128 Is Not Enough - Data Structures in Pure Data

Frank Barknecht
GOTO10
Neusser Wall 2
D-50670 Köln,
Germany,
fbar@footils.org

Abstract

A lesser known feature of Miller Puckette's popular audio and media development framework "Pure Data" is the possibility to create user defined graphical data structures. Although the data structures are included in Pd for several years, only recently a sigificant number of users discovered and used this feature. This paper will give an introduction to the possibilities of Pd's data structures for composers and musicians and present several example applications.

Keywords

Pure Data, graphical score, multimedia, programming language, composition

1 Introduction

Since its introduction in 1996[1] Miller Puckette's[2] software environment Pure Data[3], or short Pd, has grown to become one the most popular open source applications amongst media artists. Today Pd not only supports the production of audio or midi data - with extensions like Gem or PDP it is also widely used in the field of video art and multimedia. While the user base of Pd literary is huge compared to most other related free software, one central feature of Pd is not in such a wide use, although it was one of the motivations to write Pd in the first place, according to (Puckette, 1996):

> Pd's working prototype attempts to simplify the data structures in Max to make these more readily combined into novel user-defined data structures.

"Novel user-defined data structures" is the key term here. Simple numbers, let alone just 128 of them as in the MIDI standard, do not provide a sufficient vocabulary for artists in any field. Moreover predefining a limited vocabulary at all is not flexible enough for unforseen and novel uses.

In (Puckette, 2002) Puckette further states about his motivation:

> 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.

So the data structures Pd offers carry another property: They are *graphical* structures, that is, they have a visual representation as well.

Defining a structure for data alone is of not much use unless there are ways to access and change the stored data. For this task Pd includes several accessor objects, which will be explained below. It also is possible to edit the stored data through the graphical representation of a structure using mouse operations.

2 Defining data structures in Pd

The central object to create a structure definition in Pd is called `struct`. While a `struct` object theoretically can be created anywhere in a Pd patch, it generally is put into a Pd subpatch to be able to associate it with instructions for its graphical representation.

Every `struct` needs to be given a name and then one or more fields to carry its data. For example a structure defining a note event might look like this: `struct note float freq float vel float len`.

Besides fields for floating point numbers, a structure can also have fields of type `symbol` which stores a word, and `array`. The latter can be used to carry collections of structures defined elsewhere in Pd. Members of an `array` field have to be of the same type. An example for this could be a score structure, which holds several of our `note` structs in a "melody" and also gets a score-name field: `struct score`

[1] (Puckette, 1996)
[2] `www-crca.ucsd.edu/~msp/`
[3] `www.puredata.org`

`symbol score-name array melody note`. A fourth type that can be used in a `struct` definition is the `list` type, which similar to `array` can hold a collection of other structures, however these elements can be of different `struct` types.

Two field names are treated in a special way: `float x` and `float y` are used to specify the x- and y- coordinates of a structure's graphical representation. Such representations are specified by adding objects for drawing instructions to the same subpatch, that carries the structure definition. A very simple instruction is `drawnumber`, which just displays the value of the field given as its first argument: `drawnumber freq` will draw the current value of the freq-field of a structure. It also is possible to change that value using the mouse by click and drag.

3 Pointer Magic

The structure definition in Pd can be compared to `struct` in programming languages like C. This analogy is taken even further if we look at the way, instances of data structures are created and their data is accessed. Because this is done through pointers much like in C.

Pointers in Pd are a special data type. Other types would be float and symbol — also called "atoms", because they reference a single, atomic value — and lists made of several atoms. Pointers can be made to reference instances of `struct` structures. Pd has an object to hold these pointers which is called `pointer` as would be expected. To make a `pointer` object actually *point* to something in a Pd patch, it has to be told, where to find that something. Subpatches play an important role here.

