Mikael Laurson, Mika Kuuskankare, and Vesa Norilo
Centre for Music and Technology (CMT)
Sibelius Academy
P.O. Box 86
00251 Helsinki
Finland
{laurson, mkuuskan, vnorilo}@siba.fi

An Overview of PWGL, a Visual Programming Environment for Music

This article provides an overview of our free Lisp-based, cross-platform, visual programming environment called PWGL (the name is an acronym for PatchWork Graphical Language). PWGL can be used for computer-aided composition, music analysis, and sound synthesis. Our work is influenced by our past experience with PatchWork (Laurson 1996; Assayag et al. 1999). However, PWGL has been completely rewritten and redesigned in our attempt to develop and improve many of the useful concepts behind PatchWork.

PWGL and several of its applications have already been discussed in our previous work, including Laurson and Kuuskankare (2002, 2006), Kuuskankare and Laurson (2006), and Laurson, Norilo, and Kuuskankare (2005). Since 2002, PWGL has undergone several revisions and additions including the public release, launching of the PWGL home page, inclusion of the tutorials and new programming interfaces, etc.; each of these have been attempts to make our system more available and more easily approachable within the computer music community. The main purpose of the present article is to give a general overview of the current state of the system with an emphasis on features that are useful for potential PWGL users, such as composers, music analysts, and researchers.

PWGL is programmed ANSI Common Lisp (Steele 1990) from LispWorks (www.lispworks.com), which is source-code compatible across several different operating systems, such as Macintosh OS X, Microsoft Windows, and Linux. LispWorks provides support for Foreign Language Interfaces, by which a program written in one programming language can call routines or make use of services written in another. LispWorks also supports the OpenGL graphics library API. (The graphics part of PWGL is based on OpenGL.) The use of the same codebase aids the development process and in distribution to multiple platforms.

PWGL integrates several programming paradigms (functional, object-oriented, and constraint-based) with high-level visual representation of data, and it can be used to solve a wide range of musical problems. A PWGL patch is the main workspace where the user can add boxes and create relations among them using connections. In its simplest form, a PWGL patch can be seen as a visualization of Lisp: a patch consists of boxes that represent Lisp functions or Common Lisp Object System (CLOS) methods. As PWGL provides a direct interface to its base languages, every Lisp function or CLOS method can be automatically transformed into a box. As a result, PWGL offers the flexibility of both a traditional text-based programming language and a visual programming language. Besides a library of basic boxes (e.g., arithmetic, Lisp functions, list handling, loops, abstractions, and conversion), PWGL contains several large-scale applications, such as the Expressive Notation Package or ENP (Kuuskankare and Laurson 2006), 2D-Editor (Laurson and Kuuskankare 2004), PWGLSynth (Laurson, Norilo, and Kuuskankare 2005), and PWGLConstraints (Laurson and Kuuskankare 2005).

Currently, PWGL runs under Macintosh OS X 10.4 and 10.5 (Universal Binaries) and Windows XP operating systems. PWGL is distributed in two different configurations: as a standalone application—PWGL Application—that is targeted mainly to end-users, and as a developers version—PWGL Binaries—that requires the LispWorks programming environment. The latter version is made available as a pre-compiled module that is loaded on top of LispWorks. The downloads are made available through our projects home page at www.siba.fi/PWGL. Our Web page also includes links to some introductory material, examples, news, and a comprehensive support section.

Computer Music Journal, 33:1, pp. 19–31, Spring 2009

The most important key features of PWGL are its high-quality visual appearance; its ergonomic, uniform, and efficient graphical user interface; the ability to operate visually and directly with high-level musical data; powerful intermixing of textual and visual programming; tight integration of its major tools (i.e., music notation, sound synthesis, scripting, and constraint-based programming); and cross-platform code-base.

The rest of this article is organized as follows. We start by comparing PWGL to two other Lisp-based composition environments. Then we compare text-based and visual programming in general, and we describe how PWGL is positioned within these categories. After this, we enumerate the most important design issues that have guided our development work. Following these introductory sections, we present the main visual components of the system and describe the primary principles that allow users to manipulate a visual patch. Next, we introduce the programming interface that allows one to extend the system with user libraries, boxes, and menus. This section includes a novel programming interface that can be used to import new synthesis boxes into our synthesis environment. We give also a short survey of the PWGL Tutorial concept, which can be used for demonstrations, pedagogical purposes, or for private study. We conclude with an extended PWGL example.

Related Work

Currently, PWGL is one of the three major Lisp-based composition environments, along with IRCAM's OpenMusic (OM; Assayag et al. 1999) and Heinrich Taube's Common Music (CM; Taube 1991). Of these systems, CM differs most from the other environments as it relies on text-based programming and various third-party applications for data visualization, sound synthesis, and music notation. PWGL and OM, by contrast, rely heavily on visual programming and contain many aspects (boxes, music notation, visual editors, etc.) that are integrated to the kernel part of the system.

Like PWGL, OM is a successor of PatchWork and thus has several points in common with our system. OM includes several important extensions and improvements compared to PatchWork, including its persistent object system, the visual programming of objects, the "maquette" editor, etc. Since its introduction during the late 1990s, OM has gained a large user base (Agon, Assayag, and Bresson 2006).

PWGL and OM are complex environments with unique features, and thus an extended comparison within this article is not feasible. We will simply note here some of the most important differences between the two systems. First, despite their obvious similarities inherited from the PatchWork tradition (both use patches with boxes and connections), OM and PWGL take somewhat different approaches to visual programming. On the one hand, OM has a tendency to emphasize the visual programming aspect more than PWGL; for example, it provides tools where CLOS classes can be created using dialogs, where class properties such as inheritance, slots, and methods can be edited visually. PWGL, in turn, is more text-oriented, where this kind of programming is performed textually. On the other hand, PWGL has a more sophisticated visual environment than OM. For instance, because our graphics are based on OpenGL, all visual entities (windows, boxes, input boxes, and music notation) are fully scalable. (OpenGL will be discussed in more depth later in this article.)

