Using Pd as a score language *

Miller Puckette
University of California, San Diego
msp@ucsd.edu — http://crca.ucsd.edu

July 27, 2020

1 Introduction

The original idea in developing Pd [Puc97] was to make a real-time computer music performance environment like Max, but somehow to include also a facility for making computer music scores with user-specifiable graphical representations. This idea has important precedents in Animal [LdC91] and SSSP [B$^+$85]. An even earlier class of precedents lies in the rich variety of paper scores for electronic music before it became practical to offer a computer-based score editor. In this context, scores by Stockhausen (*Kontakte*, *Studie* II) and Yuasa (*Toward the Midnight Sun*) come most prominently to mind, but also Xenakis's *Mycenae*-α, which, although it was realized using a computer, was scored on paper and only afterward laboriously transcribed into the computer.

Pd is designed to to offer an extremely unstructured environment for describing data structures and their graphical appearance. The underlying idea is to allow the user to display any kind of data he or she wants to, associating it in any way with the display. To accomplish this Pd introduces a graphical data structure, somewhat like a data structure out of the C programming language, but with a facility for attaching shapes and colors to the data, so that the user can visualize and/or edit it. The data itself can be edited from scratch or can be imported from files, generated algorithmically, or derived from analyses of incoming sounds or other data streams.

*Reprinted here from *Proceedings*, ICMC 2002, pp. 184-187.

Figure 1 shows one simple example of a very short musical sketch realized using Pd. The example, which only lasts a few seconds, is a polyphonic collection of time-varying noise bands. The graphical "score" consists of six objects, each having a small grab point at left, a black shape to show dynamic, and a colored shape to show changing frequency and bandwidth. The horizontal axis represents time and the vertical axis, frequency (although, as explained later, this behavior isn't built into pd). The dynamic and frequency shapes aren't constrained to be connected or even to be proximate, but since they pertain to the same sound their horizontal positions line up. In this example the last (furthest-right) object is percussive (as seen by the black shape) and has a fixed frequency and bandwidth, whereas the large, articulated shape in the center has a complicated trajectory in both frequency and dynamic. The color of the frequency trace determines the voice number used to realize it.

Each object is thus composed of a combination of scalar values (color; aggregate position in X and Y coordinates) and array values (time/value pairs for the black traces and time/frequency/bandwidth triples for the colored ones.) This is all specified by the user using Pd's "template" mechanism.

Figure 2 shows the template associated with the graphical objects shown in Fig. 1. Templates consist of a data structure definition (the `struct` object) and zero or more drawing instructions (`filledpolygon` and `plot`). The `struct` object gives the template the name,

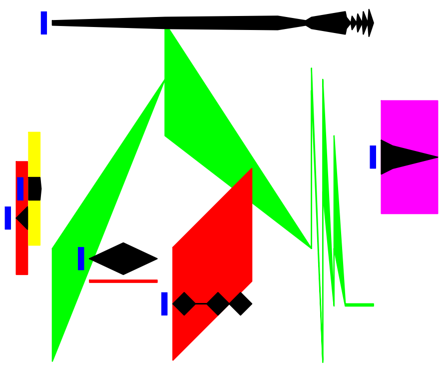

Figure 1: A simple score in Pd

```
struct template-toplevel float x float y float voiceno
array pitch template-pitch array amp template-amp

filledpolygon 9 9 0 0 -10 0 10 5 10 5 -10

plot pitch voiceno 3 10 0

plot amp 0 3 10 0
```

Figure 2: Template for objects in the score of Fig. 1

"template-toplevel." The data structure is defined to contain three floating point numbers named "x", "y", and "voiceno," and two arrays, one named "pitch" whose elements belong to another template named "template-pitch," and similarly for the array "amp."

In general, data structures are built from four data types: scalar floats and symbols, arrays (whose elements share another, specified template) and lists (whose elements may have a variety of templates). The contents of a Pd window themselves form a list. Pd's correlate of Max's "table" object is implemented as a top-level array whose elements are scalars containing a single floating-point number.

Data structures in Pd may nest arbitrarily deeply using the array and list types. For example, a collection of sinusoidal tracks from an analysis engine could be implemented as an array of arrays of (pitch, amplitude) pairs; this appears as example 12 in Pd's FFT object online tutorial.

After the `struct` object in the template of Fig. 2, the remaining three objects are *drawing instructions* , first for a rectangle (`filledpolygon`), and then for two arrays. The various graphical attributes that are specified for drawing instructions may be numerical constants or data structure field names; in the latter case the value varies depending on the data. For instance, the second creation argument to `plot`" is the color. The first `plot` plots the "amp" field and the color is given as 0, or black. The second one plots "pitch" using the color "voiceno". In this way the color of the second trace is attached to the "voiceno" slot in the data structure, so that color will vary according to its "voiceno" slot.

