PD's Data Structures Tutorial

by Gregorio Garcia Karman

I. Introduction

"The original idea in developing Pd 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 graphic representations" (Puckette, Pd Documentation).

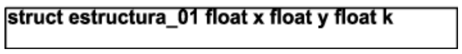
"Pd is designed to to offer an extremely unstructured environment for describing data structures and their graphic appearance. The underlying idea is to allow the user to display any kind of data he or she wants to, associating it in any way with the display. To accomplish this Pd introduces a graphic 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" (Puckette, Pd Documentation).

This tutorial is a product of the author's own experiences with data structures and the related objects. This is not official documentation for these objects and the author cannot be made responsible for any unwanted results.

II. Basic operations

struct

The **struct** object defines the data structure. Its arguments are: the name of the data structure, and the name and type of the field(s) contained within. In the following example we have created a data structure named *estructura_01* with three *float* fields (x, y, k).



```
struct estructura_01 float x float y float k
```

Fig. 1 - A **struct** object generating three *float* fields. **[tut_01.pd]**.

The **struct** object is to be placed inside a subpatch. A **struct** object inside a subpatch makes a *template* [fig 2.a]. Each *template* should contain only one **struct** object and the name of that object should be unique within all open pd patches [1].

For data storage it is also necessary to create a *datawindow*. This is an empty subpatch, also with a unique name - it's helpful to think of this second subpatch as a

[1] If this isn't followed the pd terminal will deliver an error message and things will tend to not work properly.

'holding space' in which our data will be kept. This *datawindow* is also the place where the data structure's graphic representation will be seen and manipulated, as we shall see later on [2].

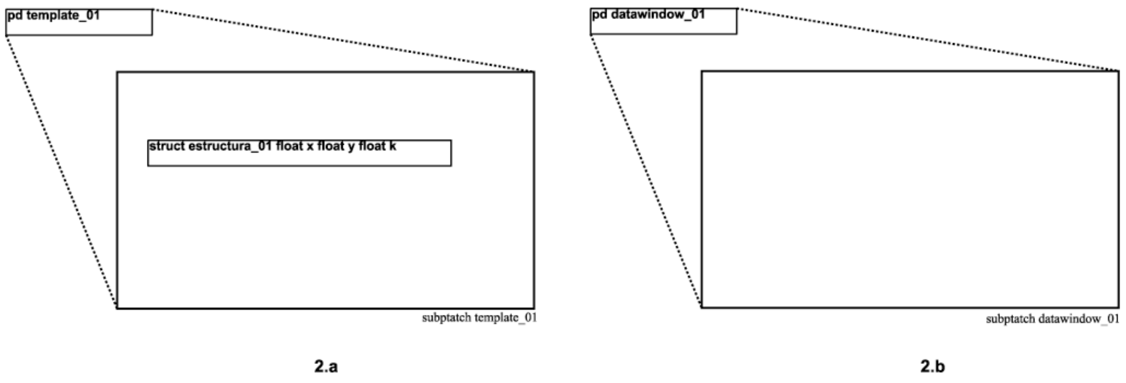


Fig. 2. The first steps for creating a data structure are a) create a struct object inside a subpatch b) Create another empty subpatch for the storage of the data and the graphic representation and modification of that data.

Although our list is now empty, the data elements 'held' in a *datawindow* will be organized as a list of ordered scalars. As you can see in the following diagram [Fig. 3], each one of these scalars is comprised of multiple data fields organized according to the definitions inside the struct object. Here the first scalar of the list will hold the values x_1 ,y_1 ,k_1 the second scalar x_2 ,y_2 k_2 etc....

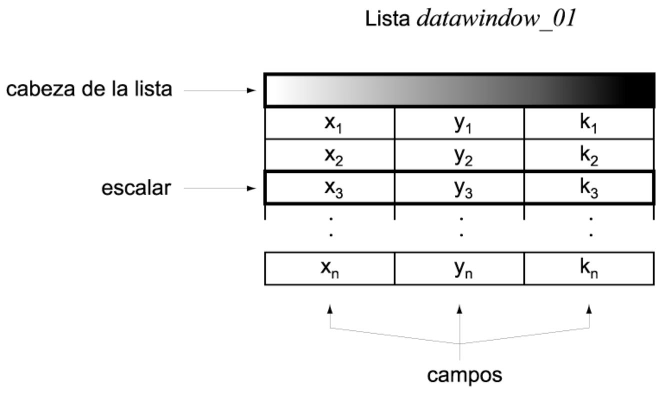


Fig. 3 Data stored in the subpatch *datawindow_01* is comprised of a list of ordered *scalars*.

Access to specific points on the list is effectuated by pointers, an atom type that enables the localization of a specific *scalar*. Navigating pointers is one of the keys to making data structures work in Pd, so it's necessary to understand how the **pointer** object works before we can add scalars to our list.