Second, PWGL contains a dedicated synthesis environment that allows one to visually create instrument definitions, musical scores, and mapping of control information within one unified framework. OM, by contrast, does not contain a dedicated software synthesizer, and so control information must be exported to external synthesis environments (Agon, Stroppa, and Assayag 2000). Finally, our music-notation environment, ENP, when compared to the somewhat rudimentary music-notation facilities of OM, offers a rich notation tool in which scores can be produced either manually or algorithmically. ENP has a mouse-driven graphical user interface, a large set of musical primitives, an extended concept of expression markings, and it allows an algorithmic control over scores.

Text-Based Programming and Visual Languages

According to a definition given in Myers (1986, p. 60), "Visual Programming (VP) refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Conventional textual languages are not considered two dimensional since the compiler or interpreter processes it as a long, one-dimensional stream." When discussing the benefits of visual programming languages, Myers gives the following argument on why visual languages are considered appealing: "The human visual system and human visual information processing is clearly optimized for multi-dimensional data" (p. 60).

According to Ambler and Burnett (1989), visual languages can be divided into two categories. *Visually transformed languages* are inherently non-visual but possess superimposed visual representations. *Naturally visual languages*, by contrast, attempt to develop entirely new programming paradigms. The inherent natural expression of these languages is visual and may not have strictly textual equivalents.

As our system is built on top of Common Lisp, a text-based language, PWGL is closest to a visually transformed language, and we will use the term "visual language" in this restricted sense. Furthermore, it is important to note that PWGL is not a purely visual language, as our system also depends on text-based programming.

An important aspect of the text-based programming languages is that there are many universally known standards among them—C, Java, Common Lisp, etc. Also, in cases such as complex control structures like loops and recursion, they are usually more compact than visual languages.

Among the strengths of visual languages, in turn, is the fact that the user can build the programming logic graphically by selecting different modules (usually boxes) and by making connections among these. The result resembles a flow chart that is often easier to read, modify, and understand than text-based programs. Also, visual languages are simpler to master as they usually have a more uniform and intuitive syntax than text-based languages. Essentially, the syntax consists of making the "right" connections among the "right" boxes. Furthermore, many visual languages provide boxes that are functional already before they are connected to other boxes: the inputs contain meaningful default values.

A music language can be based on any programming approach—text-based, visual programming, or a combination of both. In our opinion, incorporating a visual front-end to a music language makes it more attractive and flexible. Musical objects can be recognized and modified efficiently, and program structures can be parsed, understood, and manipulated more easily. When displaying musical material visually onscreen, we can ideally combine the benefits of traditional score representations with the novel and dynamic possibilities of the computer. However, if the system becomes overly oriented toward visual programming, then having a visual front-end can become counterproductive, resulting in patches that are crowded and confusing. In these cases, the textual approach is often more economic and efficient. Thus, the user should ideally be able to switch to whatever programming technique is appropriate to a given problem. As PWGL is closely related to its base Lisp language, mixing textual and visual programming is straightforward.

General PWGL Design Issues

There are several important cross-platform technologies behind PWGL, such as Automatic Source-Code Generation (ASG) and OpenGL Vector Graphics. We briefly discuss these technologies next, and then we end this section by enumerating the key concepts behind the graphical user interface of PWGL.

ASG is an integral part of PWGL providing simple persistent object storage. ASG can be used to transform virtually any Lisp object into an ASCII representation. This, in turn, can be used to produce copies of the original, or it can be saved to a file. There are several free and commercial Persistent Object Storage systems, such as William's Object Oriented Database (WOOD) for Macintosh Common Lisp (MCL), PCLOS (Paepcke 1990), AllegroStore for Allegro Common Lisp (www.franz.com), Statice

for Symbolics Lisp Machines, etc. These systems are usually implemented using CLOS's metaobject protocol (Kiczales, Rivières, and Bobrow 1991) or some kind of database system, such as Berkeley DB (Olson and Seltzer 1999). Our CLOS-based Persistent Object Storage system was designed specifically for the needs of PWGL. ASG enables programmers (including the original developers) and users to transparently design new objects without having to write complicated and error-prone code to make them reproducible.

The graphics part of PWGL is programmed using OpenGL (Woo et al. 1999), a cross-platform application programming interface (API) for developing portable, interactive 2D and 3D graphics applications. It features double-buffering, anti-aliasing, and hardware acceleration, for example, and is an inexpensive way of creating state-of-the-art graphical output and user interaction. One of its most important features is its ability to interact with arbitrary complex graphical objects. This mechanism—called "picking"—is a powerful tool and it is used extensively in PWGL's graphical user interface. Gradient colors, transparency, and textures can be used to create not only appealing visual appearance but also meaningful and functional user interface. This allows, for example, users to emphasize certain factors of the workspace while keeping other aspects in the background.

PWGL has an object-oriented graphical user interface (GUI) that is based on direct manipulation. According to Cooper (1995), direct manipulation can be broken down into three basic principles: (1) visual representation of the objects, (2) the use of dragging instead of text entry or menu selection, and (3) feedback on the impact of the manipulation. Object-oriented user interface design, in turn, is a well known interface paradigm that has several advantages, including (1) familiarity of the conceptual model, (2) seeing and pointing versus remembering and typing, (3) WYSIWYG ("what you see is what you get"), (4) universal and consistent commands, (5) simplicity, and (6) modeless interaction (Smith et al. 1982). The underlying idea behind PWGL GUI is to allow the user to manipulate the information contained in the patch and its various graphical editors as straightforwardly as possible. As a general guideline, any object of any complexity can be edited directly, and all editing operations provide synchronized visual feedback for the user.