3.1 Subpatches as named regions

Subpatches in Pd are commonly used to group related functionality and to make better use of the limited screen estate. However they also have a syntactic meaning, because they create a *named region* inside of a Pd patch. This region can be a target for various operations. Every subpatch in Pd can be accessed by sending messages to a receiver that is automatically created by prepending the subpatch's name with the string "pd-". An example for an operation supported by subpatches is `clear`, which deletes every object inside the target subpatch. A subpatch called "editor" thus can be cleared by sending the message `clear` to a sender called `pd-editor` as shown in Figure 1:

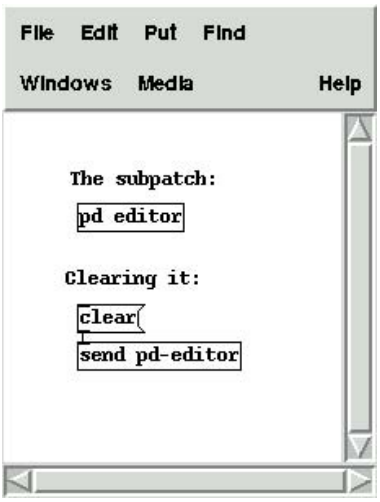

Figure 1: clearing a subpatch with a message

Now if a subpatch contains instances of data structures, these are organized as a linked list, which can be traversed using `pointer` objects. For this, `pointer` supports traversal operations initiated by (amongst others) the following messages:

- `bang`: output pointer to current element
- `next`: output pointer to next element
- `traverse pd-SUBPATCH`: position pointer at the start of the list inside the subpatch called "SUBPATCH".

3.2 Creating instances of data structures

Given a pointer to any position in a subpatch canvas, it is possible to insert new instances of structures using the `append` object. For this, `append` needs to know the type of structure to create and at least one of the fields to set.

Using our note example from above, one could use `append` like this: `append note freq`. For every field specified this way, the `append` object will generate an inlet to set the creation value of this field in the new instance. Additionally it will have a further, rightmost inlet, which has to be primed with a pointer, that specifies the position, after which the new structure instance should be inserted.

Supposing we have a subpatch called `editor` in our patch, we can get a pointer to the start of this subpatch by sending the message `traverse pd-editor` followed by `bang` to a `pointer` object, that itself is connected to `append`s rightmost inlet. Sending a number like 60 to the

leftmost inlet (the `freq` inlet) will then create a graphical instance of `struct note` inside the subpatch `editor`, whose frequency field is set to 60 and whose other fields are initialized to zero. For now, as we only used a `drawnumber freq` as single drawing instruction, the graphical representation is quite sparse and consists just of a number in the top-left corner of the subpatch.

3.3 Get and set

The current values of fields in this new instance of a `struct note` can be read using the `get`-object, which on creation needs to know, which `struct`-type to expect and which fields to read out. If a valid pointer is sent to a `get note freq vel` object, it will output the current value of the frequency and the velocity field stored at that pointer's position.

The opposite object is called `set` and allows us to set the fields inside an instance, that is specified by sending a pointer to the rightmost inlet of `set` first.

By traversing a whole subpatch using a combination of `traverse` and `next` messages sent to a `pointer` that is connected to `get`, reading out a whole "score" is easily done.

Accessing `array` fields is slightly more complicated, because it requires an intermediate step: First we need to `get` the array-field out of the initial pointer of the structure. The array field is itself represented by a pointer. This pointer however can be sent to the right inlet of an `element` object, that on its left inlet accepts an integer number to select the element inside of the array by this index number.

4 Drawings

Unless the subpatch containing the `struct` definition also has drawing instructions, the structure instances will be invisible, when they are created. Pd offers several graphical primitives to display data structures. `drawnumber` was already mentioned as a way, to draw a numeric field as a number. If the `struct` has float-typed fields called `x` and `y` this number also can be moved around in the subpatch in both dimensions. The x- and y-fields are updated according to the current position with the upper left hand corner of the subpatch window being at a point (0,0). The x-axis extends from left to right, the y-axis extends from *top to bottom* to make (0,0) stay at the upper left.

