Interactive Displays for End-Users: A Pluto Tutorial

Shamim P. Mohamed

TR 93-16

Abstract

This is a tutorial for Pluto, a system that allows end-users—technically sophisticated non-programmers—to design and implement graphical displays of data. Presenting data graphically can often increase its understandability—well-designed graphics can be more effective than a tabular display of numbers. Most visualisation systems to date, however, have allowed users to only choose from a small number of pre-defined display methods. They also present a static display—users cannot interact with and explore the data. The more innovative displays and the systems that implement them tend to be extremely specialised, and closely associated with an underlying application. Pluto is a system that enables users to specify the displays to be drawn in a flexible manner. It provides facilities to integrate user-input devices into the display, encouraging an exploratory approach to data understanding. The specification takes the form of a visual language that also provides means for repeating elements of the display a variable number of times based on the amount of data to be displayed, and for conditional structures that can depend on either user input or the data itself.

June 17, 1993

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

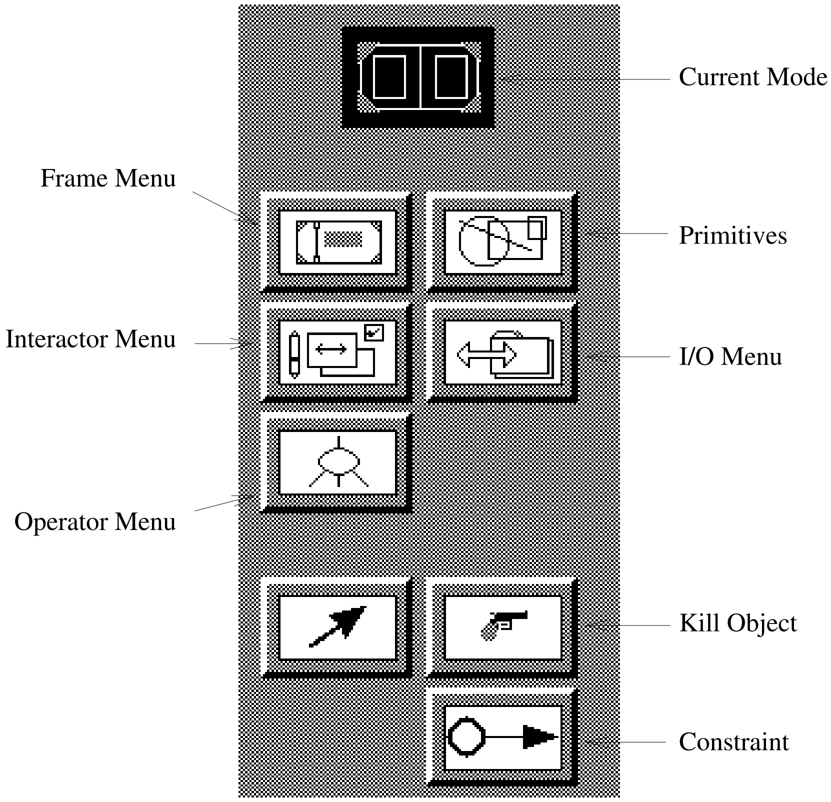


Figure 1: The Pluto main menu

1 Introduction

Pluto is a system to allow users to easily design and implement graphical displays of data. It uses a visual language to filter interesting data and to specify displays of that data. Pluto is especially useful if the display you need to draw is not offered by any of the off-the-shelf graphing packages, or if you're a programmer and want to quickly make up a graphical display for your program without having to worry about window systems or programming.

When Pluto first starts up, it draws a window with a menu and a drawing canvas. The menu is used to tell the program what you are going to do next. After selecting a command from the menu, you can perform the action (whether it's creating new objects, removing objects etc.) on the canvases.

The menu (Fig. 1) has two parts. At the top, it shows the current mode on a black background. Below it are the mode selectors, in two groups. The top group consists of a set of pull-down menus, to handle the placement of objects and system functions; the bottom group comprises buttons that select other modes—constraints, deletion of objects or "normal" mode, where objects can be moved around, re-sized etc. to clean up the windows,

In the rest of this tutorial, we will follow the specification of a simple histogram, and an annotated scatter-plot of data.

