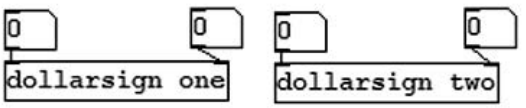
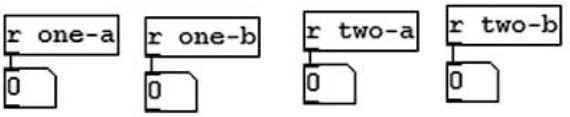
You can use dollarsigns in abstractions to get local sends and receives as shown here.



Open both copies to see what's happening...



updated for Pd version 0.34

What it takes to be a RRADical

Frank Barknecht

The acronym RRAD means "Reusable and Rapid Audio Development" or "Reusable and Rapid Application Development". The Pd patches that form RRADical are intended to solve real-world problems on a higher level of abstraction than the standard Pd objects. Where suitable, these high level abstractions should have a graphical user interface (GUI) built in. As I am more focused on sound production, the currently available RRADical patches mirror my preferences and mainly deal with audio, although the basic concepts also apply to graphics and video work using, for example, the Gem and PDP extensions of Pd.

Pre-fabricated high-level abstractions may not only make Pd easier to use for beginners, they also can spare lot of tedious, repetitive patching work. For example, building a filter using the [lop~] object of Pd usually involves some way of changing the cutoff frequency of the filter. So another object like a slider will have to be created and connected to the [lop~]. The connections between the filter's cutoff control and the filter can also be done in advance inside a so-called abstraction: in a Pd patch file saved on disk. If the graph-on-parent feature of Pd is used, the cutoff slider can be visible when using that abstraction in another patch. The new filter abstraction now carries its own GUI and is immediately ready to be used.

In my experience a lack of these kinds of abstractions in the Pd distribution is one big hurdle for new users. This is reflected in typical questions asked on the Pd mailing lists: Does someone know a good drum machine for Pd? Does anyone have a sequencer? One reason for the success of Reaktor, surely, is the availability of a lot of usable example patches. Reason takes this to the extreme in that it practically only consists of "example patches" and does not allow for building custom modules at all.

Problems and Solutions

In the course of designing RRADical as such a modularized system of high-level abstractions, several problems had to be solved. Two key areas are:

Persistence
: How to save the current state of a patch? How to save more than one state (state sequencing)?

Communication
: The various modules are building blocks for a larger application. How should they talk to each other?

It turned out that it is possible to solve both tasks in a consistent way using a unique abstraction. But first let's look a bit deeper at the two problems.

Persistence

Pd offers no direct way to store the current state of a patch. Here's what Pd author Miller S. Puckette writes about this in the Pd manual in the section "2.6.2. persistence of data":

> Among the design principles of Pd is that patches should be printable, in the sense that the appearance of a patch should fully determine its functionality. For this reason, if messages received by an object change its action, since the changes aren't reflected in the object's appearance, they are not saved as part of the file which specifies the patch and will be forgotten when the patch is reloaded.

Still, in a musician's practice some kind of persistence turns out to be an important feature, which many Pd beginners do miss. And as soon as a patch starts to use lots of graphical control objects, users will – and should – play around with different settings until they find some combinations they like. But unless a way to save this combination for later use is found, all this is temporary and it is gone as soon as the patch is closed.

For RRADical I chose Thomas Grill's [pool] external to handle persistence. Basically, [pool] offers something that is standard in many programming languages: a data structure that stores key-value pairs. This structure is also known as hash, dictionary or map. With [pool] these pairs also can be stored in hierarchies, and they can be saved to or loaded from disk. Several [pool]s can be shared across patch borders by giving them the same name. The [pool] object in RRADical patches is hidden behind an abstracted "API", so it could be replaced by some other object in the future.

Communication

Along with persistence, it also is important to create a common path through which the RRADical modules will talk to each other. Generally, the modules will have to use what Pd offers them, and that is either a direct connection through patch cords or the indirect use of the send/receive mechanism in Pd. Patch cords are fine, but tend to clutter the interface. Sends and receives on the other hand will have to make sure that no name clashes occur. A name clash is when one target receives messages not intended for it. For a "library" like RRADical it is crucial that senders in RRADical abstractions use only local names with as few exceptions as possible. This is achieved by prepending almost all RRADical senders with the string "$0-", that is replaced with a number unique to a certain instance of an abstraction.

