a critical factor in leveraging the user's familiarity with an interface.

Apple's Mac OS X introduced far more than a new GUI; it brought the UNIX core and powerful frameworks from the NeXT operating system. The UNIX core allowed Miller Puckette to bring Pd to the Macintosh. The NeXTStep/OpenStep-derived frameworks, referred to as "Cocoa" for the purposes of this paper, brought a highly dynamic, object-oriented programming model to the Macintosh, including a very elegant tool for visually building GUIs [3]. The visual tool, Interface Builder (IB), allows one to drag widgets to a window, and graphically make connections amongst the widgets and between the widgets and the program code. It occurred to the authors that Pd's and IB's visual programming metaphors were highly compatible, and that users could easily transfer skills from one to the other. Paradiddle was conceived to bring the two worlds together.

## 2 Use example

It is best to see the capabilities of Paradiddle through a tutorial example presented to user-developers.

### 2.1 Tutorial

To start, double-click on the MainMenu.nib file that is included in the Paradiddle project template. This opens up Interface Builder. You should now see various windows, including a palette with different Cocoa controls, and a window (labeled "Window") with a pre-generated "Connect to Pd" button.

Drag a Cocoa control from the palette to the Application window. You can use a *Button*, *Slider*, *PopUpButton*, *ComboBox*, or *TextField*, but for now, we will use a slider with an oval handle (a continuous slider). Click for the "Cocoa-Other" Palette, and drag an oval handle slider to the Paradiddle application's window, as illustrated in figure 1.

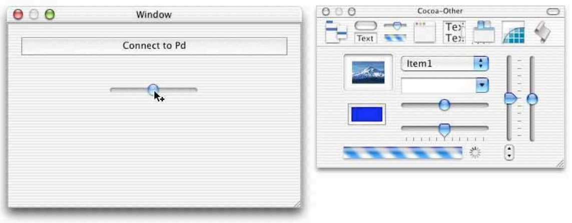

Figure 1: Drag a slider from the palette window to the application window.

Next, open the "Info" window in Interface Builder, and use the pop-up button in the Info window to select "Help". Select the slider control and label it using the "Tool Tip" field. A label of "foo_bar" will send messages to a receiver called "foo_bar" in your patch.

Now you must connect the controller to Paradiddle framework. Make sure the controller is selected, and control-drag from the controller to the cube icon labeled "PDController" in the MainMenu.nib window. When you release the mouse over the PDController, the Info window changes to show "Connections." The "Outlet" pane should have "Target" selected with two choices. Click on "SendToPd:" and then click the "Connect" button at the bottom of the Info window, as illustrated in figure 2.

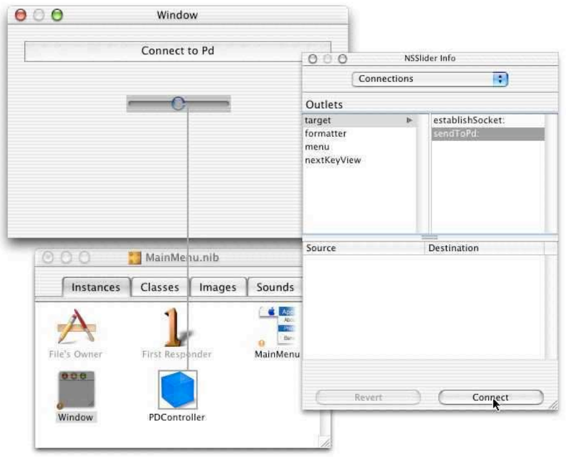

Figure 2: Connect the slider to the Paradiddle controller and its main method.

After this step you should be able to save the patch, return to Project Builder, and click on the "Build" icon in Project Builder. The project will link your interface with the framework, and you can run the resulting application with Pd receiving "foo_bar *number*;" messages.

### 2.2 Patch design

Paradiddle requires little in the way of modification to Pd patches in order to be made compatible. The biggest issue is making sure that interface elements that need to be accessible for performance are set to receive messages. Several recent objects graphical control objects within Pd, like the *Hslider*, the *Vradio*, and the *Toggle*, do this for 'free'; one only needs to assign it a name as a receiver in the object's preferences. By using these objects that offer a degree of performance-oriented control, the user-developer only needs to give identifying names to the control elements of the patch. As organizing and identifying inputs to a system is a good design principle anyway, it is not seen as a great burden to place on the user-developer.