2 Use and Terminology

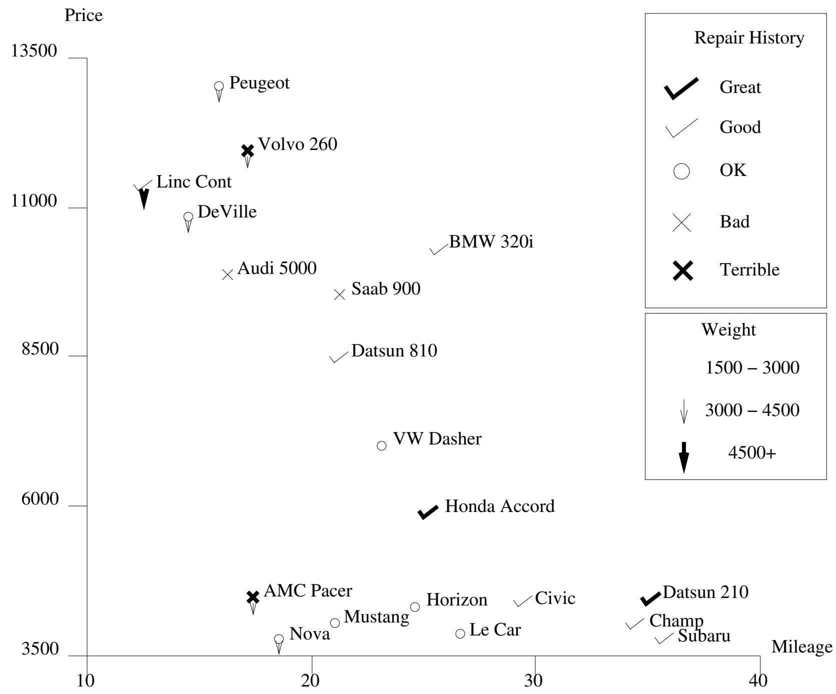
Very often, data is arranged in a table. Each row of the table can be considered a datum, with its component values being named in the column headings. Here is an example, showing price, mileage, weight and repair histories for a collection of cars from 1979:

Car	Price	Mileage	Weight	Repair
Datsun 210	4000	35	2100	Great
Accord	5799	25	2240	Great
Pacer	4749	17	3350	Terrible
...				

Table 1: Cars of 1979

The values for one car, say "Datsun 210," are one item of data, and it has as its components, values "price" (an integer), "mileage" (an integer), "weight" (an integer) and "repair" (a string, or one of four possible values).

This collection of data in this table can be displayed much more clearly if we represent it as a scatter-plot. Consider this display of the same data:



Cars of 1979

The graph shows relationships between cost and mileage very clearly. It also brings out certain other relationships—like the partitioning of the cars into three classes.

Let us now look at how this scatter-plot can be specified in Pluto.

3 Overall Organization

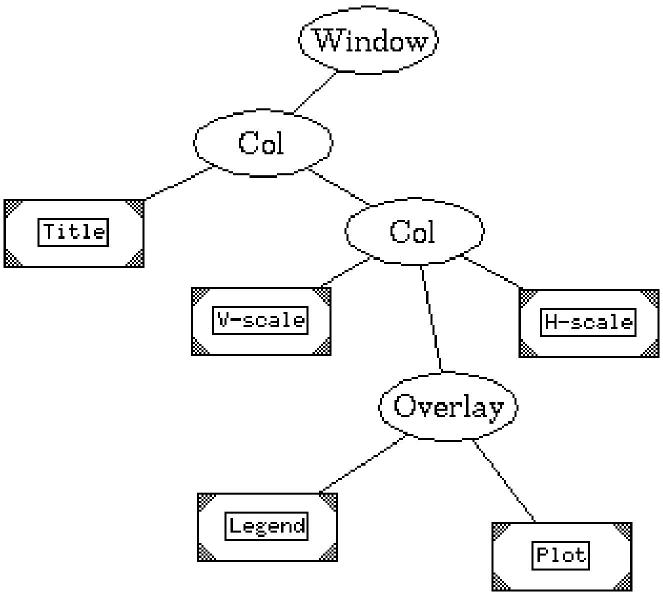
There are three parts to the Pluto specification:

- the query filter
- the operator tree
- the GLS specification

The operator tree describes the overall layout of the display to be drawn. In this case, there are four basic components to the picture.

1. The title
2. The horizontal and vertical scales
3. The legend, describing the symbols used
4. The points of the scatter-plot

This is the operator tree that we need to construct:



This is called a tree because it looks like an upside-down tree, with its trunk at the top and the branches and leaves at the bottom. Each oval in the picture (called an *operator* represents an operation on more basic elements. For instance, the operator marked "Column" takes the pictures from the two lower operators, arranges them in a column and passes on this new picture to the operator above it.