[2] Later on, we'll see how to incorporate drawing instructions - the other (optional) element of the template.

pointer

Pointer, like **float**, is a storage object, though instead of storing a number, it stores the position of a *scalar* on the list. This stored position can correspond to either an existing *scalar* or to the list head, a special position that is located before the first scalar and that cannot hold any data [3] [Fig. 3]. Pointing to the list head is very useful now in order to be able to refer to our empty list. To get a list-head-oriented pointer we send a *traverse [pd-datawindow]* message to the left inlet of the **pointer** object.

When this is done, the **pointer** object is initialized and is pointing towards the *datawindow* list head. Now send **pointer** a *bang* to its left inlet and it will output a *pointer* (an atom like *float* or *symbol* [4]) out its left outlet that reports its position (the *datawindow*[5] list-head in this case).

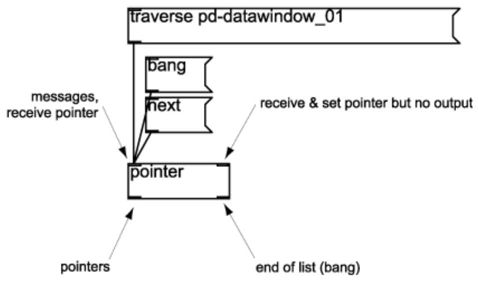

traverse pd-datawindow_01
bang
next
messages, receive pointer
receive & set pointer but no output
pointer
pointers
end of list (bang)

Fig. 4 Operation of the pointer object

If we send **pointer** a next message to its left inlet, the object will output pointers out its left outlet that correspond to each successive scalar on the list. Because our list is currently empty, though, a *next* message will output a *bang* out the right outlet indicating that we have reached the end of the list and the left outlet will output nothing. Yet another next message will generate an 'non-existent pointer' error in the Pd terminal: `error: ptrobj_next: no current pointer`. Watch the two pointer outputs while you try these different messages.

Once we have defined storage for our data, and the pointers that allow us to access that data, we can begin adding scalars to our list using the **append** object.

append

To add scalars to a list we use the **append** object. The object inserts new scalars into the specific position on the list that is determined by a *pointer*. The creation

[3] The use for this is twofold: to be able to point to an empty list, and to be able to be able to add a new first scalar to an already populated list using the **append** object.
[4] It can be used with some routing operations like *trigger* or *pack*, but can't be visualized with a <number> or <symbol> object or displayed through pd's terminal.
[5] When a pointer points to a list head, it's known as an 'empty pointer.'

arguments for **append** are: name of the relevant data structure, and the names of the fields we want to update (there should be at least one).

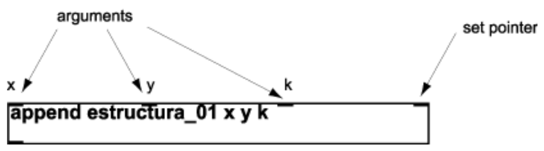


Fig. 5 The **append** object and its inlets

Once the object and its arguments are initialized, the following steps are needed to use the **append** object [figure 5]:

- Establish a pointer: First we specify the list to which we want to insert a *scalar* by sending a *pointer* to **append**'s right inlet. In the previous example, we saw that by clicking on the *traverse datawindow_01* message, the **pointer** object generated a *pointer* positioned at the list head of *datawindow_01.* Upon receiving such a *pointer*, the **append** object is ready to add a scalar to the corresponding position (1).
- Add values. These are introduced by way of the remaining inlets. There are as many inlets as there are scalar fields as defined in **append**'s arguments (in this case *x, y, k*) (2). When the far left (hot) inlet receives a value, it, along with those stored in the remaining inlets, are added to the list, creating a new *scalar.*

Append internally maintains a pointer-location that is established, as we have seen, by sending a *pointer* to its right inlet. When a new scalar is added to the list, this internal pointer-location changes so that it is always pointing to the scalar just added (this is also echoed out the left output) (3). Once **append**'s internal pointer has been established, there is no need to resend the *traverse* message when adding more scalars, unless you want to return to the list head.

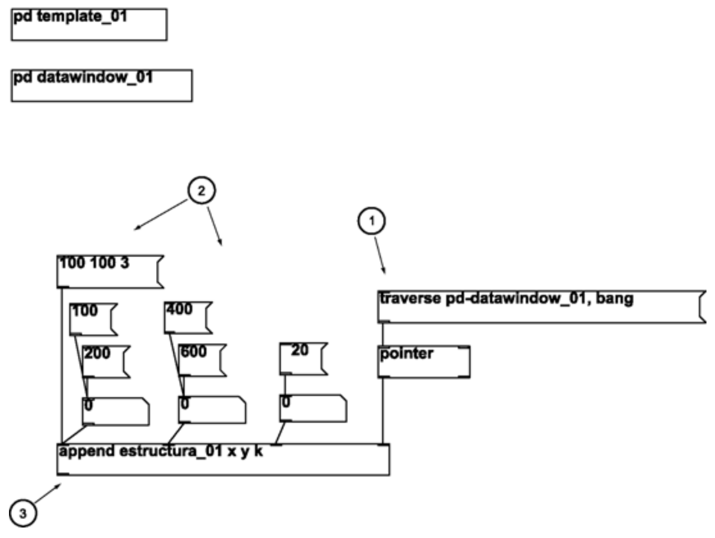
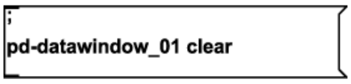


