MORE ON ARRAYS

Arrays have methods to set their values explicitly; to set their "bounds" rectangles, to rename them (but if you have two with the same name this won't necessarily do what you want) and to add markings. To set values by message, send a list whise first element gives the index to start at. The second example sets two values starting at index three. Indices count up from zero.

```
;
array98 0 -1 1 -1 1 -1 1 -1 1 -1
```

```
;
array99 3 -0.5 0.5
```

renaming an array:

```
;
array99 rename george
```

```
;
george rename array99
```

setting the bounds rectangle:

```
;
array99 bounds 0 -2 10 2
```

```
;
array99 bounds 0 -1 5 1
```

adding x and y labels: give a point to put a tick, the interval between ticks, and the number of ticks overall per large tick.

```
;
array99 xticks 0 1 1
```

```
;
array99 yticks 0 0.1 5
```

adding labels. Give a y value and a bunch of x values or vice versa:

```
;
array99 xlabel -1.1 0 1 2 3 4 5
```

```
;
array99 ylabel 5.15 -1 0 1
```

You can also change x and y range and size in the "properties" dialog. Note that information about size and ranges is saved, but ticks, labels, and the actual data are lost between Pd sessions.

last updated for release 0.33

A Divide Between 'Compositional' and 'Performative' Aspects of Pd [*]

Miller Puckette

In the ecology of human culture, unsolved problems play a role that is even more essential than that of solutions. Although the solutions found give a field its legitimacy, it is the problems that give it life. With this in mind, I'll describe what I think of as the central problem I'm struggling with today, which has perhaps been the main motivating force in my work on Pd, among other things. If Pd's fundamental design reflects my attack on this problem, perhaps others working on or in Pd will benefit if I try to articulate the problem clearly.

In its most succinct form, the problem is that, while we have good paradigms for describing *processes* (such as in the Max or Pd programs as they stand today), and while much work has been done on *representations of musical data* (ranging from searchable databases of sound to Patchwork and OpenMusic, and including Pd's unfinished "data" editor), we lack a fluid mechanism for the two worlds to interoperate.

1 Examples

Three examples, programs that combine data storage and retrieval with realtime actions, will serve to demonstrate this division. Taking them as representative of the current state of the art, the rest of this paper will describe the attempts I'm now making to bridge the separation between the two realms.

1.1 CSound

In CSound[2], the database is called a *score* (this usage was widespread among software synthesis packages at the time Csound was under development). Scores in Csound consist mostly of 'notes', which are commands for a synthesizer. The 'score' is essentially a timed sequence. A possible score might be as shown:

```
i1 0 1 440
i1 1 1 660
i1 2 1 1100
e
```

[*] Reprinted from the 1st International Pd~convention, Graz, 2004: <http://puredata.info/community/projects/convention04/>

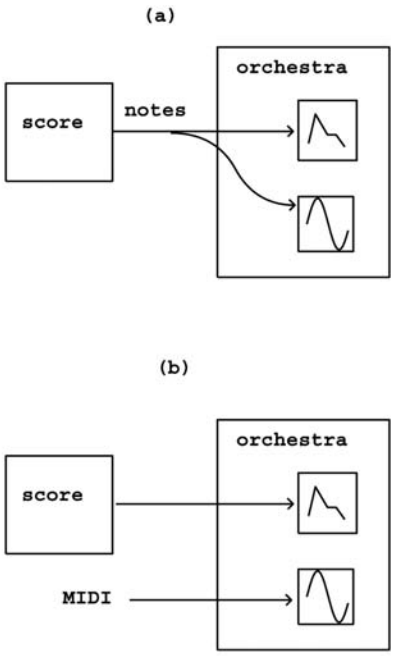

Figure 1: A Csound performance: (a) score and orchestra; (b) using real-time control via MIDI.

A Csound performance works as shown in Figure 1. Part (a) shows the "classical" performance configuration, in which parameters in the notes update synthesis control values, each note acting at an effective time also calculated from the note's parameters.

Part (b) of the figure shows how to use real-time inputs (here from MIDI messages) in a Csound performance. The real-time inputs are simply merged with the (pre-scheduled) notes. In effect, there is no facility for intercommunication between the two control streams; they simply affect different variables in the orchestra, and the orchestra's audio output is controlled by the union of the two sets of variables.

1.2 Patchwork and OpenMusic

Michael Laursen's Patchwork program[4] and its descendant, OpenMusic, by Carlos Agon and Gérard Assayag[1], offer a much tighter integration of data. Figure 2 shows a simple OpenMusic patch.