In this specification, the picture displayed is composed of two elements arranged in a column: the scatter-plot with attached scales, and the title. The scatter-plot itself is made of three objects—the horizontal and vertical scales and the actual points of the plot (overlaid with the legend)—arranged together by the operator marked with the little diagram.

We separate the legend from the points because we want to be able to position it in a picture at some suitable place—depending on the data, we will choose some unoccupied area of the display for it.

The points of the scatter-plot itself are made by the box in the middle called "plot." This is called a "leaf," because it's a the end of the tree. It is a rectangle rather than a box, because it is described in more detail in a system called *GLS*. It has a small rectangle in it, which is a field to enter the name of the object. To edit any such text field, click inside it with the mouse and start typing.

The oval marked "Window" represents the final output to the screen.

4 The Graphical Layout System

The GLS (Graphical Layout System) desribes in detail how objects are arranged in the displays. Every object is represented by a rectangle, with some distinguishing characteristics. These are the basic objects of the system:

Lines these are represented by rectangles with a line joining two opposite corners.

Rectangle these are just plain rectangles

Buttons these are rectangles with images of buttons in them. There are two kinds of buttons—push-buttons and on-off buttons. These are represented by mnemonic images.

Sliders these are objects that can be used to describe a value between two extremes by a sliding a thumb-box. These can be horizontal or vertical.

These basic objects are placed on the display by selecting them from the menu and placing them in the drawing area of the display you want to build. Relationships between objects are described by arrows called *constraints*. Each constraint has a value and indicates that the object it's pointing to is that much to the right (or below) the object it's anchored at. In Fig. 2, the rectangle on the right is drawn 10 units to the right of the right edge of the other one. Notice how the edge is drawn with a double line—that means it cannot accept any more constraints, because it already has a value.

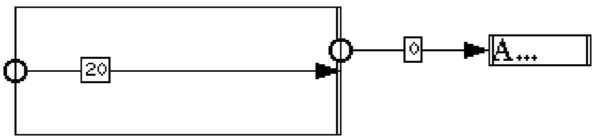


Figure 2: An example of constraints

A constraint can only be drawn between edges of the same type, i.e. vertical or horizontal. To draw a constraint, click on the "constraint" button in the main menu. On the GLS window, press the mouse button down near the edge of an object, and holding the button down, move the mouse. A line appears drawn from the starting point, and tracks the position of the cursor. At the same time, a cross appears on the edge of the object that the drag was started at (Fig. 3). Notice how the drag does not need to be started exactly on the side of the rectangle.

To change the label on a constraint, select the "normal" mode and click on the constraint. This will make the constraint visible. Now if you click on the text inside the label, a vertical line will appear, indicating that anything you type will be inserted there. If instead of clicking the mouse button, you drag across a section

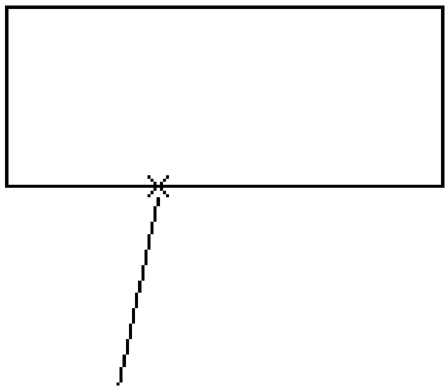


Figure 3: A constraint being created

of text, that part is drawn in reverse mode (white letters on a black background) indicating that anything you type will *replace* those letters.

Once a constraint drag is started, it can only be ended at some edges. For instance, some edges already have a value and cannot accept a constraint, and are drawn with a double line; others are the wrong type, i.e. vertical when the drag was started on a horizontal edge or vice versa. To show you which edges are ok, if while dragging at constraint creation you approach close to a legal edge, the line will "snap" to that edge, indicating that if you were to let go of the moouse button, a constraint would be created.

There is another class of objects called *frames*. These are drawn as rectangles with shaded corners. They establish a reference for the display in each GLS window. They behave just like the basic objects described above, except that they are not drawn on the final display.

Another type of object that is closely related to the frames but is not drawn on the final display are reference lines. A reference line is attached to a frame and can be used to neaten up a specification. It can also be used to maintain proportionality—in Fig. 4 the toggle's left side will always be halfway between the frame's sides.

