Some Box Design Issues in PWGL

Mikael Laurson[1] and Mika Kuuskankare[2]

[1]Center for Music and Technology, [2]Department of Doctoral Studies in Musical Performance and Research, Sibelius Academy, P.O.Box 86, 00251 Helsinki, Finland
email: laurson@siba.fi, mkuuskan@siba.fi

Abstract

This paper gives an overview of how boxes are created in PWGL. PWGL is a visual language based on Common Lisp, CLOS and OpenGL. PWGL boxes can be categorized as follows. Simple boxes define the basic interface between PWGL and its base-languages Common Lisp and CLOS. Visual editors constitute another important subcategory of PWGL boxes. Finally, more complex boxes can be used to create PWGL applications ranging from simple ones to complex embedded boxes that can contain several editors and other types of input-boxes. We discuss the components of a PWGL box, how boxes are constructed and give some remarks on how to define the layout of a PWGL box.

1 Introduction

PWGL (Laurson and Kuuskankare 2002) is a novel visual language based on similar concepts than PatchWork (PW, Laurson 1996). PWGL has been designed from scratch and it contains several improvements when compared with PW. The graphics part of the system has been realized in OpenGL (Woo et al. 1999). OpenGL offers several advantages such as multi-platform support, hardware acceleration, floating-point graphics and sophisticated 2D and 3D-graphics. Thus the PWGL system offers new potential that can be used to design more refined visual systems.

Like PW, PWGL is a multi-window system. A PWGL window is called a patch. A patch, in turn, contains boxes and connections. In the simplest case a box is a visual equivalent to a Lisp function or method. It has a number of input-boxes - containing typically constants such as numbers or lists - and one or several outputs. When evaluated a box reads its inputs, calls a function or method associated to it and finally returns a result. Connections are used to define relations between boxes. An output of a box can be connected to an input-box of another box. Thus the system works in a similar fashion than Lisp where function calls can have as arguments either constants or functions calls.

PWGL has a library of predefined input-boxes which typically handle numbers, lists and popup-menus. PWGL has also an important subgroup of input-boxes that are associated to editor-windows. These editor-windows contain complex objects, such as scores, chords, break-point functions, bezier functions and sound samples. These input-boxes can be opened and inspected or edited by the user.

PWGL offers several ways to construct boxes ranging from completely automatic to methods that allow to specify the exact type of input-boxes, default-values and layout options. All PWGL boxes can be resized both vertically and horizontally. This option adds new requirements to our system. It has to deal with boxes that are not just simple fixed-sized rectangles. Instead, boxes have to behave in a coherent manner after the size of the box has been changed.

The paper is organized as follows. First, we briefly give a general overview of a PWGL box and enumerate the main features that are shared by all PWGL boxes. The next section deals with the creation of boxes. We start with simple Lisp definitions and go over to more complex options that allow to define a box in a more precise manner. Then we discuss a method which allows the user to define a box in great detail. The user can specify using various layout options how the input-boxes will be distributed within the box and how the input-boxes will respond when the box is being resized. We end with a complex embedded application box example.

2 PWGL Boxes

2.1 Box Components

This section discusses the main components of a PWGL box (Figure 1). A box consists of a main-box and a number of input-boxes. The function-name is given above the top-left corner of the main-box. The bottom-right corner contains a zoom area allowing the user to modify the size and shape of the box. The user can evaluate the selected box by typing the character 'v'. If there are several outputs the user can select the desired one directly by clicking the output triangle. If no output is selected then the left-most output is used.

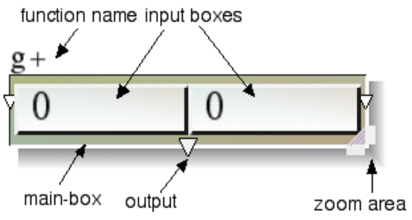

Figure 1. PWGL box components.

When the user moves the mouse above a box the cursor changes its shape depending on which part of the box the mouse is currently located in (Figure 2). Figure 2 shows also the actions that will occur if the user clicks the mouse and starts to drag it.