Fig. 6 View of the patch. Sequence of actions needed to add the first scalar to the list using **append**. [**tut_06.pd**]

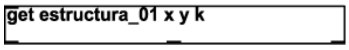
get

We can use the **get** object to view the scalars we have added. This object recalls and displays the contents of any number of fields of a scalar. The creation arguments for **get** are: the name of the structure where the scalar we are to see resides (in this case *estructura_01);* and the names of each constituent field we wish to see. As such, **get** is created with one inlet and as many outlets as fields are invoked in the object's arguments [6].

If you want to RE-INITIALIZE YOUR LIST by erasing all its data (to add a new set of scalars for example) you can send a clear message to the data window:



```
;
pd-datawindow_01 clear
```



```
get estructura_01 x y k
```

Upon receiving a pointer that references an existing scalar, the **get** object outputs the values for each field of that scalar. As was the case with **append,** we have to initialize the object's internal pointer by sending a *traverse* message to the **pointer** object. We can then move through each scalar with subsequent *next* messages.

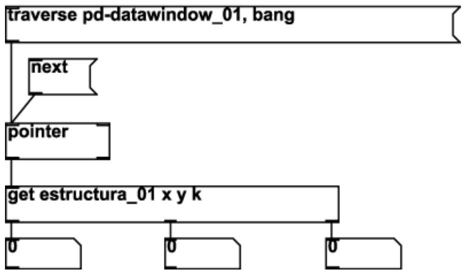
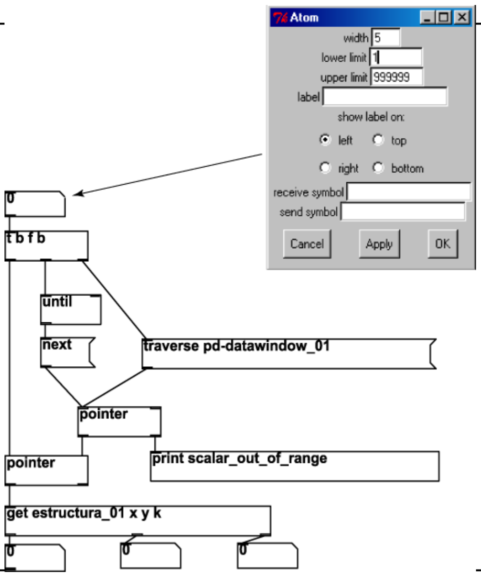


Fig. 08. Basic connections for use of the **get** object. **[tut_08.pd]**

EXAMPLE tut 09.pd
A small patch for accessing any position on the list using the **until** command. This method is very useful if we already know the numerical position of the scalar we wish to see. For patches that include **until,** it's important to safe-guard against infinite loops; here we can set the upper limit of the number box to anything greater that zero *(eg: lower limit = 0 , upper limit = 99999).*



width 5
lower limit 1
upper limit 999999
label
show label on: [✓ left] top right bottom
receive symbol
send symbol

6 The fields should exist in the named data structure.

set

Like **get**, the creation arguments for **set** are the name of the structure that defines the scalar, and the names of the fields that are to be modified. The far right inlet expects to receive a *pointer* indicating the scalar's position whose value we want to set (*traverse, next,...).* The remaining terminals are used to add new values to the corresponding fields. This works the same way as **append**, the far left inlet is 'hot' and initiates the update to the current scalar. Unlike **append,** the **set** object does not add new scalars, but only modifies field values for existing ones.

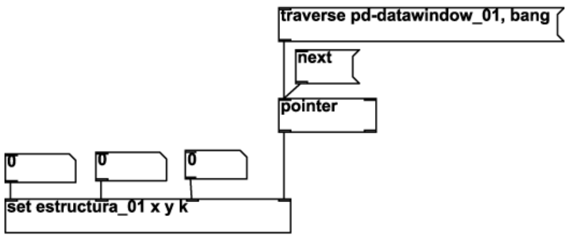


Fig. 10 Object assembly for use of the **set** object.

We have now covered all the basic mechanisms for creating, storing, modifying, and viewing a basic data structure. In the next section, we will see how to link the data in our list with graphic objects.