The various drawing primitives accept coordinates to specify their positions and dimensions as well, however these are calculated in a local coordinate system, whose (0,0)-point is translated to the point specified by the value of the fields `float x float y` in the `struct` definition. For example, using this definition `struct coord float x float y` we can use two `drawnumber` objects to draw `x` and `y` without overlapping by creating `drawnumber x 0 0` and `drawnumber x 0 15`. The y-field will be drawn 15 pixels below the x-field, as shown in figure 2 (which also shows colors and labels in `drawnumber`).

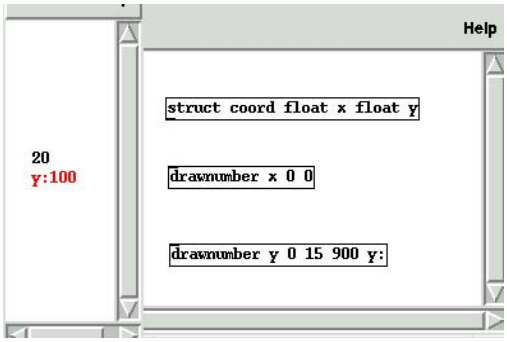

Figure 2: relative positioning

4.1 More drawing instructions

Further objects for drawning data are `drawpolygon`, `filledpolygon`, `drawcurve`, `filledcurve` and `plot`. They are described in their respective help patches. Here we will only take a look at `drawpolygon`, that is used to draw connected line segments. Like all drawing instructions it accepts a list of positional arguments to control its appearance and behavior. In the case of `drawpolygon` these are:

- optional flag `-n` to make it invisible initially
- alternatively a variable given by `-v VAR` to remotely control visibility
- Color specified as RGB-values.
- line-width in pixels
- two or more pairs of coordinates for the start and end points of the line segments.

The next instruction would draw a blue square of width `w`: `drawpolygon 9 1 0 0 w 0 w w 0 w 0 0` (Fig. 3)

The size of the square, that is, the variable `w` in this structure, can be changed using the mouse or with `set` operations. The current value of `w` always is accessible through the `get` object, see section 3.3.

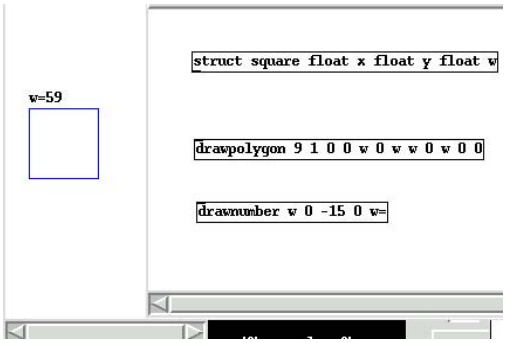

Figure 3: a blue square

4.2 Persistence

Saving a patch containing instances of data structures will also save the created data structures. Additionally it is possible to export the current contents of a subpatch to a textfile by sending the message `write filename.txt` to the subpatch's receiver (described in 3.1) and read it back in using a similar `read`-message.

Such a persistence file contains a textual description of the data structure templates and their current values, e.g. for our `square`-structure:

```
data;
template square;
float x;
float y;
float w;
;
;
square 136 106 115;
```

5 Data structures in action

Instead of explaining all the features in detail, that data structures in Pd provide, I would like to present some examples of how they have been used.

5.1 Graphical score

Pd's data structures most naturally fit the needs of preparing and playing graphical scores. In his composion "Solitude", north american composer and Pd developer Hans-Christoph Steiner used data structures to edit, display and sequence the score. The graphical representation also controls the sequencing of events. He describes his approach in a post to the Pd mailing list:[4]

> The experience was a good combination of visual editing with the mouse and text editing with the keyboard. The visual representation worked well for composition in this style. [My] biggest problem was finding a way to represent in the score all of the things that I wanted to control. Since I wanted to have the score generate the piece, I did not add a couple features, like pitch shifting and voice allocation control, which I would have liked to have.