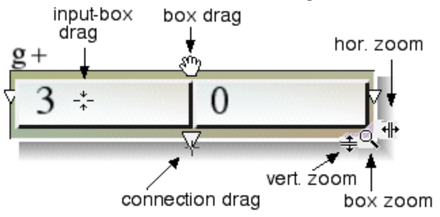

Figure 2. Various cursor shapes and the associated actions of a PWGL box.

2.2 Lambda-list Keyword Support

PWGL supports automatically the most commonly used Common Lisp lambda-list keywords (i.e., *&optional*, *&rest* and *&key*, Steele 1990). In the simplest case where a Lisp lambda-list contains only the required arguments - Figures 1 and 2 show an example of such a box having 2 required inputs - the system generates automatically one input-box for each required argument. In the case of keyword arguments the box is extendable and contains a downward pointing arrow at the bottom-left corner of the main-box. Figure 3 shows a box that represents the Lisp function '+' that has in the argument list the keyword *&rest* (i.e., the box can have an arbitrary number of input-boxes). Figure 3 shows instances of the '+' box having 1, 2 and 4 input-boxes:

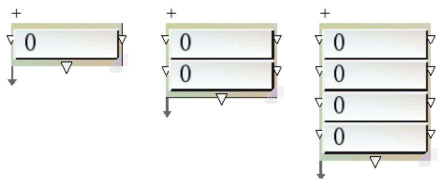

Figure 3. Extendable boxes of type *&rest*.

Figure 4, in turn, gives a more complex example using the Lisp function '*position*' that has 2 required arguments and 6 *&key* arguments. The left-most box contains only the required arguments while the one to the right has one *&key* argument. The *&key* arguments always extend the box with 2 input-boxes where the first one is a popup-menu indicating the keyword (here *:key*) and the latter one giving the value for this keyword (here *first*):

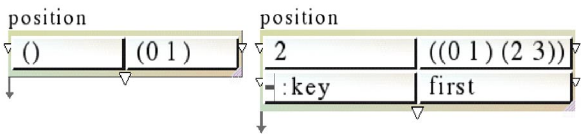

Figure 4. Extendable boxes of type *&key*.

3 Box Creation

There are three different schemes that can be used to generate PWGL boxes. In the first one the user simply defines a Lisp function using the standard macro *defun*. The system generates automatically the corresponding PWGL box using the knowledge of the underlying Lisp system. For instance, let us assume the following Lisp function:

```
(defun add3 (a b c)   "simple add" (+ a b c))
```

The function can be converted automatically to a box by typing the name of the function in a dialog box (see the resulting box to the left in Figure 5).

The second and somewhat similar approach to create boxes consists of using a PWGL macro called *PWGLDef*. The most important difference between *defun* and *PWGLDef* is that in the latter case the user can specify the input-box type and the default value for each argument. Furthermore, *PWGLDef* accepts a list of extra keyword/value pairs that allow to define the outlook and behavior of a box in more detail. Let us assume that we would like to change the previous box definition in two ways. First, we give default arguments for each input. Second, we change the default grouping so that the box would consist of a column of 3 input-boxes. These changes are achieved by the following definition (the corresponding box can be found to the right in Figure 5):

```
(PWGLDef add3 ((a 0)(b 2)(c 4))
 "simple add"
 (:groupings '(1 1 1))
 (+ a b c))
```

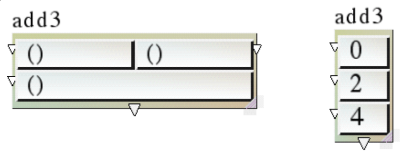

Figure 5. Two boxes representing the function '*add3*'.