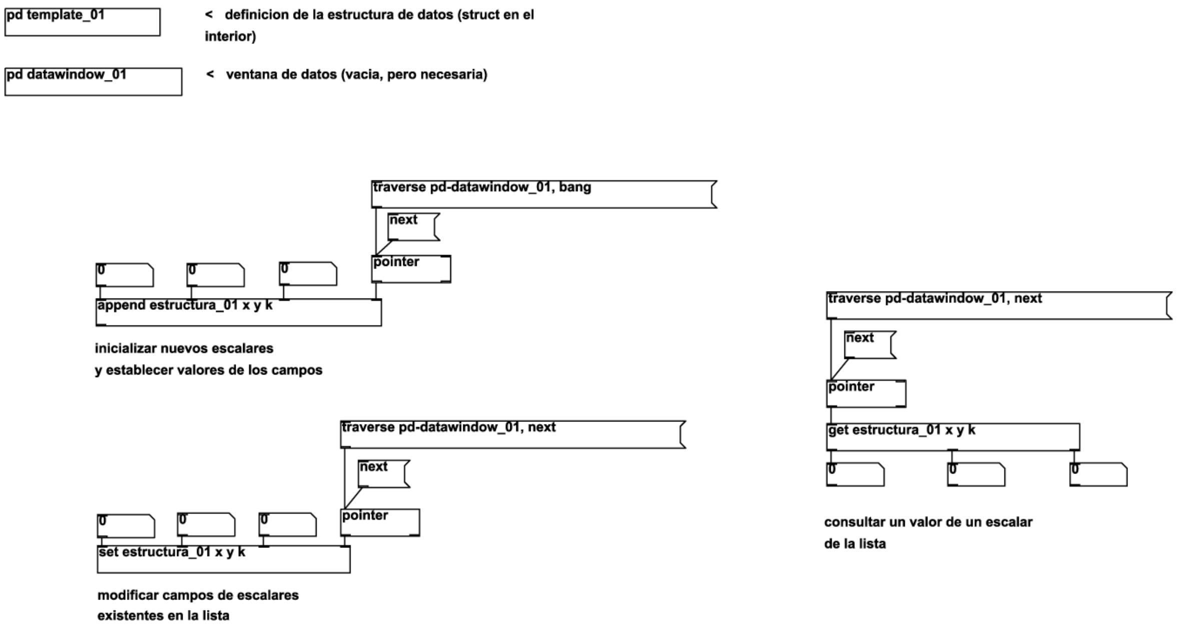
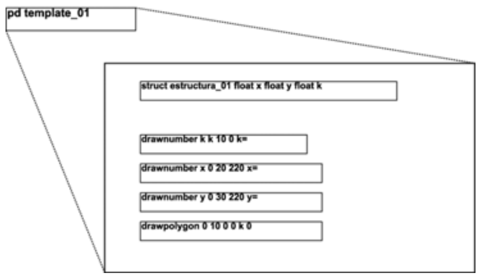

pd template_01
< definicion de la estructura de datos (struct en el interior)
pd datawindow_01
< ventana de datos (vacia, pero necesaria)
traverse pd-datawindow_01, bang
next
pointer
0
0
0
append estructura_01 x y k
inicializar nuevos escalares
y establecer valores de los campos
traverse pd-datawindow_01, next
next
pointer
0
0
0
set estructura_01 x y k
modificar campos de escalares
existentes en la lista
traverse pd-datawindow_01, next
next
pointer
get estructura_01 x y k
0
0
0
consultar un valor de un escalar
de la lista

Fig. 11. View of the complete patch with all the basic objects for manipulating a basic data structure. **[tut_11.pd]**

Drawing Instructions

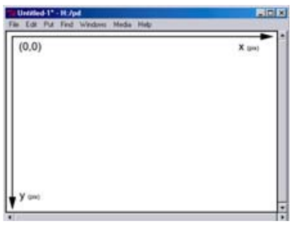
The drawing instructions, **drawnumber, drawpolygon, filledpolygon, drawcurve** and **filledcurve** establish a link the between field values (*floats)* of a given scalar and graphic objects, enabling the visualization and graphic modification of these values. These opcodes are to be placed inside of a *template*, accompanying the **struct** object. Each drawing instruction creates an *object* (a drawing inside the *datawindow* subpatch) for each existing scalar inside a *datawindow* list. One *template* can contain any number of drawing instructions.

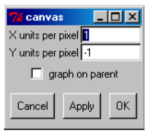


```
pd template_01

struct estructura_01 float x float y float k

drawnumber k k 10 0 k=
drawnumber x 0 20 220 x=
drawnumber y 0 30 220 y=
drawpolygon 0 10 0 0 k 0
```

Fig. 12 View of a template with a **struct** object and various drawing instructions.

CANVAS GEOMETRY
The size of each pixel in a window can be adjusted in the canvas properties box (left click over patch). The default values are X units = 1, Y units = -1, which means that the origin for the drawing instruction coordinates will be located in the top left corner of the window, as can be seen in the figure.