Still we will want to control a lot of parameters and do so not only through the GUI elements Pd offers, but probably also in other ways, for example through hardware MIDI controllers, through some kind of score on disk, through satellite navigation receivers, etc.

This creates a fundamental conflict:

We want borders
: We want to separate our abstractions so they do not conflict with one other.

We want border-crossings
: We want to have a way to reach their many internals and control them from outside.

The RRADical approach solves both requirements in that it enforces a strict border around abstractions, but drills a single hole in it: the **OSC-inlet**. This idea is the result of a discussion on the Pd mailing list and goes back to suggestions by *Eric Skogen* and *Ben Bogart*. Every RRADical patch has (must have) a rightmost inlet that accepts messages formatted according to the OSC protocol. OSC stands for *Open Sound Control* and is a network transparent system to control (audio) applications remotely. It has been mainly developed at CNMAT in Berkeley by Matt Wright.

OSC can control many parameters over a single communication path (like a network connection using a definite port can transport a lot of different data.)

The OSC-inlet of every RRADical patch is intended as the border-crossing: everything the author of a certain patch intends to be controlled from the outside can be controlled by OSC messages to the OSC-inlet. The OSC-inlet is strongly recommended to be the rightmost inlet of an abstraction. All of my RRADical patches follow this guideline.

Trying to Remember it All: Memento

To realize the functionality requirements laid out so far, I resorted to a design pattern called Memento. In software development Mementos are quite common. Computer science literature describes them in great detail, for example in the Gang-Of-Four book "Design Patterns" (Gamma, E. et al. 1995). To make the best use of a Memento, science recommends an approach where certain tasks are in the responsibility of certain independent players.

The Memento itself is a kind of state record. A module called the "Originator" is responsible for creating this state and managing changes in it. The actual persistence, that could be the saving of a state to hard disk, but could just as well be an upload to a web server or a CVS check-in, is done by someone called the "Caretaker" in the relevant literature. The Caretaker only has to take care that the Mementos are in a safe place and no one else fiddles with them. It does not deal with changing the content of a state.

Memento in Pd

I developed a set of abstractions that follow this design pattern. Memento for Pd includes a [caretaker] and an [originator] abstraction, plus a third one called [commun] which is just a thin extension of [originator] and should be considered part of it. In fact, a soon to be published new version of [originator] seeks to get rid of the [commun] objects and replaces them with special messages to the [originator].

Due to the constraints of the scope of this text, I have to omit a detailed explanation of how the objects work in practice and would like to encourage you to read through the help patches that accompany RRADical and Memento. They are free.

Using Memento it is possible to "freeze" the whole state of a patch to a file on disk. A state here not only consists of the current position of sliders etc., but also includes "possible states", that is, for example, the position of a slider at a different part of a piece of music. Through OSC it is possible to select which of the "possible states" should become the next active state. A way to interpolate between states, to use weighted sums of various possible states, is currently being researched, however, I must admit that this is indeed a tricky problem to solve inside the infrastructure that Memento provides.