According to Nielsen (1993), there are five main attributes of usability: learnability, efficiency, memorability, error handling, and user satisfaction. For example, in PWGL, learnability and memorability are aided by the facts that there is only a small number of mouse shortcuts and that these shortcuts behave similarly across the application. Efficiency, on the other hand, is ensured by allowing the user to manipulate data directly in place. As PWGL is a programming environment, and we have chosen to favor a more open approach (in contrast to a closed and restricted API), errors are generally left to the underlying Lisp system for handling.

In most cases, the objects themselves act as handles for maneuvering the data, but in certain applications, PWGL uses the concept of additional handles (Nielsen 1993). PWGL also uses cursor hinting (Nielsen 1993). This is important, first, for showing the user which object can be directly manipulated, and second, for giving hints as to the type of the operation that can be performed. For example, when resizing a box, a horizontal arrow indicates that the size parameter is horizontally constrained, and a vertical arrow, in turn, indicates the size parameter is vertically constrained.

Visual Programming Components

A PWGL box is the most important component of the system. It is usually represented by a rectangle showing the name of the box, an arbitrary number of input boxes, and one or more outputs. When evaluated, a box reads its inputs, calls a function or method associated with it, and finally returns a result. Connections are used to define relations between boxes. The output of a box can be connected to the input of another box. Thus, the system works in a manner similar to Lisp, in which function calls can have either constants or functions calls as arguments. When evaluated, a box returns a Lisp value resulting from the evaluation and prints it in an output browser. (Normally, evaluation is done by

selecting a box and typing "v" from the keyboard.) PWGL—like Lisp—is an untyped language. Usually, a patch contains a "master box" that is evaluated to calculate the final outcome of the patch. The user can, however, evaluate practically any box in a patch to inspect intermediate results. Often, the main interest is not in the textual output but in the side effects of the chained evaluation of the boxes in the patch. For instance, the final and intermediate results can be visualized using editor boxes, or the calculation can result in changing the state of an object.

The PWGL evaluation scheme can be called request-driven. The user requests the Lisp value of any box in a patch to be calculated. The box receiving the request applies its Lisp function by evaluating its inputs. If an input is connected to another box, a new request is sent to the connected box to apply its Lisp function by evaluating its inputs, and so on. Thus, the requests move upward. By contrast, the information and data in visual systems such as Max/MSP (Puckette 1988; Zicarelli 1998) and Pd (Puckette 1996) is passed downward. A scheme like the latter is sometimes called data-driven—a patch is triggered either by incoming data from MIDI equipment or internally by a so-called `bang` message. (PWGL also supports a triggering scheme, as shown later in this article in Figure 4).

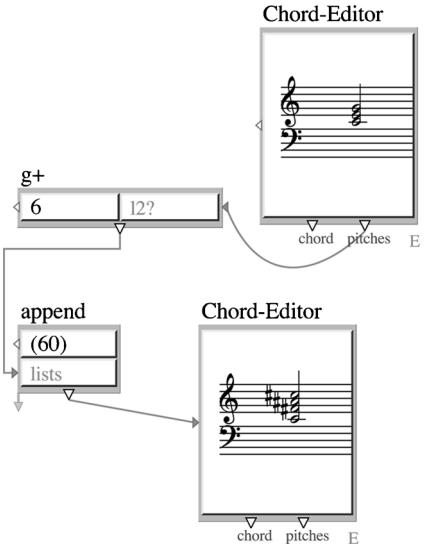
Figure 1 demonstrates the evaluation scheme in PWGL and how it relates to the following Lisp expression:

```
(append '(60) (g+ 6 '(60 64 67)))
```

The main difference between these representations (textual and graphical) is that in the graphical patch, we use two Chord-Editor boxes to visualize the input and the output of the evaluation process.

In the Lisp expression, we read the expression from left to right. We first encounter a Lisp function called `append` that takes two arguments: the first one is the one-element list `(60)`, and the second argument is a new function call that must be evaluated before `append` can return a result. The function `g+`, in turn, takes two arguments: `6` and the list `(60 64 67)`. It then adds the first argument to each element in the list and returns the list `(66 70 73)`. After `g+` has returned its result, `append` can return the final result: `(60 66 70 73)`.

Figure 1. This PWGL patch corresponds to the Lisp expression `(append '(60) (g+ 6 '(60 64 67)))`. *In both cases, a C-major chord is transposed upward by a tritone, and a middle C is appended to the result.*

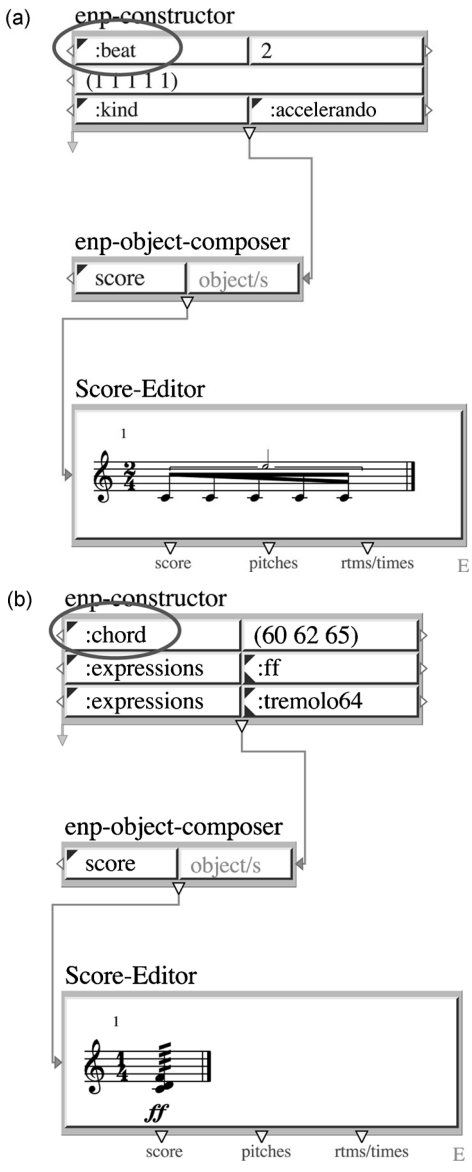


The visual patch works in a similar way. We first evaluate the Chord-Editor that is found in bottom part of the patch. (At this stage, no notes are shown yet.) Next, we follow the connection upward and encounter a box called `append`. The first input is the list `(60)`, and second input is connected to a `g+` box. In the latter box, the first input is `6`, and the second input is connected to a Chord-Editor that returns the C-major chord as a list of MIDI values `(60 64 67)`. Now, the `g+` box can return the result `(66 70 73)`, and then the `append` box can calculate the final result `(60 66 70 73)` that is shown in the bottom Chord-Editor box.