To attach a reference line to a frame, select a reference line type from the "frame" pull-down menu. Now start a drag near the edge of a frame. A line appears that tracks mouse movement but stays attached to the frame at its endpoints. In the case of a "proportional" line, it moves in a "jerky" manner, snapping to some special values—these are the values that represent being 1/10, 1/5, 1/4, 3/10, 1/3, 2/5, 1/2, 3/5, 2/3, 7/10, 3/4, 4/5 and 9/10 of the way across the frame. (If you want an arbitrary value that is not one of these, hold down the "Control" key while dragging—then the line will track the mouse smoothly.)

There is also a plain reference line that can be used to reduce clutter. Also notice how some of the labels for the constraints are not drawn—again, to reduce clutter. The labels can be switched on or off by clicking on the constraint in the normal mode.

Another way of neatening up and organizing large specifications is to group logically related objects together and hiding the details. This can be done with a "parent-child" type of frame. If we were drawing a plain scatter plot, with just labelled points, we can separate that from the rest of the specification (Fig. 5). Now the window labeled "labeled point" separates the composite object—a point (a rectangle of zero width and height) and label (text drawn 10 units to the right of the point)—from the rest of the specification, where it is drawn as a grey box with the name of the composite object. (When a parent-child object is created, its name is empty; click on the name—the small rectangle—and type in its name.)

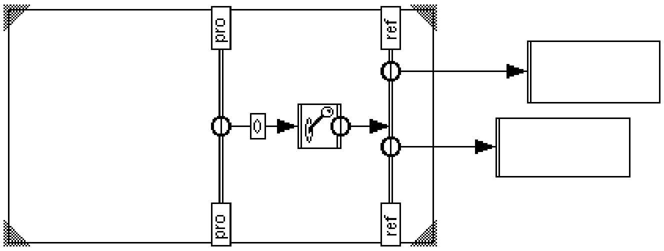


Figure 4: Examples of reference lines

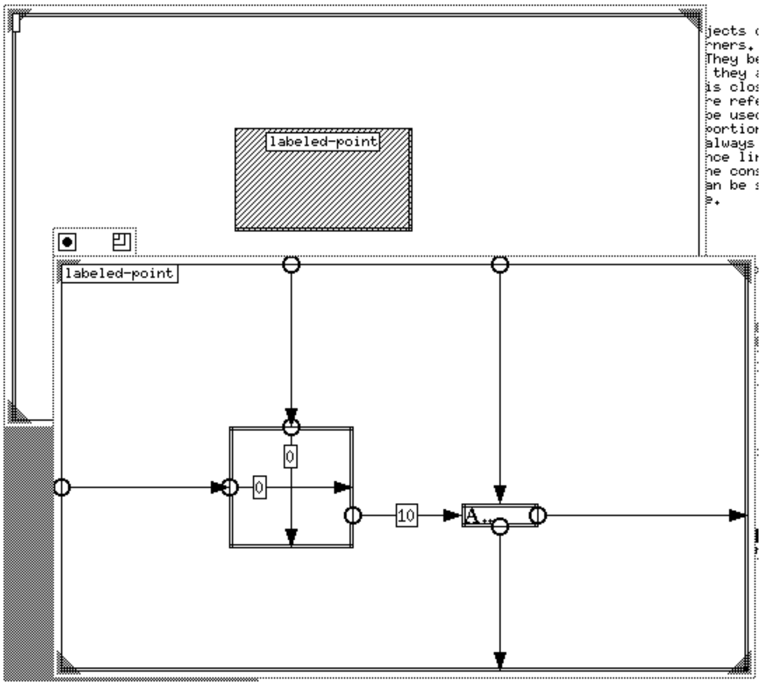
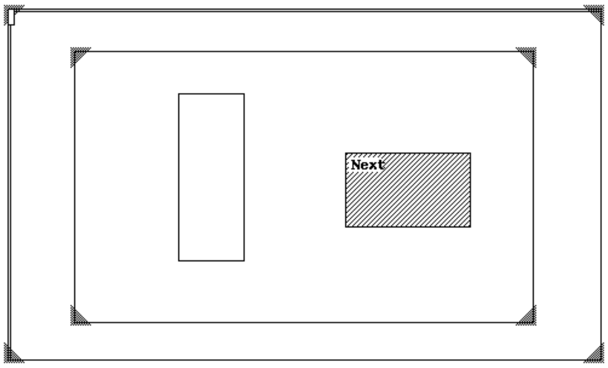


