PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL

Mikael Laurson and Mika Kuuskankare

Center for Music and Technology, Sibelius Academy, P.O.Box 86, 00251 Helsinki, Finland
email: laurson@siba.fi, mkkuusk@siba.fi

Abstract
This paper introduces a new Lisp based visual language called PWGL. PWGL is based on PatchWork (PW) which has been completely rewritten and redesigned in order to create a new and modern programming language. The main difference between PW and PWGL is that the graphics part of the implementation is based now on OpenGL and not on QuickDraw as was the case with PW. OpenGL has many attractive features and offers several advantages when compared to QuickDraw. The aim with this project is also to simplify, clarify and improve many of the useful features and concepts found in PW.

1 Introduction
PatchWork (PW, Laurson 1996) has been used after its first introduction in 1993 by a large community of composers, musicians and researchers. While been based on many powerful concepts (direct relation to its base languages Lisp and CLOS, musical objects, visual editors) PW has suffered from several handicaps both from the user and from the programmers point of view. PW editors, for instance, while being useful tools for manipulating and representing complex musical data, have a non-uniform and often a confusing user-interface. Also PW musical objects have different protocols when creating them. As PW has no persistent object system, each object introduced to the environment must also include code how to save and load the new object. Finally, QuickDraw, the graphical language behind the graphic part of PW, is quickly becoming obsolete. This has become lately even more obvious after Apple has introduced its newest operating system OS X, where QuickDraw is supported only for compatibility reasons.

The research team at IRCAM for computer assisted composition introduced in the late 90’s a new graphical programming language called OpenMusic (OM, Assayag et al. 1999). OM, while being based on similar concepts than PW, includes several important extensions and improvements when compared to PW: persistent object system, visual programming of objects, the “maquette” editor, etc. More information on OM and its recent developments can be found at IRCAM.

Meanwhile, our research team at Sibelius Academy continued to use and extend PW with several new user libraries dealing with constraint-based languages, music notation and sound synthesis: PWConstraints (Laurson and Kuuskankare 2001), Expressive Notation Package (ENP, Kuuskankare and Laurson 2001) and PWSynth (Laurson et al. 2001; Laurson and Kuuskankare 2001). It has lately become obvious, however, that PW had to be rewritten in order to meet our new research targets. This paper gives an overview of our recent experiences when converting the old PW system to a new and modern visual language.

The rest of this paper is divided in four main sections. First we discuss OpenGL (Woo et al. 1999) in general terms and show how it is an appropriate tool for our purposes. Then we go over to some important design issues behind PWGL. In the next section we discuss how editors and constructors are interfaced to the system. The final section gives example patches that show how the system can be used in practice.

2 OpenGL
This section enumerates some of the most important features that motivated us to move from QuickDraw to OpenGL.
2.1 Multi-platform support

OpenGL is currently supported by all main operating systems (Windows, Linux, Macintosh OS9 and OSX).

2.2 Hardware acceleration and efficiency

Almost all recently introduced personal computers are equipped with powerful graphic acceleration cards. This means that currently OpenGL runs extremely fast on a relatively cheap hardware on all major operating systems. Such an efficiency boost improves directly the quality of any application utilizing OpenGL.

2.3 Floating point graphics

Coordinate information in OpenGL can be represented as floating point numbers. By contrast, coordinate data in QuickDraw has been limited to 16-bit integers. The ability to use floating point graphics offered by OpenGL has several practical consequences both from the user and from the programmer point of view. A typical problem in an integer-based graphic system - such as PW - is that data has to be continuously translated from floating point values to integers or vice versa. In an OpenGL-based system these unnecessary operations can be avoided.

2.4 2D and 3D graphics

OpenGL is a very large language and supports sophisticated 2D and 3D graphics. It is obvious that within this short space we can only hint to some ideas of the almost myriad possibilities that OpenGL offers.

OpenGL applications are typically associated with complex animated 3D graphics (games, simulators, animation movies). Thus the kernel part of our project - consisting patches with boxes and connections having mostly a 2D look - may not seem to be a typical example of an OpenGL application. Our system uses nevertheless several OpenGL 3D features. An example of this is the way connections are always drawn behind boxes in a patch. This is achieved simply by assigning a z value for connections that is farther away from the viewer than the z values of boxes.

OpenGL provides anti-aliasing for its line and texture font drawing routines. These features improve the outlook of a patch on the screen and on paper. For instance, all figures given in this article are screenshots without any further processing.

Blending of colors can be used effectively to produce graphics that is lively and subtle. This subtleness is not only pleasant but can be used to emphasize certain factors while keeping other aspects of an image in the background. The net-effect is that we can now produce patches which contain much more information than in the old PW system without having patches that seem extensively crowded.

2.5 User interaction

QuickDraw supports user interaction only in a fairly rudimentary form. Typically the user can select simple graphical objects such as points, rectangles or regions.

OpenGL, by contrast, allows any selectable object to be arbitrary complex. Furthermore, the selection mechanism - or in OpenGL terminology the “pick” utility - is almost automatic and does not require special coding tricks. Any OpenGL drawing code can be dispatched so that it collects objects within a user definable cube - typically centered around the current mouse position - in a pick-buffer. The pick-buffer can then be parsed by the software which decides which objects have been selected by the user.

3 General PWGL design issues

Next we describe some PWGL design principles that we considered to be essential when we started the rewrite of PW.