PWGL Box Overview

PWGL offers several ways to construct boxes, ranging from completely automatic to methods that allow the user to specify the exact type of input boxes, default values, and layout options. PWGL automatically supports the most commonly used Common Lisp lambda-list keywords (i.e., `&optional`, `&rest`, and `&key`). Boxes extend either with one input box (`&optional` and `&rest`) or with two input boxes (`&key`) at a time. Custom-made

Figure 2. Two applications of the enp-constructor box to create (a) a beat object and (b) a chord object. The results are seen in the Score-Editor boxes in the lower part of the figure. In (a), the enp-constructor box has three parameters: the beat-unit 2, the rhythm list `(1 1 1 1 1)`*, and the input-pair* `:kind` *and* `:accelerando`*. Other unspecified information, such as pitch, is generated automatically by the system based on default values. In (b), the first parameter consists of MIDI pitch information given as a Lisp list* `(60 62 65)`*. The other parameters are given pair-wise and deal here with two expressions (*`ff` *and* `tremolo64`*).*



extension patterns are also possible, allowing the programmer to organize the input-box distribution into meaningful groupings, where subgroups can be shown or hidden according to the task at hand (Laurson and Kuuskankare 2003). PWGL supports abstraction boxes that allow the user to hide patches inside sub-windows.

PWGL also has a novel database-oriented box type that is able to dynamically control the extension behavior of input boxes. Here, typically the first input box is a menu that allows the user to choose an option that will affect the input-box types, default values, and extension patterns of successive input boxes. This database-oriented approach is of primary importance, as it allows us to drastically reduce the number of boxes in our system. One representative database-box example is `enp-constructor`, a special constructor-box that uses the Expressive Notation Package (see Figure 2). This box can be used to build music-notation 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, etc.); see the circled inputs in Figure 2. 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 this provides the user with a clear and intuitive insight to the available musical objects and their attributes.

PWGL Input Boxes and Editors

PWGL includes a library of predefined input-box types to represent numbers, lists, sliders, switches, simple menus, hierarchical menus, editable text, static text, and buttons. It 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, Beziér functions, and sound samples. The associated editor can be opened in a separate window, where the data can be viewed and manipulated in more detail. All editors support direct manipulation, which allows the user to edit the contents with synchronized visual feedback. Two of the most central editors in PWGL are the Score-Editor (shown in Figure 2) and the 2D-Editor.

The Score-Editor allows us to represent an array of musical material types, such as chord sequences, melodic lines, time notation (timing information is given in seconds), frame notation, and traditional Western metric notation—all within one editor. Different material types can even be mixed. That all of these can be integrated in one editor greatly