Figure 5: Grouping in a parent-child construct

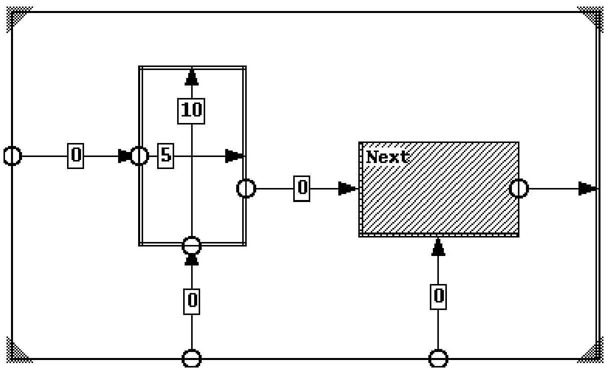
5 Repetition

In the previous sections we saw how to build simple displays. Most displays feature a repeated section. For instance, in a scatter plot, the points are repeated, one for each data item; in a histogram, rectangles of varying width are repeated. How can we do this in Pluto?

We use another form of frame: it is called a "repeat frame." Here is an example of how to draw many rectangles:



This will draw one rectangle for each data item that is to be displayed. A loop frame repeats everything that is inside it. We also need to express relationships between these rectangles—we do that with constraints. Each repeat frame has a shaded rectangle called "Next." This represents the next instance of this loop frame; in this case, a frame containing a rectangle. If we want all these rectangles to be the same size, and lined up next to each other, this is how we would describe that:

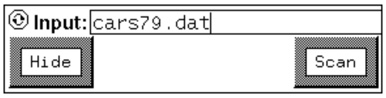


The rectangle and the next one (represented by the grey box) both have the lower edges at the same height. The next unit's left side is 0 units away from the right edge of this rectangle; and since the left edges of the rectangles are all 0 away from the left edge of the repeat frame, all the rectangles are lined up in a row with no space between them. Therefore in a sense the "Next" box consists of all the remaining units, since it contains a unit with another "Next" box, and so on.

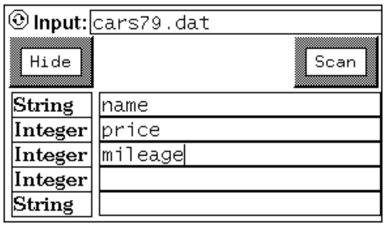
Now if the height of each rectangle depended on the data we were displaying, we would have a simple histogram.

6 Incorporating the Data

Since each loop frame repeats its included objects as many times as necessary to display the data, it needs to know where that data is coming from. So each loop frame has a little dialog box with field for the source of the data. To display this box, hold down the "Control" key and click near the edge of a repeat frame. A little window pops up, asking for the input source:

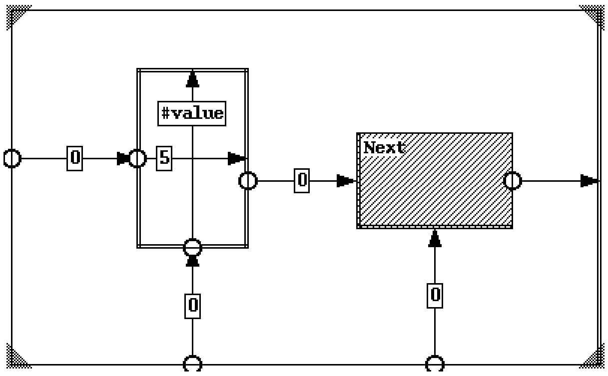


To specify the source, click the mouse inside the field next to where it says "Input:" and type. Then if you click on the button labelled "Scan," it looks at the file you named and figures out what it consists of, and gives you the types of values in each field, along with text-entry devices to enter the name of each field:



Now these names can be used in a GLS specification. In the label for a constraint, an incoming value can be used by prefixing its name with the #' character, i.e. to use the price from the query in Fig. 6 one would use `#price`.

To specify a simple histogram, we need only to modify Fig. 5 so that the heights depend on a data value, say `#value`:



Now this histogram can be put together with scales etc. to form the final display.

Another use of the repeat frame is to repeat something a fixed number of times. For instance, if you wanted to create a scale for a graph, you might want ten labels and short vertical lines next to a long horizontal line. To do this, click on the small button next to the label and it (the label) changes to "Repeat:". Now type in the number of times you want the objects in the repeat frame repeated.