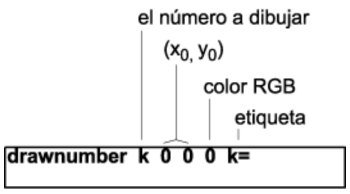


canvas
X units per pixel 1
Y units per pixel -1
graph on parent
Cancel Apply OK

These values can be changed to zoom in and out of a patch, or to change the position of the x and y axes. To use the (tricky) *graph on parent* (GOP) option, see Frank Barnecht's tutorial at <footils.org/cms/show/31>

Drawnumber. Draw a number in a specific location inside the window. The arguments are:

- The number to be drawn
- a pair of relative coordinates (x_0, y_0)
- RGB color[7]
- label (optional)



[7] RGB colors are expressed by a three digit number. Each number corresponds to a particular shade**,** 0 being the minimum, and 9 the maximum:

000 – black
900 – red
090 – green
009 – blue
999 – white

Once this object is created in the *template_01* subpatch, Pd will immediately draw the value for field k_i at the position (x_i , y_i) for each existing scalar (x_i y_i k_i) inside the *datawindow* subpatch [Fig. 13]. This means that, without it having been made explicit, the *x* and *y* fields of the scalar control the draw location of the variable *k.* This is because x and y are reserved variable names that, when are defined as scalar fields, control the absolute location of the associated graphic object(s).

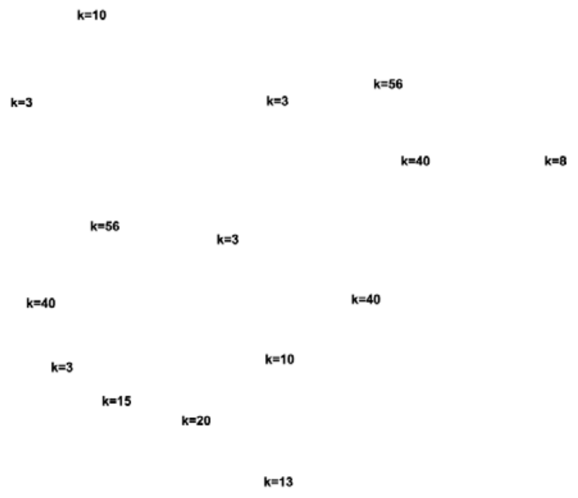
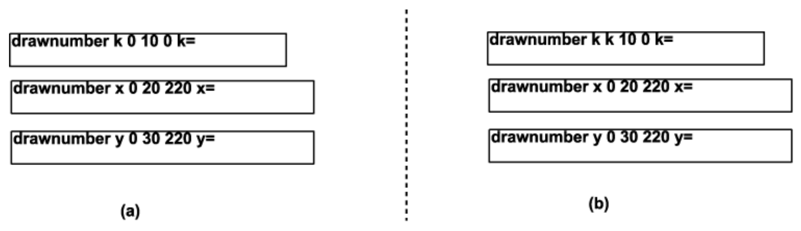


Fig. 13 Screen capture of the data window displaying the drawing instruction <**drawnumber k 0 0 0 k**>

Moving the mouse over the numerical value k_i , you can modify the value of the *k* field for each created scalar. You can verify how the *k* value for the specific scalars has changed by using the **get** object.

See the results of the following instructions on the data structure *estructura_01.*



(a)

```
drawnumber k 0 10 0 k=
drawnumber x 0 20 220 x=
drawnumber y 0 30 220 y=
```

(b)

```
drawnumber k k 10 0 k=
drawnumber x 0 20 220 x=
drawnumber y 0 30 220 y=
```

a) In addition to <*k*>, the first set of drawing instructions will yield two new graphic objects (<*x*> and <*y*>) showing the coordinates for the variable *k* [Fig. 14]. Looking at the example, you can see how the location of each number is obtained by adding the scalar field values, or absolute coordinates (x_i , y_i), with the values of the relative coordinates (x_0, y_0) as are defined in the relevant drawing instruction object's arguments. Therefore, <k=k_i > is drawn in the position (x_i , y_i + 10), <x=x_i > is drawn at (x_i , y_i + 20), and <y=y_i > at (x_i , y_i + 30). This behavior can be described by the general formula:

$$\boxed{(x_i + x_0, y_i + y_0)}$$

x_i , y_i are the values of the *x* and *y* fields for the scalar *i* (absolute coordinates)
x_0, y_0 are the *x* and *y* values defined in the drawing instructions object(s) (relative coordinates)

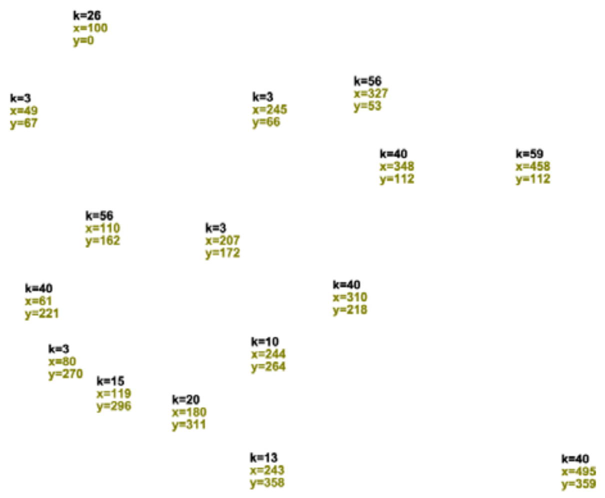


Fig. 14 Screen capture of the data window displaying example a).
The variables x, y are the location coordinates of object k.