3.1 Persistent object system

PWGL is based on a persistent object system that allows the user/programmer to add new objects to the system without having to worry how these objects will be saved.

3.2 Compatibility with PW

Whenever possible we tried to be compatible with the old PW system. Thus, for instance, most of the boxes in the kernel part of PW have been imported to PWGL (they are, however, not identical, see the Section 3.4 for more details). The main exceptions are the old PW editors (BPF-editor, chord-editor, chord-sequence-editor and RTM-editor) which have been completely redesigned (see also Section 4). The user can, however, import the contents of a PW RTM-editor or BPF-editor into PWGL.
3.3 PWGL windows

PWGL contains two basic types of windows. The first type is a patch window, typically containing boxes and connections. The second type is a PWGL editor window. Although the patch windows and the various editor windows have very different functionalities they all share some basic ways the user can manipulate the contents of a window. These operations are triggered with the help of modifier keys (in Macintosh terminology: ‘Command’, ‘Control’ and ‘Option’ keys) and can be categorized into two main groups.

The first group consists of navigation tools that allow to zoom (‘Option’) and pan (‘Command’) the contents of the window.

The second group, in turn, deals with context sensitive menus (‘Control’). These menus allow the user to inspect or modify the contents of the window. A typical example is for instance a situation where the user clicks to the content area of a patch window. In this case the system opens the kernel-menu which allows to add a new object into the patch. However, if the click hits an object (box, input-box or a connection), a specialized menu is opened that allows to inspect or modify the selected object.

3.4 Boxes and connections

A PWGL box consists in its simplest form of a main-box and of one or several input boxes (there are however many exceptions: sliders, editor-boxes and special recursive boxes that can contain embedded main-boxes).

The simplest way to add a box into patch window is to type the name of a Lisp function or method in a dialog window. The system supports automatically the most commonly used Common Lisp lambda-list keywords.

If the user wants to control the type of the input-boxes, default values and layout-options in a detailed manner then she/he can use the PWGLDef macro. PWGLDef generates both a generic function and a method in its most general form. The user can add more specific methods by using parameter specializers thus allowing a single box to be used in many contexts.

PWGL boxes can be resized by dragging a resize triangle that is situated in the low-right corner of a box. This allows, besides making a box smaller or larger, to hide the input-boxes of a main-box. Also any input-box of a PWGL box can be connected either from the left or from the right side of a main-box. All boxes can contain a graphical lock that prevents a box to be re-evaluated. Specialized boxes – typically abstractions and editors - can have several outputs. PWGL connections can have various shapes (straight line, 5-point line or bezier function with two control points).

4 Editors and Constructors

Currently PWGL supports two main editors - called Score-editor and 2D-editor - which are both complex applications of their own. Due to space limitations we can give here only some brief remarks of how these editors are incorporated into the system.

The Score-editor utilizes internally the ENP2.0 music notation package. ENP2.0 is based on the former ENP user library of PW. The Score-editor combines all PW music notation related editors (i.e. chord-editor, chord-sequence-editor and RTM-editor) into one package. ENP2.0 supports both mensural and non-mensural notation. Also ENP2.0 has been completely rewritten and is now based on OpenGL.

The 2D-editor is a general-purpose editor which can contain objects that have data in 2 dimensions. A typical example of this kind of an object is a break-point function object (thus the 2D-editor replaces among other things the old PW BPF-editor). Other types of 2D-objects that are supported are bezier-functions, chord-sequences and sound samples.

Besides these main editors PWGL contains also simpler editors. For instance the Matrix-editor allows to edit visually numerical data in a matrix form.

All editors are associated with special constructor boxes which are used to create instances of the required objects for PWGL editors.

Constructor boxes accept either instances – typically coming from constructor boxes – or data-lists which allow to partially modify the objects situated inside the editors. The latter case allows, for instance, to change only the pitches of a score without changing the rhythmic structure.

5 Example patches

In this concluding section we give three patch examples that demonstrate some of the features of PWGL discussed in the previous sections.

Figure 1 shows a patch containing a 2D-editor. The patch generates break-point-functions by interpolating two sine waves. This example shows different connection drawing modes, a box (‘bpf-constructor’) with hidden inputs
and left/right connections and a 2D-editor box with three outputs.

![Diagram of a 2D-editor box with 3D connections](image1)

**Figure 1.** The foreground window shows a 2D-editor containing 60 break-point functions. The patch that generated the contents of the 2D-editor is shown in the background.

The next example (Figure 2) shows a synthesis patch where a Score-editor box is generating control information for a guitar synthesizer. The patch has also a 2D-editor box which contains a sound sample:

![Diagram of a guitar model synthesis patch](image2)

**Figure 2.** A guitar model synthesis patch containing a Score-editor, two Matrix-editors, a synth box and a 2D-editor.

Figure 3 gives our final example where a guitar string model is controlled in real-time with the help of a special guitar module (this is an example of a complex imbedded PWGL box). Note also the complex ‘synth’ box which consists of a main-box, a slider for master volume control and a real-time signal display. This patch allows the user to trigger with the mouse the current string, fret and pluck position.

![Diagram of a synthesis patch containing an interactive guitar module](image3)

**Figure 3.** A synthesis patch containing an interactive guitar module.

### 6 Acknowledgments

This research has been conducted within the project “Sounding Score—Modelling of Musical Instruments, Virtual Musical Instruments and their Control” financed by the Academy of Finland.

### References