The third method to create boxes consists of using the *mk-box-function* method. Here the user can specify the required input-box types, default values, outputs and layout-options in the most detailed form. We give next the code to create a complex box with 3 editor input-boxes each having a predefined initial state, 1 horizontal slider and 3 outputs. The resulting box can be found in Figure 6.

```
(defgeneric test-box () (:documentation "this is a test-box"))
(defmethod mk-box-function ((self (eql 'test-box)) x y)
  (mk-PWGL-box  'PWGL-box self "test-box" x y 1.0 0.9
      (list (mk-2D-subview  :application-window
            (mk-2D-application-window
                :2D-subviews (list (mk-bpf '(0 2 3) '(0 1 0)))))
            (mk-chord-subview :application-window
                (mk-chord-editor-window '(60 64 67)))
            (mk-score-subview :application-window
                 (make-enp-window
                  '(((((1 (1 1 1 1))(1 (1 -1 1))(1 (1.0 3))))))))
             (mk-slider-subview :value 50
                 :minval 0 :maxval 100 :grid t :horizontal t))
       :proportional-coordinates
         '((1/12 1/12  5/9 4/10)     (8/12 1/12  2/9 4/10)
           (1/12 7/12  10/12 4/15) (1/12 11/12  10/12 1/20))
       :outputs (list "1" "2" "3")))
```

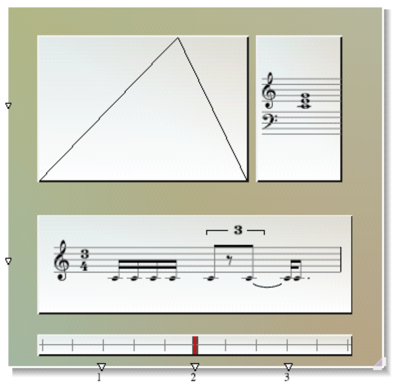

Figure 6. A box containing a 2D-editor, chord-editor, score-editor, slider and 3 outputs.

4 Box Layout

As all PWGL boxes can be resized special attention must be given to layout options as the input-boxes of a box cannot be positioned simply by using static x-y coordinates. The key point here is to use proportional values instead of fixed coordinate values. Similar kind of dynamically resizable objects - typically dialog windows - can also be found for instance in Mac OSX and in some programming environments such as the CAPI system (LispWorks CAPI User Guide).

When defining the layout of a box PWGL uses two sets of keywords:

(1) *:groupings, :x-proportions, :y-proportions*
OR
(2) *:proportional-coordinates*

In (1) the data lists *:y-proportions* and *:x-proportions* give proportional *delta-values* (the delta-values are scaled so that their sum equals 1.0 in order to guarantee that the subviews will always be inside the main-box).

The *:groupings* keyword is a list of values where each value gives the number of subviews for each row (thus a list (3 3) groups 6 subviews into two rows, where each row has 3 subviews). *:y-proportions* is a list of proportional delta-values defining the height of each row. If not given then all rows have equal height. *:x-proportions* is a list of lists of proportional delta-values. Each sublist defines the internal x proportions of the respective row of boxes. If not given then each subview within a row has equal width.

Option (1) is often easier to use than option (2) as it requires only a small amount of data to be functional. For instance the second version of the '*add3*' box example (see Figure 5) required only the groupings list (1 1 1) to define the layout of a box where the input-boxes form a column. There are, however, some restrictions. There can be no overlaps, no holes between input-boxes (holes can though be simulated with special subviews) and subviews are always aligned in horizontal direction.

In option (2) - using *:proportional-coordinates* - the data lists give proportional coordinates for each subview in the form: *((<x1> <y1> <w1> <h1>)) ... (<xN> <yN> <wN> <hN>))* where each sublist defines the proportional x- and y-position and proportional width and height of the respective subview (note: these values are not scaled). While the *:proportional-coordinates* option requires often more data than option (1), it has some advantages. Subviews can be freely distributed, they can be positioned outside the main-box and overlaps can occur. Figure 6 shows an example how to use the

:proportional-coordinates option to define a box layout.

Sometimes the use of pure proportional delta-values or coordinates leads to undesired results. A typical example is for instance a box containing sliders that function as scroll-bars. In this case it is probably more desirable if scroll-bars have fixed size in one dimension while other subviews are resized dynamically as before. This behavior can be achieved by using a mixed form of delta-values or coordinates. Whenever the system encounters a list consisting of the keyword :fix and a value, then the value is considered to be fixed and not proportional.