b) In the second example, the x_0 value is substituted by the k variable. Now the location of the object <k= k_i > depends on the the value of k_i . You can see how its position changes when we move the k value in the data window. In this case the location of <k= k_i > in the data window is described by ($x_i + k_i$, y_i).

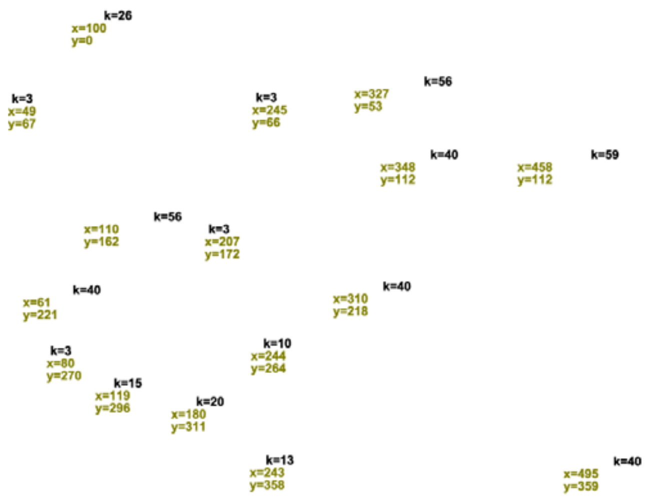


Fig. 15 Screen capture of the data window with the drawing instructions in example b).
Note the position change of object <k> = ($x_i + k_i$, y_i).

As you can see, float fields defined in the struct object can be used as arguments in any of the drawing instruction objects, linking list data with a graphic representation.

Changing to edit mode (CTRL+E) in the data window you can graphically move, cut, paste, copy and delete scalars from the list. Note how the numeric values change when the objects are moved around the screen.

drawpolygon. Draws a series of connected straight-line segments.

- RGB color
- line thickness
- two or more coordinate (a, b) pairs

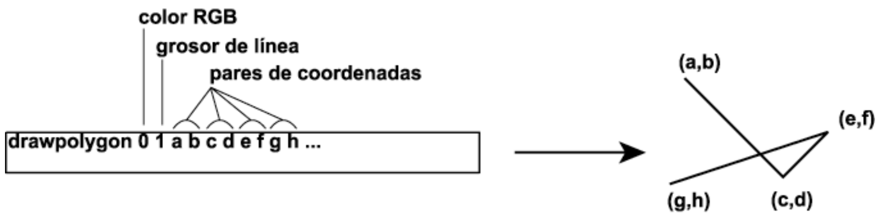


Fig. 16 The **drawpolygon** object with its arguments. The figure to the right shows the result of the drawing instruction - a graphic object that corresponds to a scalar.

Like **drawnumber** any numeric argument can be substituted with data structure variables.

- If all of the numeric arguments are constants, the resulting drawing will not be linked to any scalar.

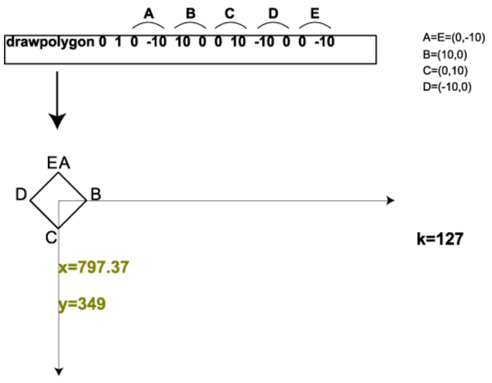


Fig. 17 Drawing of a static shape (not linked to any scalar fields).

- When a drawing instruction argument is a variable we can change its value graphically.

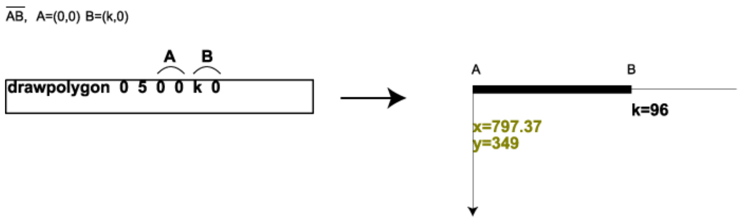


Fig 18 Drawing of a horizontal slider linked to the *k* field of the scalar.

This way [Fig. 18], we have control of the *k* value in the form of a 'slider'.