7 Annotated Scatter Plot: An Example

Let us now complete the specification of the scatter plot that we started in Fig. 3. The GLS specification to draw the points for the scatter-plot starts with a repeat frame, that plots a point at the (price, mileage) point. Let us represent the actual point by a sub-picture, by using a parent-child frame. This is the first step:

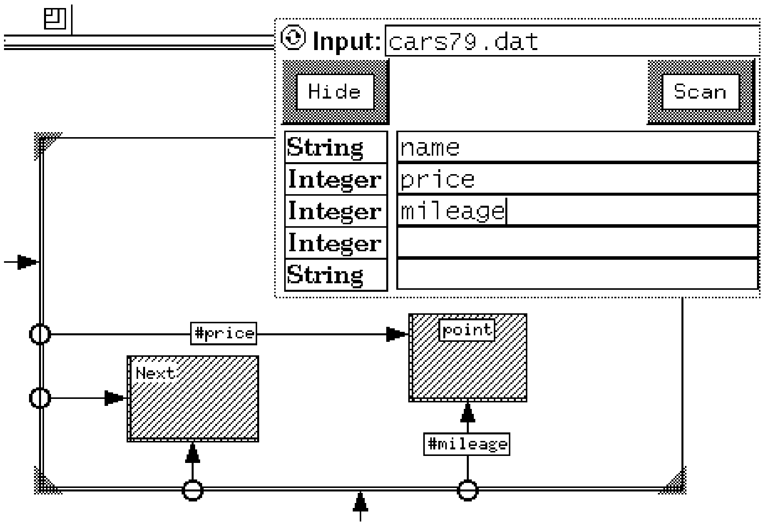


Figure 6: The repeat frame for a scatter plot

The box marked "point" represents the plot of each data point. Since the "Next" box is in the same position as the current frame, all the frames are at the same position; therefore each instance of the "point" box will be placed at its respective (price, mileage) point.

We can break down each point into two main components: the icons representing the car (and some of its characteristics) and a text label for its name. The text label takes its value from the field named "name":

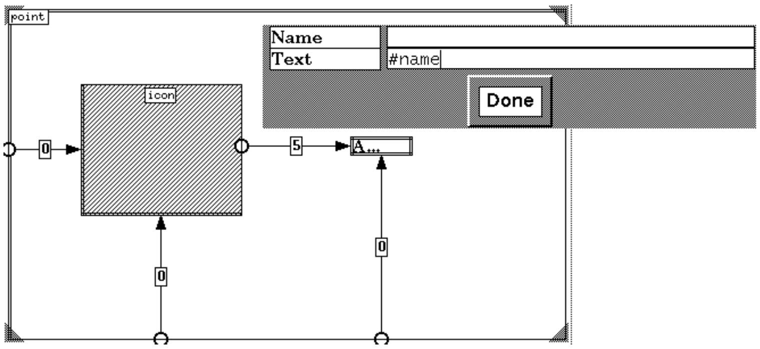


Figure 7: The point and its text label

(The "Name" attribute of the text display object can be ignored for now. Use the "Done" button on the attribute editor to remove it from the display. It can be brought back by pressing the mouse button on an object while holding down either the Shift or the Control key.) Now the box named "icon" can be fully described.

The appearance of the point representing each car depends on the values of the variables "weight" and "repair". Depending on the "repair" field, one of five symbols is placed at the right point; then another symbol is placed below this one depending on "weight." For this, we need to introduce another type of frame: the *conditional frame*. This is a frame that displays one of many alternatives depending on the value of some expression.