Once the patch is designed to receive messages, one only need to make sure the Pd program is listening on a socket for input. The easiest way to do this is with a *netreceive* object, identifying the port to listen on, whether to use udp or tcp packets, and to automatically dispatch messages to receivers in the patch. The only other issue to consider in making an internally consistent patch is ensuring that the initial state of the patch is in synch with the initial state of the GUI.

## 3 Paradiddle

In the process of designing Paradiddle, we had a number of goals:

- Keep the tool usable to people who don't consider themselves programmers,
- Keep the visual metaphor, maximizing the amount of visual interaction with the tool,
- Don't require any computer code to be written, and
- Make as many of the Cocoa GUI elements available to control Pd as possible.

The design was essentially for two levels of user, the user-developer who would use the Paradiddle tool to make a GUI for their patches, and for an end user, who may not necessarily be the person programming the musical tool in Pd. This second user needed to be able to perform the patch using a familiar interface without knowing the internal workings of the patch.

These goals defined the project, and made it of reasonable scope, leading to a quick initial implementation, but leaving plenty of opportunity for further refinement and growth.

### 3.1 Design

No small part of Pd's power comes from the simplicity of the way it interfaces with the outside world. It connects to both the GUI and other computers with network sockets, and uses a very simple protocol, FUDI, for passing messages. Messages are in ASCII, with the recipient as the first word, followed by a space and the rest of the message, which is terminated with a semicolon and a carriage return. It was quickly apparent that Paradiddle would have to act primarily as a network "shim" between the GUI and the Pd client program that synthesizes music.

This shim was to take the form of a Cocoa framework (self-contained library, capable of containing additional resource and configuration files) that would act as an "engine" underneath the user-developer's GUI. The user-developer merely needs to link their graphically-generated GUI to the framework to create an application.

Although code was to be avoided, there were a few parameters that had to be controllable by the user-programmer. A multi-level preference system using XML format files was chosen. The user-developer could interact with the raw XML by hand-replacing certain parameters, or they could use an Apple-provided graphical tool for editing these property list ('plist') files. Defaults across all applications are held in an identically-formatted file that is kept with the Paradiddle framework.

The actual performance of the application hinges on Cocoa's dynamic message-passing. A Cocoa object need not know what other objects send it a message until it receives one. At the time of application launch, Paradiddle reads the XML preference files and decides what to do. The initialization code determines the destination and port for the socket to Pd, opens it, and prepares it for writing.

We can take further advantage of this dynamism by having a single object and a single method as the destination for all GUI elements. A GUI element sends a SendToPd: message to signify a change. Our SendToPd: method queries the object as to its name (which we take as the Pd destination object), and its type. By knowing the widget's type, we decide on the datatype to output, and how exactly to query the widget for its value. We send the simple message via the socket to Pd, and wait for another message.

### 3.2 Packaging

One key to ease-of-use on the Macintosh platform is the presentation and packaging of programs. If one is to provide a system that claims ease-of use, one must not ignore that aspect. In the case of Paradiddle, it is presented as a framework, as already discussed, and as a template in Project Builder (Apple's integrated development environment). The user-developer merely needs to open Project Builder, select the Paradiddle template, and start customizing it.

## 4 Analysis

### 4.1 Pleasant feature interactions

The most striking thing about Paradiddle is the number of features that we get 'for free' from the underlying Cocoa and Pd systems. A product of working to accommodate Pd's socket-oriented interface is that one only needs to change Paradiddle's recipient's address from *localhost* to *mycomputer.mydomain.edu* to get a network remote control. Pd could be running on another computer with a different architecture, and Paradiddle could control it from a Mac OS X computer. A side effect that comes from the Macintosh side is the recent introduction of Rendezvous (Apple's implementation of the ZeroConf standard) allows one to name a local address (e.g., *othermac.local.*) and connect to the named computer without needing an intervening DNS server.

Pd's extremely straightforward messaging architecture allows user-developers to interact with Pd directly. For example, one can incorporate a control to turn Pd's audio-rate processing on and off by simply making a toggle button with the label *pd dsp*. Toggling the button to an on state results in the message "pd dsp 1;" which turns on audio processing.