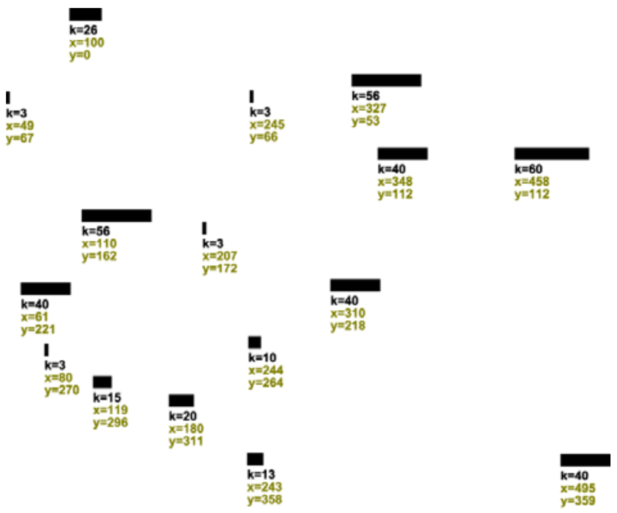
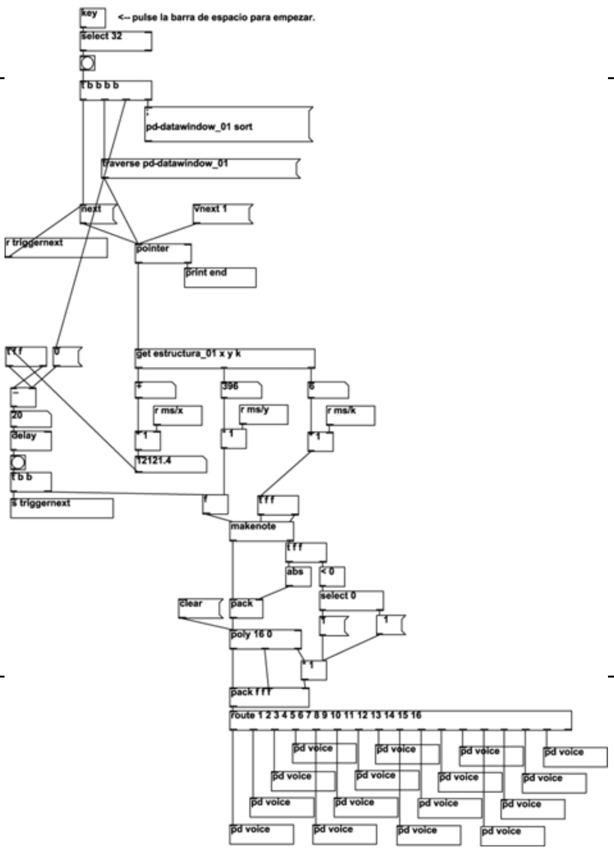


Fig. 19 Screen capture of the data window showing the drawing instruction **drawpolygon 0 5 0 0 k 0.** This object constructs horizontal line segments whose length is determined by *k,* mimicking a slider.

- When the same variable controls two or more values, we can change the variable's value by modifying any one of the graphic objects; all of the attributes linked to that scalar field will change simultaneously (this would occur, for example, with the drawing instruction **drawpolygon 0 2 k 0 0 0 0 k**).

It should always be remembered that the final location of a drawing is determined by the sum of the relative coordinates (as defined in the arguments of the drawing instruction(s): ai , bi , ci , di ,...) and the absolute coordinates (the values of the x and y scalar fields).

An **EXAMPLE APPLICATION** of the graphic interface of the data structure in Fig. 19: A 16 voice sample sequencer that manipulates an audio file loaded into a table. The graphic objects control the position and duration of each grain and the file read point. Analyze the **delay** based sequencing mechanism and the voice allocation system. **[tut_019.pd]**



drawcurve. Using coordinate pairs, draws a curve. This is similar to drawnumber. The arguments are:

- RGB color
- line thickness
- two or more coordinates pairs (a, b)

Sending a <u>sort</u> message to the datawindow: Orders the scalars by increasing value according to argument x. In the last example, before executing the score, we reordered the scalars in ascending order according to the time of initiation (in this example the variable x is used to specify the time the sample is set off).

```
;
pd-datawindow_01 sort
```

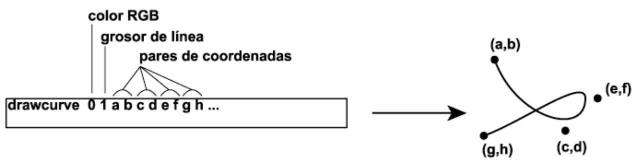


Fig 20. **drawcurve** and arguments

filledpolygon & filledcurve. These objects draw closed geometric figures. The arguments are the same here as for drawcurve and drawpolygon except for the addition of the interior RGB color.

- interior RGB color
- RGB line color
- line thickness
- two or more coordinate pairs (a, b) that determine the location of the vertices.