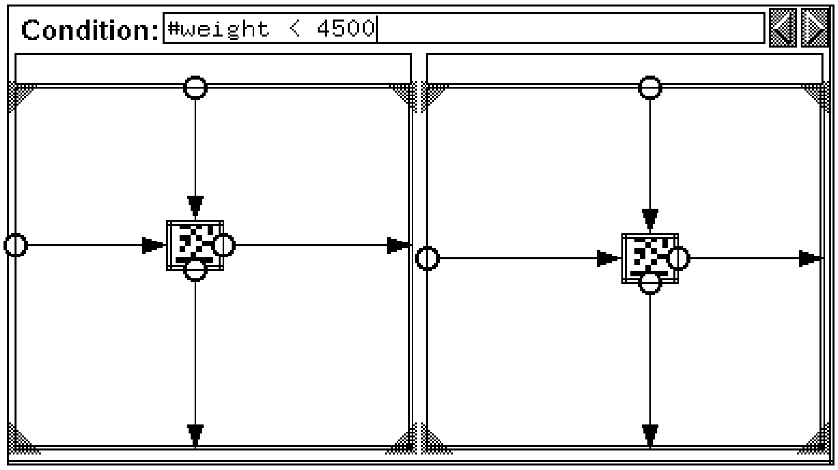
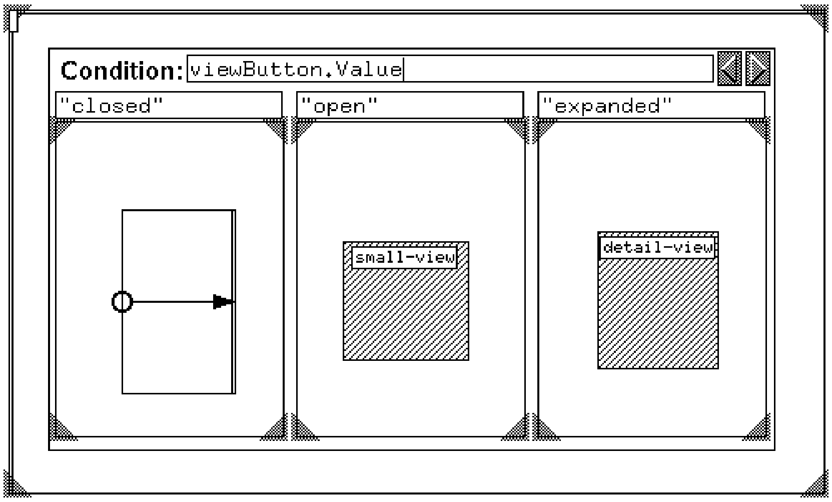


Figure 8: A conditional frame

This is a very simple conditional. Depending on the value of `weight`, one of the two possible bitmaps is chosen. In this simple case, if the condition is true, the alternative on the left is chosen; if false, the one on the right.

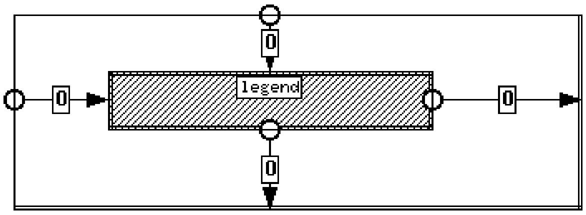
The conditional frame can be more elaborate:



In this example, depending on what "viewButton.Value" was, one the three frames would be chosen. If it was "closed" a rectangle would be drawn; if "open" the object described by "small-view" and if "expanded" the object named "detail-view would be used.

The buttons on the top-right corner control how many cases there are. The one on the right increases the number of child frames, the one on the left removes the rightmost child frame.

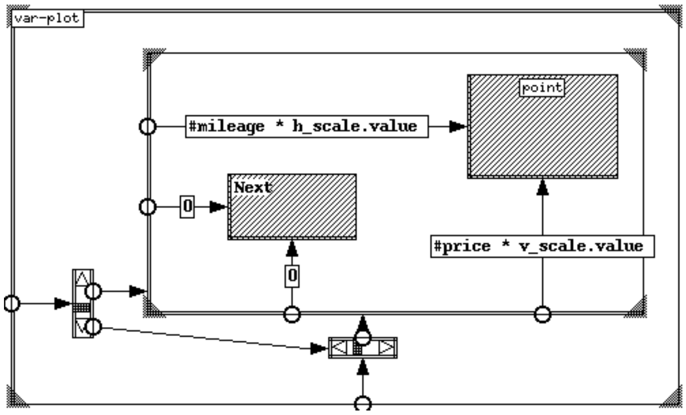
The legend for the histogram is an image (of the symbols used, along with descriptive text) that should be free to move under user control. To do this, we use the fact that for rectangles and lines, any unconstrained edges are under user control, i.e. they can be freely moved. This is how we arrange that:



The right and bottom edges of the rectangle are constrained to be the right and bottom edges of the legend, so that the rectangle is always the same size as the legend. Now after the display is drawn, since two edges of the rectangle are free, it can be picked up with the mouse and moved, and the legend will move along with it; therefore it can be placed wherever we wish.