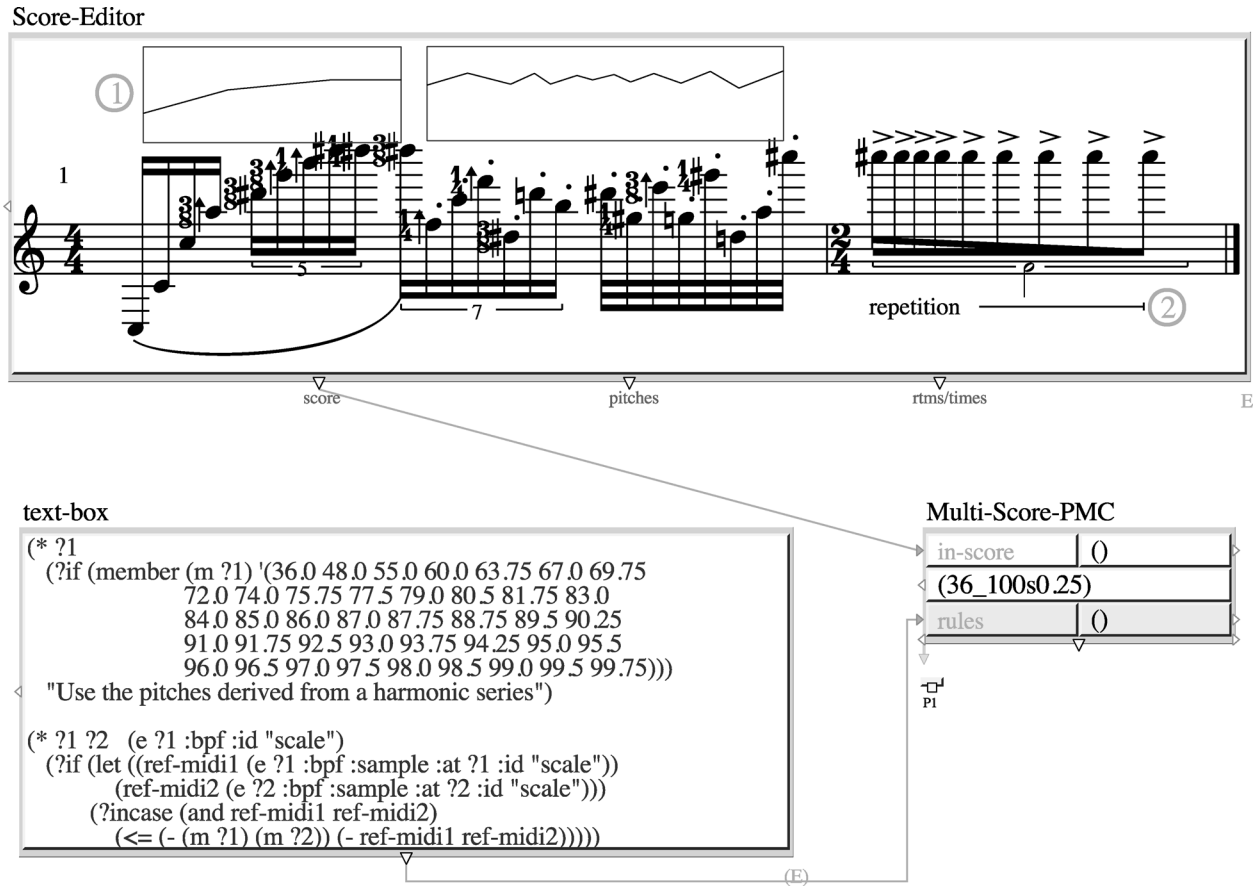


Figure 3. This example contains a Score-Editor box that shows the contents of the internal score in full detail. Besides note information, one can see in the Score-Editor different expression types containing break-point functions and textual markings. These are used to solve a constraint-based problem. The user gives a predefined rhythmic skeleton in the Score-Editor box as an input to our constraint-solver Multi-Score-PMC box. The task is to calculate the pitch content of the input score according to five textual rules that are given in the text-box. (The figure shows only a part of them.) Two of the rules access the break-point functions that are displayed in the score above the staff lines. These shapes control the overall melodic contour of the first measure. A third rule uses the expression `repetition` *to produce a repetition gesture in the second measure. The two remaining rules guarantee that all pitches in the first measure belong to a given overtone scale and disallow nearby note repetitions.*

simplifies the complex issue of how to process, present, and manage musical data within a visual programming language.

The 2D-Editor involves a similar approach. It supports a collection of objects that have two dimensions, such as break-point functions, sound samples, chord sequences, Beziér functions, markers, "scrap-objects," and so on. It also allows users to combine and visually synchronize these objects. The 2D-Editor can be used for a wide range 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 strongly emphasizes the visual appearance of the input boxes. All editor input boxes are able to directly draw the contents of their editor windows on a patch level (see Figure 3). This

means that the user can have a precise overview of the patch and the contents of the editors without having to open the respective editor windows. If necessary, this information can be zoomed or hidden by resizing the box.

User Manipulation of a Patch

As PWGL is based on the concept of direct manipulation, the user is typically not forced to go into a specific state to perform some operation (such as panning or zooming). PWGL has a mouse-driven user interface and requires a three-button mouse with scroll wheel.

When the mouse is moved with no buttons pressed, the cursor changes depending on what kind of object is under the mouse pointer (i.e., cursor hinting). The first button is used to select boxes, input boxes, and connections, or to move or resize the selected boxes. When the mouse is dragged after an input-box click, the resulting operation depends on the type of the object. For example, when an input-box containing a number is dragged, the current value can be either increased or decreased. Double-clicks are used to open an editor or dialog that, in turn, can be used to edit the properties of the associated object.

PWGL provides a uniform way of handling pan and zoom operations. Pan is accomplished by dragging the mouse while pressing the second button, and zoom is achieved with the scroll wheel. These operations are typically global, that is, 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 aspect of the patch. PWGL editor input boxes also support local pan and zoom, which can be accomplished with identical mouse gestures.

The third button is used for context-sensitive pop-up menus. All objects such as windows, boxes, input boxes, and connections have dedicated pop-up menus. These pop-up menus are of central importance to the system, as they inform the user what operations are currently possible. They 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; and an input-box pop-up menu might suggest that the user connect a slider box or a text box to the current input.