Let us assume a box consisting of two rows of subviews. The first row from the top contains a 2D-editor and a vertical scroll-bar and the second row, in turn, has a horizontal scroll-bar and a small button-subview (see the box to the left in Figure 7). If we use the following layout data:

```
:groupings '(2 2)
:x-proportions '((20 1) (20 1))
:y-proportions '(20 1)
```

we get a box - after resizing it horizontally - where the width of vertical scroll bar differs from the height of the horizontal the scroll bar (Figure 7 to the right, upper box). If, however we use the following mixed form of layout data (note the expressions starting with the keyword :fix):

```
:groupings '(2 2)
:x-proportions '((20 (:fix 0.03)) (20 (:fix 0.03)))
:y-proportions '(20 (:fix 0.03))
```

the width of the vertical scroll-bar and the height of the horizontal scroll-bar are always fixed to 0.03 units (see the lower box to the right of Figure 7).

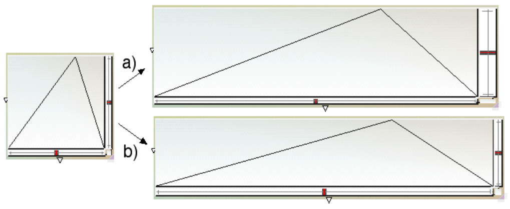

Figure 7. Two resized box versions with different layout data: a) pure proportional delta-values, b) mixed delta-values.

5 Recursive Boxes

Finally, a PWGL box can be recursive, i.e., it can contain instances of itself. This property allows to combine features described above into one complex application box. Figure 8 shows a main box containing 3 sub-boxes. Each sub-box can have its own background color, subviews and layout. This scheme is very useful as it permits to define a library of box components (similar to the library of basic input-boxes) that can be used as building blocks when constructing even more complex boxes.

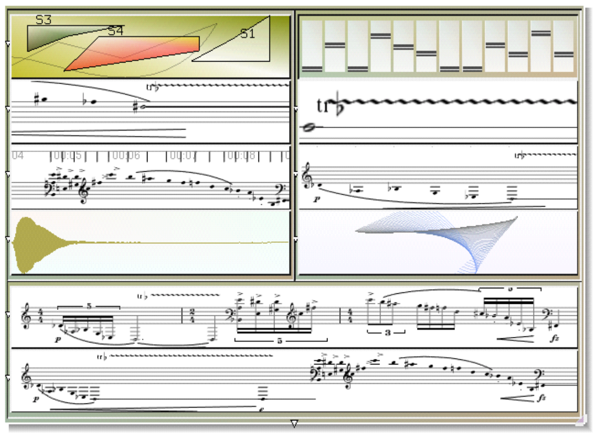

Figure 8. A complex recursive application box.

6 Conclusions

This paper gave a survey of visual PWGL boxes. We first presented the main components of a box. After this we discussed different options how to construct boxes and gave some ideas of available layout schemes. Although the system is already functional it can be extended and improved in several ways. One idea is to add more layout options that for example would allow to control in more detail how boxes respond to resize operations. The current system could easily be extended to support other types of mixed delta-values or proportional coordinates.

References

Laurson M. and M. Kuuskankare. 2002. "PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL". In Proc. of ICMC'02, Gothenburg, Sweden, pp. 142-145.

Laurson, M. 1996. *PATCHWORK: A Visual Programming Language and Some Musical Applications*. Doctoral dissertation, Sibelius Academy, Helsinki, Finland.

LispWorks: CAPI User Guide, http://www.xanalys.com/.

Steele G. L. JR. 1990. *COMMON LISP THE LANGUAGE*. Digital Press, 2nd edition, Massachusetts, USA.

Woo M., J. Neider, T. Davis, and D. Shreiner. 1999. *OpenGL Programming Guide*. Addison Wesley, 3rd edition, Massachusetts, USA.