Both the Pd patch (Fig. 4) and a recording of "Solitude" are available at the composer's website.[5]

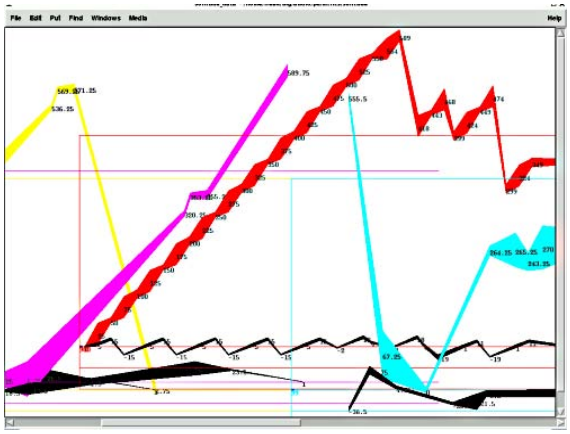

Figure 4: Score for "solitude" by Hans-Christoph Steiner

5.2 Interaction

"Solitude" changes the data stored inside a structure only during the compositional phase, but not in the performance of the piece. An example, where data structures are manipulated "on the fly" is the recreation of the classic video game "PONG" by the author[6], as shown in Fig. 5.

This simple piece uses the ratio of the current score in the game (1/2 in Fig. 5) to influence a fractal melody played in the background. The x-position of the ball is read out by a `get`-object to pan the stereo position of the melody.

5.3 GUI-building

Data structures also are useful to implement custom GUI elements for Pd. A collection of

[4] `lists.puredata.info/pipermail/pd-list/2004-12/024808.html`

[5] `at.or.at/hans/solitude/`

[6] `footils.org/cms/show/27`

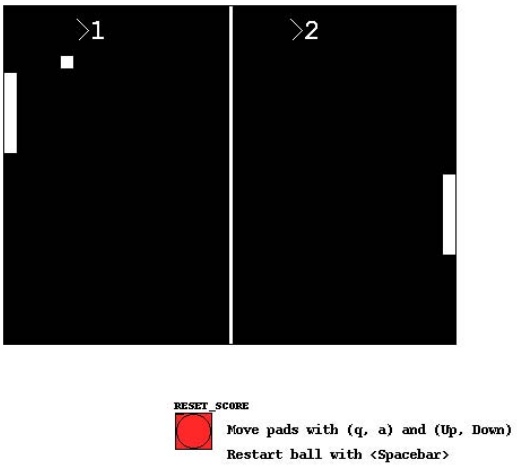

Figure 5: "PONG" by Frank Barknecht

these is currently built by Chris McCormick.[7] Figure 6 shows an envelope generator and a pattern seqencer of variable length, that can be reused several times in a patch.

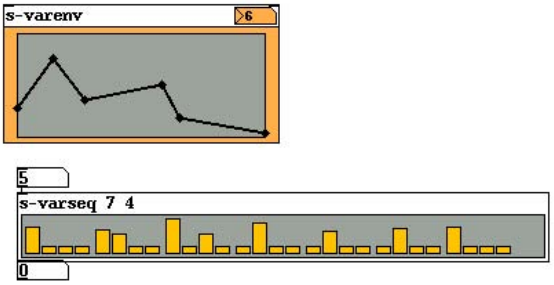

Figure 6: Two GUI objects by Chris McCormick

5.4 Visualisation

Many advanced concepts in computer music or digital art in general deal with rather abstract, often mathematical issues. Data structures can help with understanding these concepts by connecting the abstract with the visible world.

The german composer Orm Finnendahl created such interactive Pd patches using data structures to explain things like the sampling theorem or granular synthesis.