The semantic of OpenMusic (and Patchwork) is one of demand-driven dataflow. Each object is essentially a function call, which recursively evaluates its inputs, precisely as a Lisp form is evaluated. Compared to Pd, the relationship between data and process is reversed. There is no notion of real-time events or even of real time itself; rather, the contents of a patch are static

data. The paradigm gets its richness from the fact that the data types (which in the pictured example are just numbers) can in general be any lisp data structure, and so can easily describe whole sequences such as a Csound score.

OpenMusic supplies a sequencing function which, given a sequence as an argument, plays the result out the machine's MIDI port or sends it to a software synthesizer. The data managed in the patch itself are all entirely out-of-time; the sequencer's function of putting the data in time is a primitive operation. The lisp object or objects which hold rhythms, pitches, and even timbres are queried by the sequencer which does the data mining as a black box.

This is ideal from the composer's point of view, since the creation of a musical score is essentially an out-of-time activity. But performers will have little use for OpenMusic since, in live performance, the instrument doesn't query the

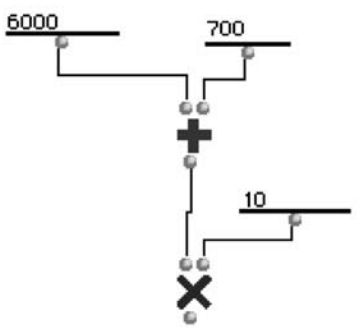

Figure 2: An OpenMusic patch (borrowed from IRCAM's documentation).

performer, but rather, the performer sends messages to the instrument. This is the Pd (and Max) organization, the reverse of that of OpenMusic.

1.3 Max and Pd

In Pd (the third example), the fundamental transaction goes in the direction favored by the performer. This idea goes back to Max, and that orientation might have been the most important single reason that Max and Pd are in wide use today. But as noted before[3, 5], the message-sending paradigm does not fundamentally lend itself well to storing and retrieving data. One is almost forced to set data aside in containers – databases, essentially – and to use a coterie of accessor objects to store and retrieve data under real-time, message-passing control.

Max's approach to data is both simple and evasive: special data-container objects such as table, qlist, etc. are provided; the data are essentially hoarded inside the container objects, and for

each kind of container object, a particular *ad-hoc* approach is taken to its storage, its editing, and its interfacing with the rest of the patch.

Retrieval (the great majority of database transactions!) is the worst fit with Max because messages don't have return values; the retrieved data must be sent as a separate return message. This leads to much misery for Max users.

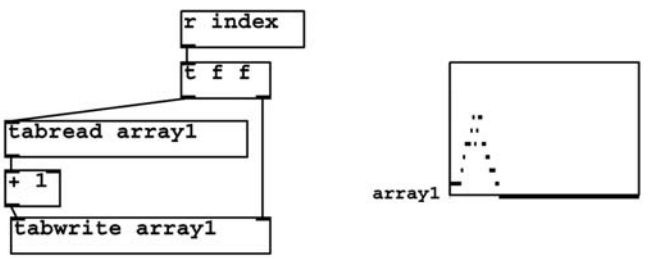

Figure 3: Incrementing an array element. The receive object can be sent integers to specify which element to increment.

Pd faithfully recreates the data-storage paradigms of Max, but in addition the design of Pd includes a more advanced paradigm that might eventually replace the Max one.

The original, defining idea behind Pd was to remove the barrier between eventdriven real-time computation (as in Max-style message passing) and data (as in points of an array or notes in a score). In Pd the two (object boxes and data structures) can easily coexist in a single window. This promiscuity, however, does not in itself make the functional objects and the data intimately connected. In fact, in the present design, data access still has to be done through a suite of accessor objects. It is far from certain that Pd will, in the end, relieve the Max user's misery.

2 Data-Plus-Accessor-Object Design Model

An example of Pd's data-plus-accessor arrangement is that maintained by floatingpoint arrays (either graphical or via the table object), and the suite of objects tabread, tabwrite, and all their relatives. For example, Figure 3 shows how to use accessor objects to increment a variable element of an array (you would do this to make a histogram of incoming indices, for example.) Here the task is straightforward, and the separation of the storage functionality of the actual "array1" from the accessor objects is not particularly troublesome.

Moving to a more interesting case, we now build a patch to do something corresponding to this using the (still experimental) "data" feature of Pd[6]. For completeness we give a short summary here, which will serve also to introduce the central example of this paper.

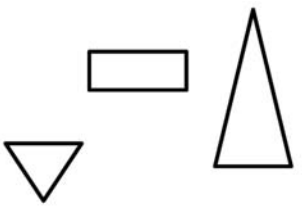

Figure 4: Two data structures in Pd.