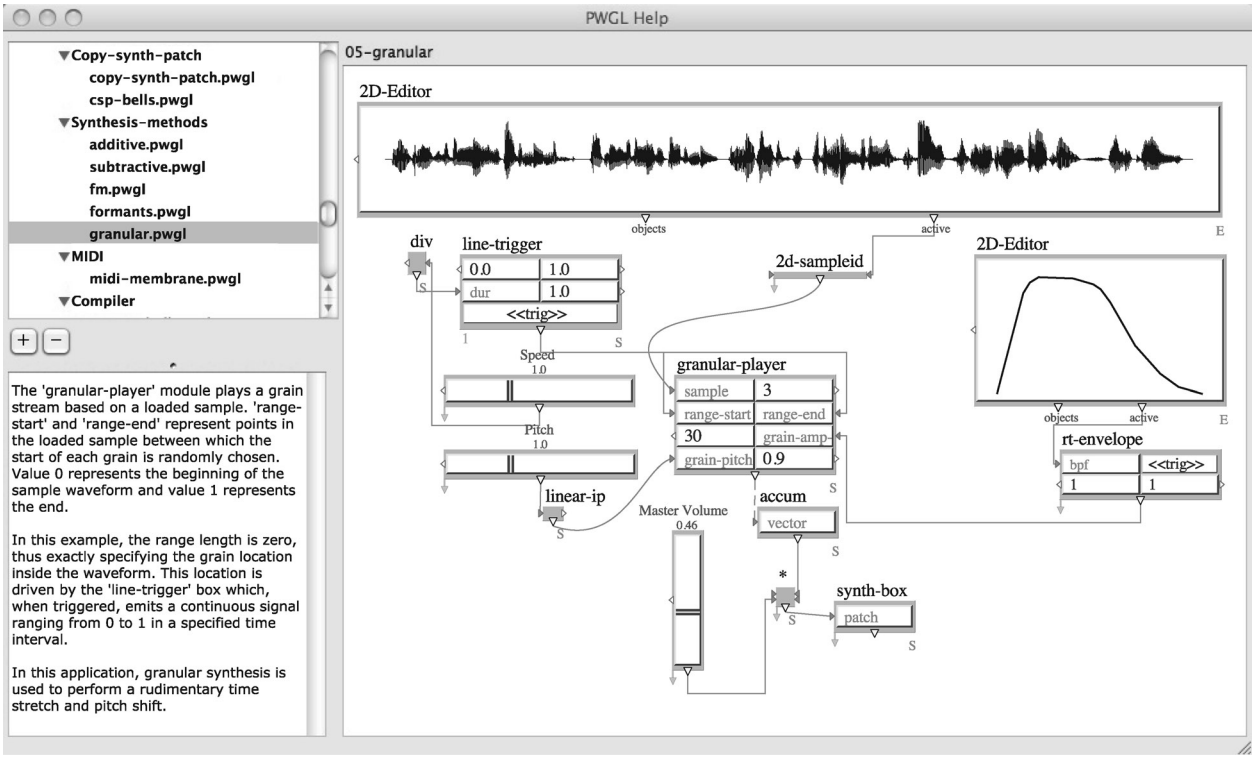
Programming Interface

Besides visual patch-level programming, PWGL offers several ways to extend the kernel 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 can be used for textual data, rules, box definitions, menu definitions, and code. The content of the box is not evaluated automatically. A Lisp-code box, in turn, is normally used only for code, and the content is compiled automatically when a patch is loaded.

For more advanced projects, the user can employ 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 developer's version of PWGL, i.e., PWGL Binaries. PWGL 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 included Documentation folder.

The synthesis part of PWGL, PWGLSynth, can be extended with user boxes in two different ways: using a development package that allows the user to make boxes in C++, or creating new synthesis binaries visually via the PWGL abstraction mechanism (Norilo and Laurson 2007). The signal-processing part of PWGLSynth consists of a number of processing elements created in C++ for greater computational efficiency. Visually, these elements are connected to special synthesis boxes that look like regular PWGL boxes except with a small `S` designation under the lower-right corner of the box.

Figure 4. A PWGL tutorial that demonstrates the use of PWGLSynth. DSP-boxes (boxes marked with an "S") are used in conjunction with two 2D-Editors. The upper one contains a sound sample used by a granular synthesizer. The lower one, in turn, defines an envelope for the grains. Three real-time sliders are used to control the speed, pitch, and master volume parameters. The patch can be triggered either from the keyboard or by clicking the button labeled `<<trig>>` *in the line-trigger box.*



In the development package, all system-level groundwork is provided by a base class called `pwsBox`. This leaves only the input/output configuration, signal-processing, and event responses to be defined by the user. These behaviors can be specified by the means of standard C++ class inheritance and specialization. We offer a small C++ project template that consists of a minimal box example that can be compiled as a Windows DLL or Macintosh OS X bundle that are in turn recognized by the PWGLSynth extension loader. This allows for distribution of individual boxes in binary format, reminiscent of the plug-in concept popular in many current software packages.

In addition to writing C++ code manually, users can take advantage of a PWGLSynth facility for visually defining binary extensions in a PWGL patch. In this case, the system automatically generates and compiles C++ code and creates the appropriate PWGLSynth binaries extension depending on the platform.

Tutorials

PWGL contains a large help facility called PWGL Help. This package consists of an introductory text and a tutorial, and it is found under the Help menu in the application menu bar. The tutorial demonstrates how PWGL works in practice (see Figure 4). The documents are arranged by topic, and the contents are displayed in a hierarchical manner in the navigation panel on the left. A topic can contain subtopics and/or different kinds of documents. When a document name is selected, the corresponding document is opened and displayed on the right. In their simplest form, these documents contain plain text. The more complex documents, in turn, are patches that allow the user to experiment with various features of PWGL. There is also an optional patch documentation that can be displayed either as a drawer on the right side of the window or below the navigation panel. The patches are fully functional—they can be evaluated

and played, boxes and connections can be added or deleted, and so on. Saving is disabled. Currently, the tutorial consists of about 100 documents dealing with general issues, control structures, loops, editors, scripting, constraint-based programming, and synthesis. The PWGL Help package can also be compiled automatically into a PDF document that allows users to study the documentation offline either in an electronic form or as a paper copy. Figure 4 shows a tutorial document of our real-time synthesizer along with 2D-Editors, special DSP boxes, and sliders.

PWGL Example

In this final section, we provide a large-scale PWGL demonstration that illustrates some of the potential of our system in a more demanding context. Other large-scale PWGL applications have already been discussed, for example, in Laurson, Norilo, and Kuuskankare (2005). This example is inspired by the ideas and work of Thomas Hummel, who has developed a multimedia library of the contemporary orchestra, called Virtual Orchestra (VO), at the Experimentalstudio der Heinrich-Strobel-Stiftung. VO includes sample banks with more than 15,000 instrumental sounds, pictures, and graphical materials as well as a detailed documentation of all sounds. VO is commercial software, unconnected with PWGL, and it is available from Forum IRCAM (www.forumnet.ircam.fr). The system has been used, for example, to simulate the human voice with an orchestra (Hummel 2005).

What is unique in this system—when compared to other available orchestral databases—is its bias toward modern playing techniques. Furthermore, all sounds have been systematically analyzed for acoustic properties like dynamic level, spectrum, microtonality, etc. Finally, VO allows free access to all its data consisting of AIFF sound files and ASCII data files. It contains three applications (ISIS, ISIDOR, and OSIRIS) that offer tools for various work situations. These applications can be used for database access, for orchestral chord simulation, and as a sampler.

