Recent Trends in PWGL

Mikael Laurson* and Mika Kuuskankare †

* Centre for Music and Technology, Sibelius Academy
laurson@siba.fi
† Department of Doctoral Studies in Musical Performance and Research, Sibelius Academy
mkuuskan@siba.fi

Abstract
This paper presents PWGL that is a free cross-platform visual language based on Common Lisp, CLOS and OpenGL, specialized in computer aided composition and analysis, and sound synthesis. It integrates several programming paradigms (functional, object-oriented, constraint-based) with high-level visual representation of data and it can be used to solve a wide range of musical problems. PWGL provides a direct access to its base languages, Common Lisp and CLOS in a form of a visual box. Besides a library of basic boxes (arithmetic, lisp functions, list handling, loops, abstractions, and conversion), PWGL contains several large-scale applications, such as: Expressive Notation Package (ENP, represented in PWGL by Score-editor and Chord-editor), 2D-Editor, PWGLSynth, and PWGLConstraints.

1 Introduction

Since its first introduction PWGL (Laurson and Kuuskankare 2002) has undergone several revisions. PWGL is based on similar concepts than PatchWork (Laurson 1996; Assayag et al. 1999). However, PWGL has been completely rewritten and redesigned in order to create a modern cross-platform environment for computer assisted composition, analysis and sound synthesis. PWGL is programmed with LispWorks ANSI Common Lisp that is source-code compatible across several different operating systems, such as OS X, Windows, and Linux. It provides support for Foreign Language Interfaces, and OpenGL (the graphics part of PWGL is based on OpenGL). The use of the same code-base aids the development process and distribution to multiple platforms.

Currently, PWGL is one of the two major Lisp-based visual composition environments along with IRCAMs OpenMusic (OM; Assayag et al. 1999). Each of these complex environments have their unique features and thus an extended comparison within this paper is not feasible. Sufficient to say that PWGL has a strong emphasis on visual programming, sophisticated music notation, constraint-based programming, and sound synthesis.

One of the key points in PWGL is the tight integration of its major tools in the system. For example, on the one hand music notation can be used for sound synthesis control, solving constraint-based problems, scripting, and music analysis. On the other hand in all these cases PWGL provides a special pattern-matching rule syntax—originally developed for our constraint-based language—that allows the user to calculate performance control data, solve search problems, manipulate complex score structures, or attach analytical information to a score. Several of these applications have been discussed in our recent papers (Laurson and Kuuskankare 2005; Kuuskankare and Laurson 2005; Laurson, Norilo, and Kuuskankare 2005).

In this paper we aim to give a general introduction to the current state of PWGL with an emphasis on visual programming, user interaction issues, programming tools, documentation, and tutorials. Due to space limitations we will not discuss the recent work dealing with sound synthesis. Also we will not refer in detail to some of the important work that has been done with our music notation package as this part of our research will be discussed in a separate paper.

Currently PWGL runs under Mac OS X 10.4 and Windows XP operating systems. PWGL is distributed in two different configurations: as a standalone application, called PWGL Application, that is targeted mainly to end users, and as a developers version, PWGL Binaries, that requires the LispWorks (www.lispworks.com) programming environment. This version is made available as a pre-compiled module that is loaded on top of LispWorks. The download page can be found in the following home page: www.siba.fi/PWGL. In our web page you can find links to some introductory material and examples; there is also a support section and the latest news about PWGL.

The rest of the paper is organized as follows. We first introduce the main visual components of the system. Then we describe the main principles that allow the user manipulate a visual patch. After this we introduce the provided programming interface that allows to extend the system with user-libraries, boxes and menus. We end with a short survey of the PWGL Tutorial concept, which can be used for demonstrations, pedagogical purposes, or for private study.
2 Visual Programming Components

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. More complex cases are dealt with in the next subsection.