The rest of the powerful Cocoa framework can be used by Paradiddle, including subclassing existing widgets, using images and icons, resizable layouts, and much more sophisticated programming, if need be.

### 4.2 Simplicity and generality

It is most instructive to ask why this system should work with such a minimum of code. The clearest answer to the authors lies in the dynamic messaging of Cocoa. User-programmers can impart all the information needed by the underlying engine by working with get-info boxes and typing labels into fields. That information is queried only at run-time, so there is no coding or compilation step needed to integrate that information with the inner workings of Paradiddle.

The other factor working in Paradiddle's favor is the fact that it sets up working conventions, and uses them as constraints on the user-developer's behavior. These constraints are not seen as such by this user, because they define the capabilities of the program: the configuration options and the number of Cocoa GUI widgets available. The range of behavior allowable using the XML configuration files defines the features of the system available to the user-developer. The number of available Cocoa widgets is the only constraint, and since it exceeds the number of Pd-native widgets, it is not a limitation at all. It is possible to support all of these widgets because Cocoa's elegant type hierarchy makes the problem tractable. These two dimensions map out the behavior of the program that requires a general meta-programming approach. Allowing for an additional degree of freedom on the part of the user-developer may step beyond the bounds of feasibility.

## 5 Conclusions

### 5.1 Future work

As of this writing, there are a variety of features that are planned to be incorporated into Paradiddle. The first one to come is the automatic launching of Pd, as mediated by the XML configuration file. The user-developer identifies the Pd executable (which may be incorporated into the application bundle itself, making for a stand-alone application), its run-time options and the patch to open, and Paradiddle can launch it on startup.

There is nothing intrinsic about Paradiddle that ties it to Pd and its communication protocol. It can be extended to other network remote controls for musical performance such as Open Sound Control (OSC) [4]. OSC has been ported to numerous platforms, synthesis programs, and APIs, and therefore makes an ideal candidate for increasing Paradiddle's utility.

The most obvious feature to add to Paradiddle is to make it a message receiver as well as a message sender. The greatest barrier to implementation is not technical but designing the interface to the user-developer. In order to suppress control feedback loops (potentially bringing controls to a standstill), we must provide a means of giving widgets different message input labels from their output labels. Interface builder provides no easy way of doing this, so alternative methods of identification must be found.

### 5.2 Summary

In this paper we discussed a framework that leverages as much as possible from the underlying Pd and Mac OS X Cocoa systems to make it possible for users to develop native interfaces for performance. There is great stress placed on simplicity of use for the user-developer, and is designed in a way to maximize flexibility without requiring any additional code to be written. This is achieved by using Cocoa's dynamic messaging and pre-formatted configuration files. By using one simple message dispatch method, a large range of control messages can be sent to a client patch. The Paradiddle framework is simple and extensible, and shows good potential for incorporating extra features.

At the time of writing, Paradiddle is distributed without any source, mostly for the purpose of making the point that no adjustments to source code are needed in order to get it to work. By the time of publication, it is expected that the full source code for the project will be available to all. For more details and to download, see:

http://homepage.mac.com/atl/pd/paradiddle.html

## 6 Acknowledgments

The authors wishes to thank Miller Puckette for making Pd so freely and openly available, and the Pd community for being exemplary in its generosity, supportiveness, and enthusiasm. The primary author also wishes to thank the computing department at Lancaster University, for affording him the freedom to pursue projects such as this, and Adrian Friday, who offered valuable feedback on short notice.

## References

[1] Puckette, M. 1996. "Pure Data: another integrated computer music environment." Proceedings, International Computer Music Conference. San Francisco: International Computer Music Association, pp. 269-272.

[2] Sarlo, J.A. 2003, "GriPD: A Graphical Interface Editing Tool and Run-time Environment for Pure Data." Submitted to ICMC2003. http://crca.ucsd.edu/~jsarlo/gripd/

[3] Apple Computer, Inc. http://developer.apple.com/

[4] Wright, M. and Freed, A. 1997. "Open SoundControl: A New Protocol for Communicating with Sound Synthesizers." International Computer Music Conference, 1997.