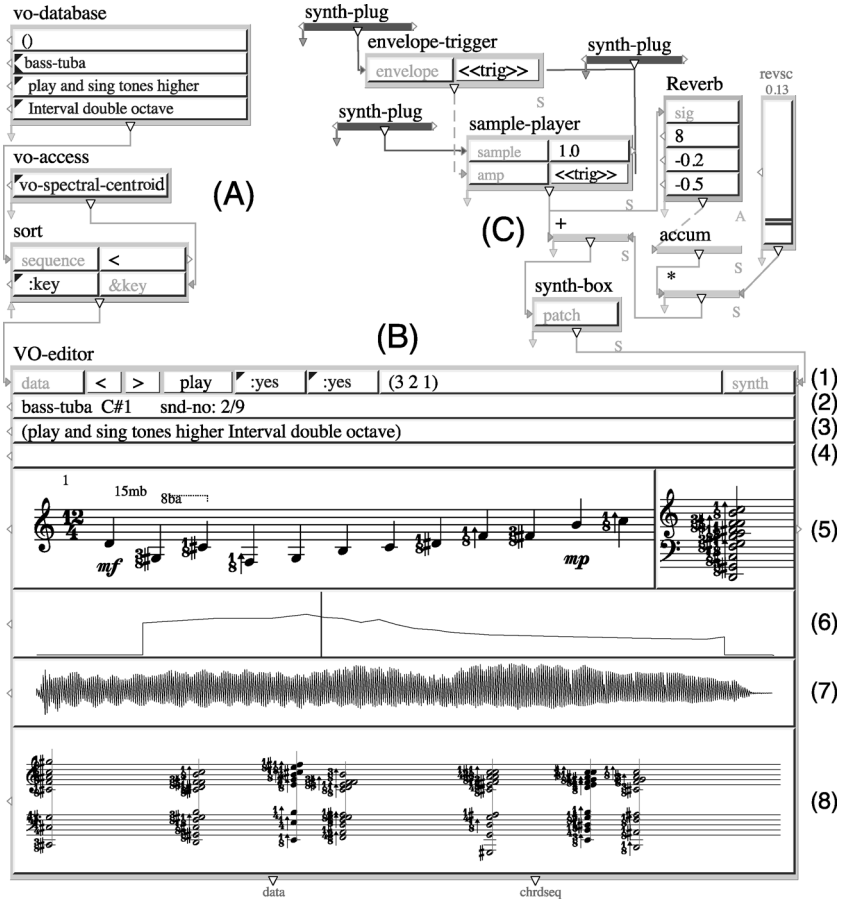
Our implementation has similar functions but also some unique features. We here demonstrate how we can combine features that are normally associated with an application (e.g., high-level data representation, hidden programming details, high-quality user interface, etc.) with the openness and flexibility of our visual programming paradigm. We will use our box scheme so that different properties of our example—such as database access, searching and/or sorting of database items, representation of results, sound synthesis, etc.—can be kept distinct. These entities can then be manipulated, chained, and duplicated in many different ways in a fashion that is typically not possible in a closed application.

We divide our project into three main parts. The first part consists of accessing data from the database according to search criteria. This is accomplished with the help of a box, called `vo-database` (see Figure 5A). The `vo-database` box is an extendable box with two required inputs, `pitches` and `instrument`. Thus, users can first constrain the search according to a pitch list (for an input of `()`, then all pitches are valid) and to the current instrument (this input is a hierarchical menu that contains all VO instruments).

By adding stepwise more menu inputs, users can further constrain the search. When menu inputs are added to the `vo-database` box, it responds intelligently, because it can determine which options are possible in the VO database hierarchy in a given situation. Thus, for instance, if the pitch input is `()` and the instrument input `bass-tuba`, then the next menu input has 23 options, such as "flutter tongue," "double tongue," "glissando," etc. If the user chooses from this list the option "play and sing tones higher," then the next input will contain two options (see Figure 5, where this last choice is "Interval double octave"). If the `vo-database` box is now evaluated, it returns nine database items that all share the criteria given by the `vo-database` box. These items contain, in addition to the sample pathnames, analytical information.

Next, we use in our example a `sort` box to sort the result according to the spectral centroid in ascending order. This part of the process can of course be replaced by any Lisp function. This result is in turn given as a data input to a box called

Figure 5. A virtual orchestra patch example. In (A), the patch is used to access data from the VO database. This result is then sorted according to the spectral centroid. In (B), the patch shows a VO-Editor box, where the first row of inputs, buttons and menus deal with general aspects of the box. Rows 2–7 display the current database item using text-boxes, music-notation editors, and 2D-Editors. The last row gives a chord sequence that is based on the overtone analysis information of the result. In (C), the synth input is connected to a synthesis patch that is used to play the sample information contained in the VO-Editor.



VO-Editor, which is the second main part, (B), in our example. The VO-Editor is a complex box consisting of several input-box types (e.g., two Score-Editors, one Chord-Editor, and two 2D-Editors), and it acts as a placeholder of VO database items. It has many properties that are organized in eight rows of input boxes according to their functionality (see Figure 5). The VO-Editor can be used to represent and manipulate its internal data—i.e., select the current database item, play all samples contained in the data list as a sequence, and select the current synthesis patch, row (1); present the current item's textual information, rows (2), (3), and (4); display the current overtone analysis information as a note sequence and as a chord using music-notation editors, row (5); display the current spectral information and the spectral centroid in a 2D-Editor, row (6); and display the current sound sample, row (7). Finally, row (8) gives the overtone analysis of all data items as a chord sequence. All music-notation editors and the sound sample can be played by selecting the respective input and pressing the "space" key. The music-notation editors of the VO-Editor box have a default pitch resolution of one-eighth tone (i.e., 25 cents).

Finally, part three, (C), of our example deals with synthesis. The last input of the first row, called `synth`, is connected to a patch that will be used when samples are played by the VO-Editor box. This scheme allows the user to dynamically edit any aspect of the synthesis part. We use here a basic patch that is able to play sound samples. The patch contains also a reverberation unit at the output. The synth-plug boxes generate symbolic control

entry points that allow the VO-Editor box to be interfaced with the synthesis part of our system. For more details, see Laurson, Norilo, and Kuuskankare (2005).