A box is the most important component of the system and typically it has a name and a number of input-boxes 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. When evaluated the patch returns a Lisp value resulting from the evaluation and prints it in an output browser window called ‘PWGL output’ (Figure 1). Often the main interest is not in the textual output but in the side-effects of the calculation within the patch. For instance, the result may be stored in an editor, or the calculation may result in changing the state of an object (a typical example of this kind of approach can be found in Figure 3).

![Figure 1. The PWGL workspace with a patch (top) and an output browser giving the result of a patch evaluation (bottom).](image)

The state of a patch can be saved to file. The resulting file has the extension ‘.pwgl’. PWGL includes a persistent object system that allows to store or duplicate automatically any new CLOS object that is introduced to the environment.

2.1 PWGL Box Overview

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.

PWGL supports automatically the most commonly used Common Lisp lambda-list keywords (i.e., &optional, &rest and &key). Boxes that represent normal Lisp function argument lists extend boxes either with one input-box (&optional and &rest) at a time or with two input-boxes (&key). Custom made extension patterns are also possible, allowing the programmer to organize the input-box distribution of boxes with complex behavior into meaningful groupings, where subgroups can be shown or hidden according to the task in hand.

PWGL has also a novel database oriented box type that is able to control dynamically the extension behavior of input-boxes. Here typically the first input-box is a menu input-box that allows the user to choose an option that will affect the input-box types, default values and extension patterns of succeeding input-boxes. This database oriented approach is of primary importance as it allows to reduce drastically the number of required boxes in complex applications. A representative database box example is for instance a special constructor box that is situated in the kernel part of our system, called ‘enp-constructor’ (Figure 2). This box can be used to build music notation oriented objects in PWGL. The first input-box is a menu that contains a collection of object types used in music notation (e.g. score, part, voice, measure, beat, chord, note). After choosing one option the box automatically adjusts its behavior and appearance according to the current object type. Thus a single box is capable of handling a great variety of different tasks, and provides the user with a clear and intuitive insight to the complex system of musical objects.

![Figure 2. Two applications of the 'enp-constructor' box to create a measure object (to the right) or a beat object (to the left).](image)

2.2 PWGL Input-Box and Editors

PWGL has a library of predefined input-boxes types to represent numbers, lists, sliders, switches, simple menus, hierarchical menus, editable text, static text, and buttons.

PWGL has also an important subgroup of input-boxes that are associated with 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 to be edited by the user.
The Score-Editor allows to present a large number of different musical material types within one editor, such as chord sequences, melodic lines, time notation (timing information is given in seconds), frame notation (used both in mensural and non-mensural context) and traditional western metric notation. Different material types can even be mixed (for instance frame notation with metric notation). This integration simplifies greatly the complex issue of how to process, present and manage musical data within a visual programming language.

A similar approach has been taken with the PWGL 2D-Editor. The 2D-Editor supports a collection of objects that have two dimensions, such as bfps, sound samples, chord-sequences, bezier functions, markers, scrap-objects, and so on. This scheme allows to combine and synchronize visually various 2D-objects within one editor. The 2D-Editor can be used for a wide area of tasks such as sound synthesis control, visualization of mathematical functions, compositional sketches, musical processes, and analysis results.

In contrast to several other existing visual systems PWGL has a strong emphasis on the visual appearance of the input-boxes. All editor input-boxes are able to draw directly the contents of their editor-windows on a patch level (Figure 3). This means that the user can have a precise overview of the patch and the contents of the editors contained in the main window without having to open the respective editor-windows. If necessary, this information can be zoomed and/or hidden either by resizing the box or by using a special minimize-button in the top-left corner of the box. Some editor input-boxes can even be edited directly in the patch.

![Figure 3. A PWGL example displaying several boxes with editor input-boxes.](image)

### 3 User Manipulation of a Patch

PWGL is based strongly on the concept of direct editing. Thus, as PWGL is not tool oriented, the user is typically not forced to go into a specific state in order to perform some operation (such as panning or zooming). The benefits of this kind of a system are obvious: the user is not forced to switch context for some operation to occur; a mouse gesture—for instance a simple click—does not suddenly change its meaning as the state is not changed; the user is not forced to perform operations in a certain order as the manipulations can be performed in a state-free system.