8 Advanced Displays

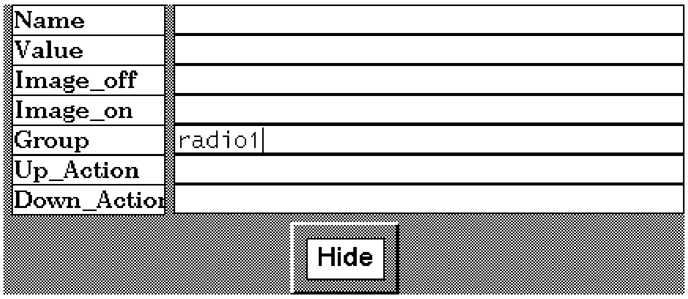
For some displays, more intricate control arrangements are required. For instance, we might decide that we want the scale of the histogram to be flexible, to be changed after the display is drawn, and we want to use two sliders, one for dimension. This is where we need to name objects. Every interactor has a component called `Value`, which can be referred to in any constraint or attribute editor field (Fig. 7. We then set up two sliders, one horizontal and one vertical, with the repeat frame in Fig. 7:



Now as the sliders are moved, the scale of the scatterplot will change dynamically. This can be used for many different types of displays, not just to control the visibility. For example, you may wish to control the appearance of an object on the basis of some input values from buttons or sliders. Or if you are exploring the relationships between data sets with many components, you can indicate which components should be plotted against which other, by selecting them with buttons.

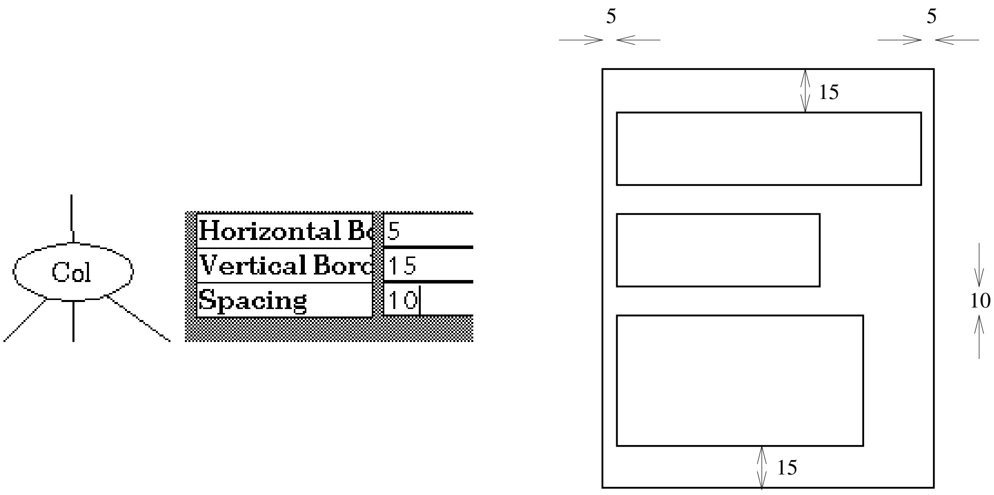
We have already seen another example of this type of use: in the example of conditional frames (Fig. 8) the expression was “viewButton.Value”—this refers to a component called “Value” belonging to an object called “viewButton” which in this case is a “radio button”—that is, a group of buttons that have the property that if any one button is “on” all the others will be “off.” This means that depending on the status of “viewButton” the data will either be represented by a simple rectangle or by the more complex views.

To create a radio button, first select “Toggle” from the “Buttons” pull-down menu under the “Interactors” pull-down menu. Now place as many buttons as you need for the radio button, and arrange the constraints for the presentation you want. Now, for each button, bring up the property list (by holding down the “Control” key and clicking on the object):



In the field labelled “Group” enter a name for this group of radio buttons. Now all the toggles that belong to the same group will behave like a radio-button. The name of this group can now be used as an object, with a “Value” field. In this case, the expression “radio1.Value” will indicate which of the buttons in the radio button group has currently been selected.

The operators in the tree also have additional parameters that can be brought up by clicking on them while holding the control key down. All operators have parameters for borders. The “row” and “column” ones also have one called “spacing” that controls the distance between the images that are put in a row (or column).



9 Saving and executing the specification

When done with the specification (or just to see the resuts of a partial spec.), select “Try It” from the I/O menu. Pluto will then invoke a system called Penguims to draw the display just specified. In another window, it will draw the specification you have entered so far. Any values that have not been constrained will take their values from their position in the GLS spec. to give you a rough idea of what it will look like in the final picture.

To save a spec, select “Write” from the I/O menu. A window will pop up asking you to type the name of a file. (If you have already done this, the name you used the earlier time will already be in the filename field.) Click in the field and type in the name of the file, and hit the button marked "OK." The spec will be saved to that file.

To run a spec that you have saved, you don't need to invoke Pluto again; there is a program called `ppc` that will run it for you. Simply start up ppc with the name of the saved file as the argument, and it will run the spec.