2 Traversal

Pd objects are provided to traverse lists and arrays, and to address elements of data structures for getting and setting. Figure 3 demonstrates how these facilities could be used, for example, to sequence the score shown in Fig. 1.

Pd has no built-in sequencer, nor even any notion that "x" values should be used as a time axis. (However, a "sort" function is provided, which reorders a list from left to right, on the assumption that users might often want to use Pd data collections as x-ordered sequences.) Recording sequences of events into lists, and/or playing the lists back as sequences, are functionalities that the user is expected to supply on top of Pd's offerings, which, it is hoped, would allow those functionalities within a much larger range of possibilities, to include random reorderings of events, score following, self-modifying scores, reactive improvisation, and perhaps much more.

Traversal of data is made possible by adding a new type of atom, "pointer", to the two previously defined types that make up messages, to wit, numbers and symbols. Unlike numbers and symbols, pointers have no printed form and thus can't be uttered in message

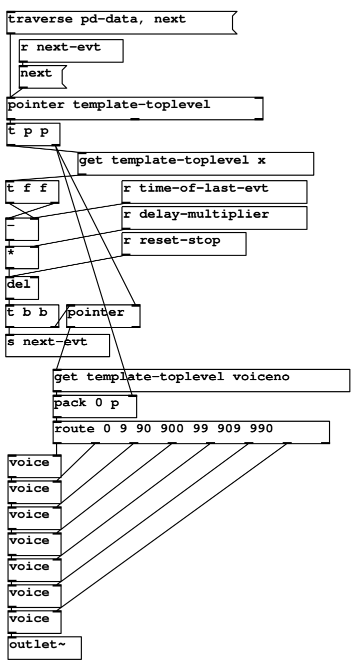

Figure 3: Traversing the list of objects

boxes. Traversal objects such as `pointer` and `get`” (among several others) can generate or use pointers. The pointer data type is also integrated into pipe-fitting objects such as `pack`, `unpack`, and `route`.

In Fig. 3, the topmost `pointer` object holds a pointer to the next object to “play” (by sending it to one of the `voice` abstractions at bottom.) The pointer object takes a `traverse` message to set it to the head of the list (named “pd-data”), and `next` messages to move to (and output) the next datum in the list (i.e., the next in the list of six objects in the score). Another `pointer` object is also used, further down, as a storage cell for pointers just as `float` is for numbers.

The center of any sequencer is always the `delay` object, which must be fed the time difference between each event (including the non-event of hitting `start`) and the next. As we extract each of the six objects in the score, we must wait the delay for playing that object, and then send its pointer to one of the `voice` abstractions to play it. However, we have to inspect the object itself to know the delay before playing it. So, in the loop, we peel off the first remaining object to play and inspect the time difference between it and the previous one, using this value to set the delay, but also storing the pointer in the lower `pointer` and `pack` objects.

The time difference needed to set the delay object is obtained using the
“`get template-toplevel x`” object. (This is converted to incremental time (“`-`”), corrected for tempo, and fed to the delay.) Pd provides the `get` and `set` objects for reading and writing values from data structures. The two `get` objects shown here obtain the `x` and `voiceno` fields of the current object. The template name (template-toplevel) is supplied to the `get` objects so that they can look up the offset of the necessary field(s) in advance, for greater run-time efficiency.

Once the delay has expired, the object’s pointer is recalled (the lower `pointer` object), and the voice number is recalled. This is packed with the pointer itself and routed, so that the pointer goes to the appropriate voice. The voice number is shown as the color of the frequency trace in "999" units (first digit red, second green, third blue) and the `route` is arbitrarily set up to select among the six primary and secondary colors plus black.

The details of extracting the pitch and dynamic breakpoints from the arrays defined in the template are managed in the `voice` abstraction. The `voice` abstraction receives a pointer to a given object and manages the sequencing of the arrays; so it contains two sequencers itself. The nesting of the overall structure of the sequencer patch mirrors the nesting of the original data structures. Finally, the voice abstraction puts its audio output on a summing bus.

More general patches can easily be constructed which access heterogeneous lists of objects (having different templates). In this way, an arbitrarily rich personal "score language" can be developed and sequenced.

3 Accessing and changing data

In general, accessing or changing data is done via `pointers` to `scalars`. Numbers and symbols withing scalars are accessed using the `get` object and changed, in the same way, using `set`. Since lists and arrays are composed of scalars, every actual number or symbol in a data heap will be a number or symbol element of some scalar. To access them, it suffices to have objects to chase down elements of lists and arrays (given either a global name or a pointer to the containing scalar).