PWGL has a mouse driven user interface and requires a standard 3-button mouse (button-1, scroll wheel, button-3).

When the mouse is moved (i.e. no buttons are pressed) the cursor changes depending on what kind of object is under the current mouse position. The cursor shape gives a hint of what operation is going to occur if the mouse is being pressed.

The left-most button (button-1) is used to select boxes, input-boxes, and connections, or to move or to resize the selected boxes. When the mouse is dragged after an input-box click, this operation can be used to edit locally the current value of the input-box. Double-clicks are used to open an editor or dialog which allows to edit the properties of the current object that has been clicked. Each box and input-box class has a dedicated editor.

PWGL provides a uniform way of handling pan and zoom operations. Pan is accomplished by dragging a mouse while pressing the scroll-wheel button, and zoom is achieved with the scroll-wheel. These operations are typically global, i.e. they affect the complete contents of a patch window or an editor-window. Global pan and zoom settings can be stored in so-called ‘snapshots’ which allow the user to quickly focus on some detail of the patch. PWGL editor input-boxes also support local pan and zoom. This is accomplished by moving the cursor above an editor input-box. The visual appearance can then be locally manipulated either by dragging the scroll-wheel button (for pan) and/or with the scroll-wheel (for zoom).

The right-most button (button-3) is used for context sensitive pop-up menus. A right-click on a patch-window opens a window pop-up menu, a box-click a box pop-up menu, an input-box-click an input pop-up menu, and finally a connection-click opens a connection pop-up menu. These pop-up menus are of central importance to the system as they inform the user what operations are currently possible. The pop-up menus are dynamic, for instance the window pop-up menu shows all abstractions that are found in the current system; a box pop-up menu can check whether the current box has special options, such as adding outputs, or adding or removing inputs; an input-box pop-up menu may suggest to connect a slider-box or a text-box to the current input, etc.
4 Programming Interface

Besides visual patch-level programming PWGL offers several ways to extend the kernel part of the system. In its simplest form the user can add Lisp code to a patch by using one of the available text-based editors. A ‘text-box’ is used for textual data, rules, box definitions, menu definitions and code. The contents of the box is not evaluated automatically. A ‘Lisp-code’ box, in turn, is normally used only for code and the contents is compiled automatically when a patch is loaded.

For more advanced projects the user can use the user-library protocol. The idea behind PWGL user-libraries is similar to the one found already in PatchWork. Typically this kind of programming effort requires the developers version of PWGL, i.e. PWGL Binaries.

The distributed PWGL system provides an example user-library template that can be used as a starting point for future development. This template contains instructions on how to create a load file, how to organize the source code within the user-library project, and how to create a user-library pop-up menu. This example contains also source-code for creating basic PWGL boxes with various input-box types. Information on how to create more complex box examples can be found in the ‘Documentation’ folder.

5 Tutorials

PWGL contains a large introductory package called ‘PWGL Tutorial’. This tutorial can be found under the ‘Help’ menu in the main menu-bar and it aims to demonstrate how PWGL works in practice. The textual information and patch examples are organized as a hierarchical folder structure—shown in the left part column of the tutorial window—that contains at the leaves either text files or patch examples. When a patch file name is selected, the respective patch window is opened in the right part of the tutorial window. The patches are fully functional, i.e. they can be evaluated, played, boxes and connections can be added or deleted, and so on. Saving is disabled. Figure 4 shows a tutorial that combines several key points of PWGL.

Figure 4. A PWGL tutorial example displaying a constraint-based search problem. The result is given in the score in the upper part of the patch.

6 Acknowledgements

The work of Mikael Laurson has been supported by the Academy of Finland (SA 105557).

References