With his patches "pmpd-editor" and "msd-editor" (Fig. 7) the author of this paper wrote tools to explore particle systems (masses connected by springs) interactively. A user can create the topology for a particle system and

[7] mccormick.cx/viewcvs/s-abstractions/

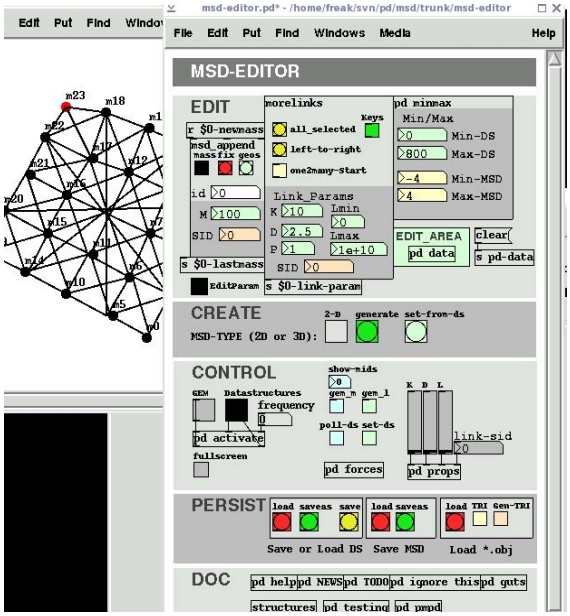

Figure 7: Editor for mass-spring-damper-topologies by Frank Barknecht

animate it directly inside the Pd patch. Various helper functions provide means for importing and exporting such topologies to be used in other applications as 3D-modellers for example. The particle systems designed with the editor can also generate control data to influence various synthesis methods. The editor is available in the CVS repository of the Pure Data developer community at `pure-data.sf.net`.

5.5 Illustration

Finally we get back to another original motivation for writing Pd in the first place. In (Puckette, 1996) Puckette writes:

> Pd's first application has been to prepare the figures for an upcoming signal processing paper by Puckette and Brown.

Almost a decade later, Puckette is still using Pd to illustrate paper, this time for his book project "Theory and Techniques of Electronic Music".[8]

All graphics in this book were made using Pd itself, like the one shown in Fig. 8.

6 Conclusion

This paper could only give a short introduction to Pd's data structures. As always they

[8] So far only available online at: `crca.ucsd.edu/~msp/techniques.htm`

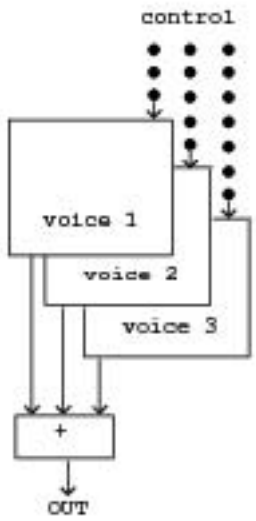

Figure 8: Illustration courtesy by M. Puckette from his book: "Theory and Techniques of Electronic Music"

are best learned by studying examples. Pd comes with documentation patches that can be edited and changed. Most patches that use data structures are collected in the directory "doc/4.data.structures" of the Pd documentation. The HTML-manual of Pd contains further information in chapter "2.9. Data structures".[9]

Pd's data structures are powerful tools that can greatly enhance the possibilities of Pd. In some areas they still are a bit awkward to use though. For example animating large numbers of data structures may influence audio generation and even lead to dropped samples and clicks. There also still are issues with providing a smooth framerate. In the author's view data structures thus cannot replace specialized extensions like Gem in this regard yet. If they should try to do so at all, remains an open question.

However problems like this can only be found and fixed, if more artists and musicians in the Pd community will actually use them—a classical chicken and egg problem. Thus it is hoped that this paper will create more interest in Pure Data's data structures.

[9] `www-crca.ucsd.edu/~msp/Pd_documentation/`

7 Acknowledgements

References

M. Puckette. 1996. Pure data: another integrated computer music environment. In *Proc. the Second Intercollege Computer Music Concerts*, pages 37–41.

M. Puckette. 2002. Using pd as a score language. In *ICMC Proceedings*, pages 184–187.