Lists are traversed in the way shown above; to get to a sublist of a scalar, the `get` object will provide a pointer, in the same way as it provides `float` or `symbol` elements of scalars. For arrays, an `element` object is provided which, given a scalar, a field name and a number, chases down the numbered, scalar, element of the named array field.

To alter `float` or `symbol` elements of scalars is straightforward using the `set` object, but arrays and lists can't be set by assignment; there is no suitable data type available withing messages. Lists could possibly be "settable" by passing pointers to other lists, but permitting this would have required either automatically doing deep copies of data structures to carry out the assignments, or else implementing a garbage collecting memory management system, either of which would be difficult to realize within realtime computation time constraints. Instead, all the data hanging from a scalar is considered as belonging to that scalar, and is left in memory until the scalar is deleted; the data may be changed atom by atom, but primitives are not provided which would imply unpredictable execution times.

The `getsize` and `setsize` objects are provided to access or change the number of elements in the array. For lists, an `append` object appends a new scalar for a given template to a list, after the element pointed to. (To insert a scalar at the beginning of a list, the pointer can be set to the "head" of the list, a formal location before the first list item.) Deletion is less flexible; the only operation is to delete an entire list. (There's no reason not to provide finer-grain deletion mechanisms except that it's not clear how to protect against stale pointers efficiently, except by voiding the entire collection of pointers into a list.)

4 Editing

The score of Fig. 1 can be edited by dragging breakpoints, or by adding and deleting them, using mouse clicks. Also, entire objects or collections of them may be copied, pasted, and dragged around the screen. Alternatively, there is an editable (or computer generate-able or parse-able) text representation for the data, which may be seen or changed in a dialog window or read and written to external text files.

Since the graphical presentation of data objects is determined by drawing instructions, the drawing instructions are interpreted backwards to alter data as a result of mouse operations. If a given graphical dimension is controlled by a variable, that variable is then controlled by dragging along that dimension; if the dimension is constant, it can't be altered by dragging.

Tricky situations can arise when the user changes the contents of templates. A change in drawing instructions can be accommodated by simply tracking down and redrawing all data objects using the template. However, changing the `struct` object itself make for less straightforward situations. The user might wish to reorder fields, delete them, add new ones, or rename them. When a `struct` object changes, Pd automatically conforms the data from the old structure to the new one. Fields with the same name as previously are maintained (reordering them as necessary); and if a field disappears but another of the same type appears, the new one(s) are taken to be renamings of the old one(s) in order of appearance. New fields which cannot be matched in this way with previously existing ones are assumed to be new and are initialized.

It can happen that two `struct` objects compete to define the same data structure, or that the user reads in data from a file which expects a different version of the structure, or alternatively, that the `struct` object for existing data objects disappears. For this reasn, Pd maintains a private representation of the last active version of a `struct` until all similarly named "structs," as well as all data using that `struct`, have disappeared. If the user introduces a new version of the `struct` and only later deletes the "current" one, the data is only conformed to the new version once the old one is deleted. In this way we avoid getting into situations where data is left hanging without its structure definition, or where data ends up belonging to two or more structures of the same name. The worst that can happen is that data may lose their drawing instructions, in which case Pd supplies a simple default shape.

5 More work

When examples get more complicated and/or dense than the one shown here, it becomes difficult to see and select specific features of a data collection; more work is needed to facilitate this. There should be some facility for turning drawing instructions on and off, or perhaps for switching between versions of a template, depending on the user's desired view. There should also be a callback facility in the template for when an object is edited with the mouse, so that the user can bind actions to mouse clicks.

More generally, the collection of traversal objects that Pd provides is adequate to support a variety of modes of data collection and use, such as analysis and sequencing. But the patches required to traverse the data collections are not always simple. It would be desirable to find a more straightforward mechanism than that provided by the `pointer`, `get` and `set` objects.

The "data" facility, although part of the original plan for Pd, has only recently been implemented in its current form, and as (hopefully) the user base grows there will surely be occasions for many further extensions of the data handling primitives and the graphical presentation and editing functions.

References

[B+85] William Buxton et al. The evolution of the sssp score-editing tools. In Curtis Roads and John Strawn, editors, *Foundations of Computer Music*, pages 376–402. MIT Press, Cambridge, 1985.

[LdC91] Eric Lindemann and Maurizio de Cecco. Animal—a rapid prototyping environment for computer music systems. *Computer Music Journal*, 15(3):78–100, 1991.

[Puc97] Miller S. Puckette. Pure data. In *Proceedings of the International Computer Music Conference*, pages 224–227, Ann Arbor, 1997. International Computer Music Association.