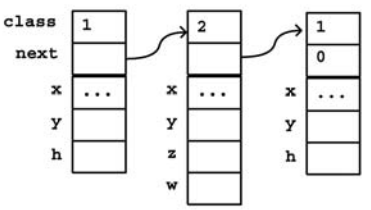

Figure 5: Possible structure for the objects in Figure 4.

Pd's "data" are objects that have a screen appearance; many such objects can be held in one Pd canvas. The canvas holds a linked list of data. A datum belongs to some data structure, which is defined by a patch called a *template*. The template also defines how the data will look on the page. Lists of data are heterogeneous; a canvas in Pd can hold data with many different structures. Figure 4 shows a canvas with three data objects. They belong to two types: two triangles and one rectangle.

As data structures, this list could appear as shown in Figure 5. The elements of the list need not all have the same structure, but they have a "class" field that determines which of the several possible data structures the element actually belongs to.

```
struct struct1 float x float y float h

filledpolygon 999 0 2 0 0 20 h 40 0
```

Figure 6: Pd definitions of the data structure for the triangles in Figure 4.

The data structures are defined by struct objects. Figure 6 shows the struct object corresponding to the triangles in Figure 4. The canvas containing the struct object may also contain *drawing instructions* such as the drawpolygon object. This object takes three creation arguments to set the interior and border colors and the border width (999, 0, and 2), and then any number of (x, y) pairs to give vertices of the polygon to draw; in this example there are three points and the structure element h gives the altitude of the triangle.

For clarity, and for the sake of comparison, we'll consider C and Pd approaches to defining and accessing the data in parallel. In C, the definitions of the two data structures might be as shown in Figure 7. In order to be able to mix the two structures in a single linked list, a common structure sits at the head of each. This common structure holds a whichclass field to indicate which structure we're actually looking at, and a next field for holding a collection of these structures in a linked list.

Now we define a task that might correspond to that of Figure 3. Suppose, given an integer *n* and a linked list of data structures, we wanted to find the *n*th occurrence of struct1 in the list and increment its h slot. (Note that incrementing the h slot of an instance of class2 wouldn't make sense.) A C function to do this is shown in Figure 8. This is an inherently more complicated problem than that of Figure 3; there, a corresponding piece of C code might simply be:

```
array[n] += 1;
```

Instead, we have to make a loop to search through the heterogeneous list, checking each one if it belongs to struct1, maintaining a count, and when all conditions line up, incrementing the h field.

An equivalent Pd patch is shown in Figure 9. The loop is managed by the [until] object. The top [trigger] initializes the [f] and [pointer] objects (corresponding to `count` and `ptr` in the C code.) The messages, "traverse pd-list" and "next", correspond to the initialization and update steps in the C loop; the two possible exit conditions of the loop (the check on `ptr` and the break in the middle) are the two patchcords reaching back to the right inlet of the [until] object.

The [pointer struct1] object only outputs the pointer if it matches "struct1"

```
struct anyclass                 /* common header in both structs below */
{
#define CLASS1 1
#define CLASS2 2
    int c_whichclass;           /* whether CLASS1 or CLASS2 */
    struct anyclass *c_next;    /* next in linked list */
};

struct class1                   /* first one */
{
    struct anyclass c1_header;  /* common part */
    float c1_x;                 /* part that is specific to class #1 */
    float c1_y;
    float c1_h;
};

struct class2                   /* second one */
{
    struct anyclass c2_header;
    float c2_x;
    float c2_y;
    float c2_z;
    float c2_w;
};
```

Figure 7: A C equivalent for the data structures of Figure 4.

```
extern struct anyclass *thelist; /* externally defined head of linked list */

void incrementclass1(int n)      /* increments the nth "class1" in list */
{
    struct anyclass *ptr;
    int count;
        /* search the list, stopping when pass "n" or run out of items */
    for (count = 0, ptr = thelist; ptr; ptr = ptr->c_next)
    {
        if (ptr->c_whichclass == CLASS1)       /* check class of this item  */
        {
            if (count == n)                        /* if it's the nth one: */
            {
                ((struct class1 *)ptr)->c1_h += 1;  /* increment "h" field */
                break;                              /* and we're done */
            }
            count++;                               /* increment count */
        }
    }
}
```

Figure 8: A C function to increment "h" for the nth occurrence of class1.

(corresponding to the first if in the C code.) The [sel] object in the patch corresponds to the check whether `count and n` are equal in the C code. The remainder, below the second [pointer] object, is much the same as in Figure 3.