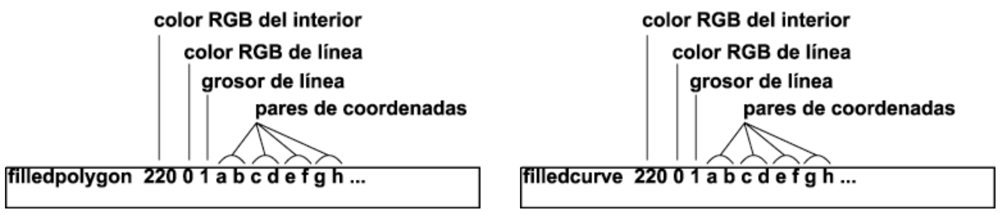


Fig. 21 Arguments for **filledcurve** & **filledpolygon**

Saving and Opening score files: The contents of the data window is saved when the whole patch is. But you can also send a write and read message to the data window to save the contents to a separate file.

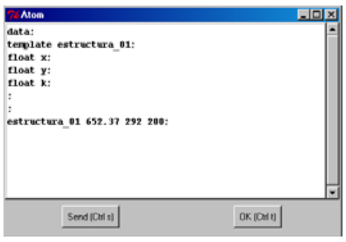
- Send a *write [nameoffile.txt]* message to the pd-datawindow:

```
;
pd-datawindow_01 write partitura.txt
```

-Send a *read [nameoffile.txt]* message to the *pd-datawindow:*

```
;
pd-datawindow_01 read otrapartitura.txt
```

Accessing data via the graphic objects: In addition to the methods we have seen for graphically manipulating scalar values, opening the properties window of a graphic object (right click/properties) will show a listing of each field belonging to that scalar and the values contained within. These can be edited.



```
data:
template estructura_01:
float x:
float y:
float k:
:
:
estructura_01 652.37 292 200:
```

* Editing field values with the mouse: click and drag the graphic object's control field.
* In edit mode (CTRL-E): move, copy, cut, paste, and delete.

Proposed Project

For the previous example patch, make the speed of reproduction of each grain, and as a result, the height and duration, controllable via the score. A possible score accomplishing this could be:

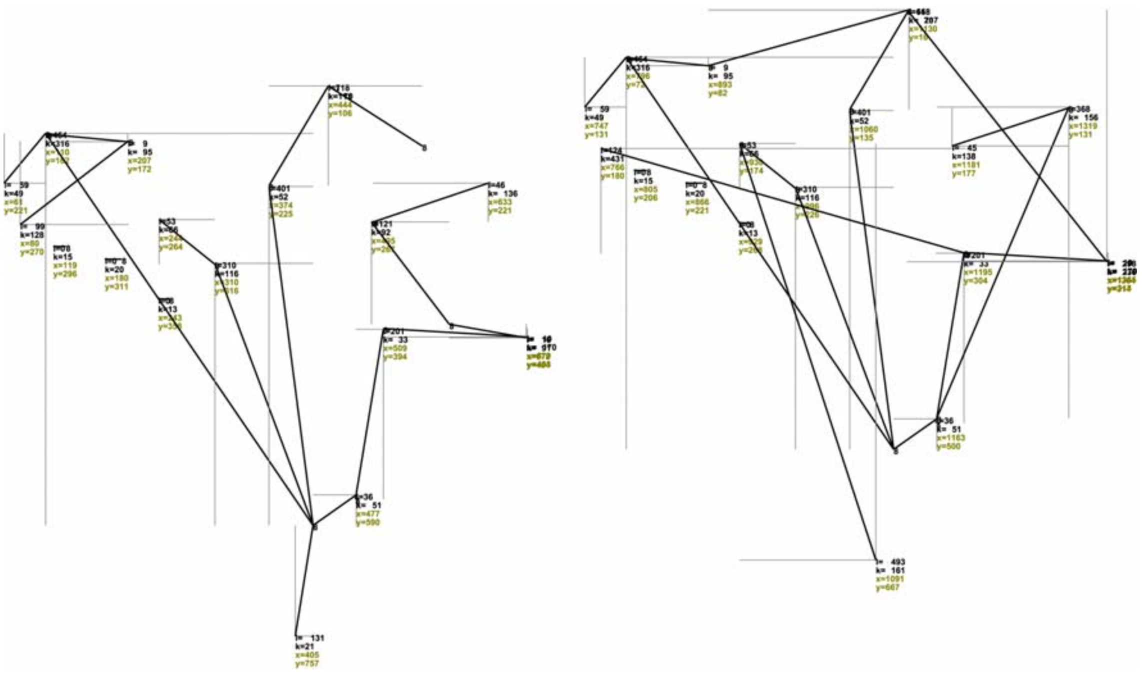


Fig. 22 Possible score for the proposed project

To do

Nested structures (array type fields), plot, vnext, etc...

Translation by: Juan Aboites
jaboit@gmail.com