Conclusions

This article presented a summary of our research and development efforts during 2002–2008. PWGL was originally written using Macintosh Common Lisp, but it has more recently been written using LispWorks and is now available as free software for Macintosh OS X and Microsoft Windows XP. (A Linux version will be available in 2009.) PWGL is based on many cross-platform technologies, and our decision to move PWGL development to LispWorks has proven fruitful. LispWorks offers several important features in terms of our application and its development, including OpenGL bindings for vector graphics; a portable GUI toolkit for user interaction; a foreign-language interface for addressing low-level languages; robust cross-platform support; source-code compatibility across all supported platforms; royalty-free runtime distribution; and a comprehensive development environment with Graphical IDE, native CLOS/MOP, and an incremental native compiler and interpreter. In addition to a Linux port, our plans include gathering user experiences and developing libraries that allow PWGL to be interfaced with other systems and protocols, such as Open Sound Control (OSC; Wright and Freed 1997) and Sound Description Interchange Format (SDIF; Schwarz and Wright 2000).

Acknowledgments

This work was supported by the Academy of Finland (SA 105557 and SA 114116).

References

Agon, C., G. Assayag, and J. Bresson, eds. 2006. *The OM Composer's Book.1*. Paris: Editions Delatour France/IRCAM.

Agon, C., M. Stroppa, and G. Assayag. 2000. "High Level Musical Control of Sound Synthesis in OpenMusic." *Proceedings of the 2000 International Computer Music Conference.* San Francisco, California: International Computer Music Association, pp. 332–335.

Ambler, A. L., and M. Burnett. 1989. "Influence of Visual Technology on the Evolution of Language Environments." *Computer* October:9–22.

Assayag, G., et al. 1999. "Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic." *Computer Music Journal* 23(3):59–72.

Cooper, A. 1995. *About Face: The Essentials of User Interface Design.* Foster City, California: IDG.

Hummel, T. A. 2005. "Simulation of Human Voice Timbre by Orchestration of Acoustic Music Instruments." *Proceedings of the 2005 International Computer Music Conference.* San Francisco, California: International Computer Music Association, pp. 495–498.

Kiczales, G., J. D. Rivières, and D. G. Bobrow. 1991. *The Art of the Metaobject Protocol.* Cambridge, Massachusetts: MIT Press.

Kuuskankare, M., and M. Laurson. 2006. "Expressive Notation Package." *Computer Music Journal* 30(4):67–79.

Laurson, M. 1996. "PATCHWORK: A Visual Programming Language and Some Musical Applications." PhD thesis, Sibelius Academy, Helsinki.

Laurson, M., and M. Kuuskankare. 2002. "PWGL: A Novel Visual Language Based on Common Lisp, CLOS, and OpenGL." *Proceedings of the 2002 International Computer Music Conference*. San Francisco, California: International Computer Music Association, pp. 142–145.

Laurson, M., and M. Kuuskankare. 2003. "Some Box Design Issues in PWGL." *Proceedings of the International Computer Music Conference*. San Francisco, California: International Computer Music Association, pp. 271–274.

Laurson, M., and M. Kuuskankare. 2004. "PWGL Editors: 2D-Editor as a Case Study." Paper presented at Sound and Music Computing '04, Paris, 21 October.

Laurson, M., and M. Kuuskankare. 2005. "Extensible Constraint Syntax Through Score Accessors." Paper presented at Journées d'Informatique Musicale, Paris, 2 June.

Laurson, M., and M. Kuuskankare. 2006. "Recent Trends in PWGL." *Proceedings of the International Computer Music Conference.* New Orleans, Louisiana: International Computer Music Association, pp. 258–261.

Laurson, M., V. Norilo, and M. Kuuskankare. 2005. "PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control." *Computer Music Journal*, 29(3):29–41.

Myers, B. A. 1986. "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy." *Conference Proceedings, CHI '86: Human Factors in Computer Systems*. New York: Association for Computing Machinery, pp. 59–66.
Nielsen, J. 1993. *Usability Engineering*. Boston, Massachusetts: Academic Press.
Norilo, V., and M. Laurson. 2007. "Development Tools for PWGLSynth." *Proceedings of the 2007 International Computer Music Conference.* San Francisco, California: International Computer Music Association, pp. 55–58.
Olson, M. K. B., and M. Seltzer. 1999. "Berkeley db." *Proceedings of the 1999 Summer Usenix Technical Conference*. Monterey, California: USENIX, pp. 42–43.
Paepcke, A. 1990. "PCLOS: Stress Testing CLOS Experiencing the Metaobject Protocol." *OOPSLA/ECOOP '90: Proceedings of the European Conference on Object-Oriented Programming Systems, Languages, and Applications*. New York: Association for Computing Machinery, pp. 194–211.
Puckette, M. 1988. "The Patcher." *Proceedings of the 1988 International Computer Music Conference.* San Francisco, California: International Computer Music Association, pp. 420–429.
Puckette, M. 1996. "Pure Data." *Proceedings of the 1996 International Computer Music Conference.* San Francisco, California: International Computer Music Association, pp. 269–272.
Schwarz, D., and M. Wright. 2000. "Extensions and Applications of the SDIF Sound Description Interchange Format." *Proceedings of the 2000 International Computer Music Conference*. San Francisco, California: International Computer Music Association, pp. 481–484.
Smith, D. C., et al. 1982. "Designing the STAR User Interface." *Byte* 7(4):242–282.
Steele, G. L. J. R. 1990. *Common Lisp the Language*, 2nd ed. Woburn, Massachusetts: Digital Press.
Taube, H. 1991. "Common Music: A Music Composition Language in Common Lisp and CLOS." *Computer Music Journal* 15(2):21–32.
Woo, M., et al. 1999. *OpenGL Programming Guide*, 3rd ed. Upper Saddle River, New Jersey: Addison Wesley.
Wright, M., and A. Freed. 1997. "Open Sound Control: A New Protocol for Communicating with Sound Synthesizers." *Proceedings of the 1997 International Computer Music Conference*. San Francisco, California: International Computer Music Association, pp. 101–104.
Zicarelli, D. 1998. "An Extensible Real-Time Signal Processing Environment for Max." *Proceedings of the 1998 International Computer Music Conference.* San Francisco, California: International Computer Music Association, pp. 463–466.