3 Discussion

The Pd patch looks more complex than the C code. One possible reason for the complexity is the difficulty of sequencing actions in Pd patches, which lack the natural sequentiality of a text programming language like C. Another is the relative lack of names; only three names (other than Pd class and message names) appear in the patch ("n", "struct1", and "h"), compared to eight in the C code (`thelist`, `n`, `ptr`, `count`, `c_next`, `c_whichclass`, `CLASS1`, and `c1_h`). Of these, two of the Pd names ("n", and "pd-list") might be considered to act as variables, compared to four of the C names.

The complexity of the patch needed to accomplish this task might be reduced somewhat if either the [value] and/or [expr] objects were extended to deal with pointers. For instance, a [value ptr] object (unsupported in the current version of Pd) could hold the output of the first pointer object in readiness for the incrementing step at bottom. So far, though, mockups using features such as this have not been observed to reduce the number of objects and lines in patches equivalent to the one shown here.

The [expr] object could conceivably handle data structures and slots with the addition of a few C-like constructs, and could also be fixed to set and retrieve the contents of value-style variables. This would cause the Pd and Max versions of [expr] to deviate from one another (they currently share the same code, maintained by Shahrokh Yadegari[7]. In general, it seems problematic to lean in too fundamental a way on [expr] as the fundamental mechanism for getting and retrieving data.

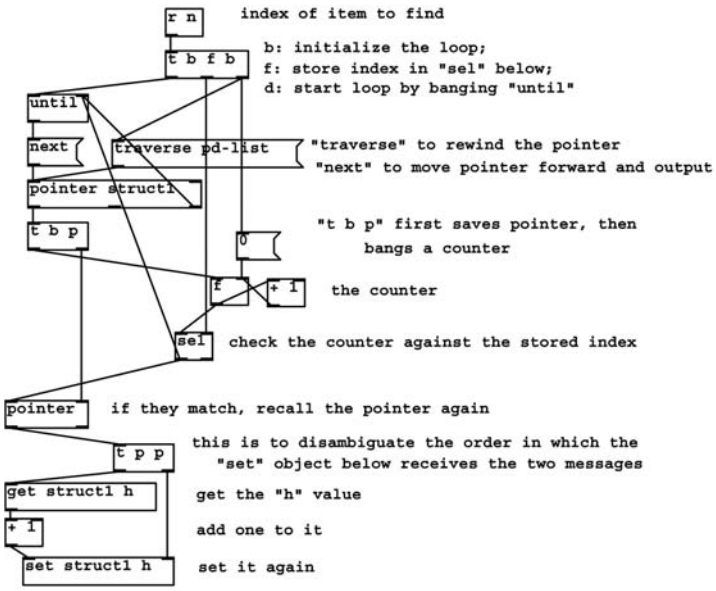

Figure 9: Pd patch to search for the *n*th instance of class1 and increment its value of h.

On a more general plane, the relationship between the data and the code that accesses it is the same in Pd as it is in C. One wishes that the functionality could somehow reference the look and feel of the data themselves, or possibly even be built into the template patch (as methods go with class definitions in C++.) So far no model has emerged that accomplishes this smoothly.

Another aspect of the question, not touched on in this paper, is the utility of somehow catching user operation (with mouse and keyboard, perhaps among other ways) with the graphical data. There should be a way to provide hooks to data when certain operations are carried out on them. Perhaps this should be realized as a way of fielding Pd messages (via [pointer] objects?) sent to objects to get or set their state.

The data structure accessor objects could easily be back-compatibly replaced or augmented with others if a clean design can be found for getting and setting the data. This "data" feature was the original motivating force behind Pd's design; it is interesting that it now appears likely to be the last aspect of Pd to be defined.

References

1 Gerard Assayag et al. Computer assisted composition at ircam: From patchwork to openmusic. *Computer Music Journal*, 23(3):59-72, 1999.

2 Richard Boulanger, editor. *The Csound book*. MIT Press, Cambridge, Massachusetts, 2000.

3 P. Desain and H Honig. Letter to the editor: the mins of max. *Computer Music Journal*, 17(2):3–11, 1993.

4 Mikael Laurson and Jacques Duthen. Patchwork, a graphical language in preform. In *Proceedings of the International Computer Music Conference*, pages 172-175, Ann Arbor, 1989. International Computer Music Association.

5 Miller S. Puckette. Max at 17. 26(4):31-43, 2002.

6 Miller S. Puckette. Using pd as a score language. pages 184-187, 2002.

7 Shahrokh Yadegari. A general filter design language with real-time parameter control in pd, max/msp, and jmax. In *Proceedings of the International Computer Music Conference*, pages 345-348, Ann Arbor, 2003. International Computer Music Association.