Putting it all to RRADical Use

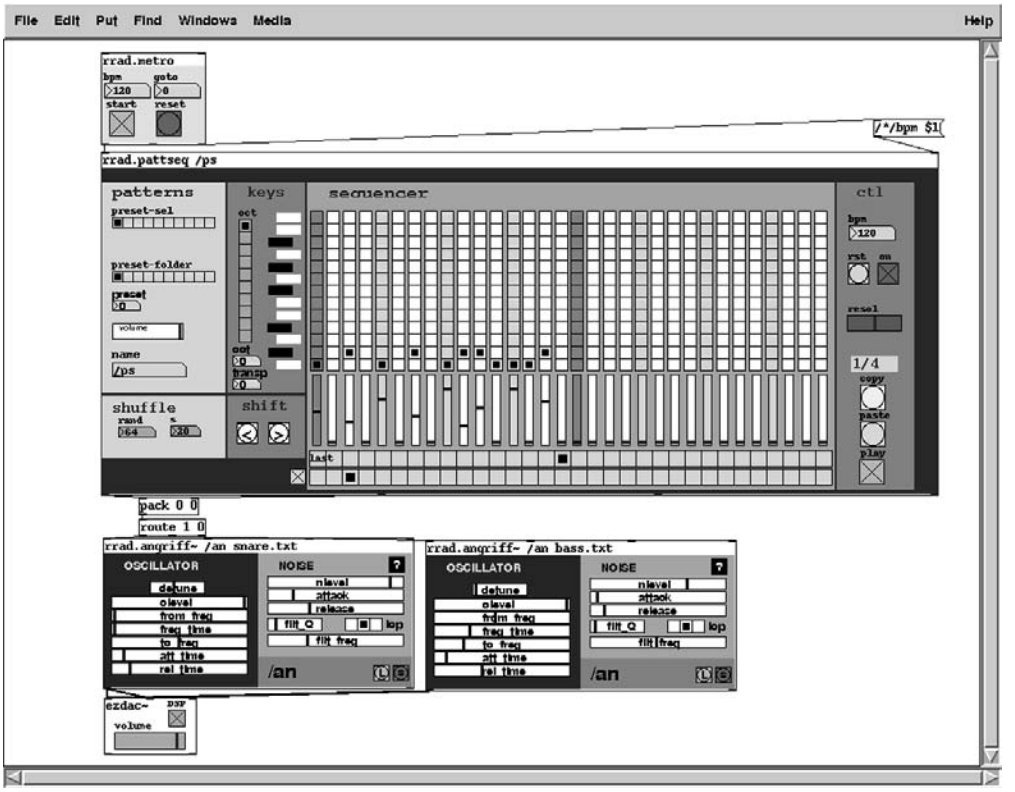
I developed a growing number of patches that follow the RRADical paradigm, among these are a complex pattern sequencer, some synths and effects and more. All of these are available in the Pure Data CVS, which currently lives at *pure-data.sourceforge.net* in the directory "abstractions/rradical".

The RRADical Patches using Memento have two main characteristics:

Rapidity
: Ready-to-use high-level abstraction can save a lot of time when building larger patches. Clear communication paths will let you think faster and devote more attention to the really important things.

Reusability
: Don't reinvent the wheel all the time. Reuse patches like instruments for more than one piece by just exchanging the Caretaker-file used.

An example of how the patches can be used is shown in the following screenshot:



Here you see a step sequencer module called [rrad.pattseq] which drives two drum synths, [rrad.angriff]. A patch like this can be made very quickly even as a Pd beginner, and it provides instant gratification.

Much, but not all is well yet

Developing patches using the Memento system and the design guidelines presented here has had a remarkable impact on how my patches are designed. Before Memento, quite a bit of my patches' content dealt with saving a state in various, crude and non-unified ways. I even tried to avoid saving states at all, because it always seemed to be too complicated to bother with it. This limited my patches to being used in improvisational pieces without the possibility of preparing parts of a musical story in advance and "designing" those pieces. It was like being forced to write a book without having access to a sheet of paper (or a hard disk nowadays). This has changed: having "paper" in great supply has now made it possible to "write" pieces of art, to "remember" what was good and what should preferably not be repeated, to really "work" on a certain project over a longer time.

RRADical patches also have proven to be useful tools in teaching Pure Data, which is important as the usage of Pd in workshops and at universities is growing – also thanks to its availability as Free Software. RRADical patches can be used directly by novices, as they are created just like any other patch, but they already provide sound creation and GUI elements that the students can use immediately to create more satisfactory sounds than the sine waves usually taken as standard examples in basic Pd tutorials. With greater proficiency the students can later dive into the internals of a RRADical patch to see what is inside and how it was done. This allows a new top-down approach in teaching Pd, which is a great complement (or even alternative) to the traditional, bottom-up way.

The use of OSC throughout the RRADical patches created another interesting possibility: collaboration. As every RRADical patch can not only be controlled through OSC, but can also control another patch of its own kind, the same patch could be used on two or more machines, and every change on one machine would propagate to all the other machines where that same patch is running. So jamming together and even the concept of a "Pd band" is naturally built into every RRADical patch.

References

Gamma, E. / Helm, E. / Johnson, R. / Vlissides, E. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1995.