A Visual Programming Environment for Music Notation

Glendon R. Diener
Center for Computer Research in Music and Acoustics
Stanford University
gri@ccrma.stanford.edu

Abstract
This paper describes an object-oriented, visual programming environment for music notation. In the system, the images and conventions of particular notational styles take the form of data encapsulated as object specifications in an object-oriented, visual programming language. These objects are characterized not only by their state and behavior, but by a specifiable graphics image as well.

Musicians create notational objects by first subclassing system-supplied classes. Instances of these subclasses are then arranged into three-dimensional piles using direct, on-screen manipulation. Equivalent to tree structures, these piles specify hierarchical relationships among their component objects. The system's interactive behavior and audio playback facilities are provided by inter-object communication, with the branches of these tree structures functioning as channels for message passing among the component objects.

1. Introduction
This paper describes the programming capabilities of Notation [Die90], a music notation system developed at Stanford University over the past four years. While most computerized music notation systems concentrate on music printing and the “music setting” problem, Notation focuses on the compositional uses of notation, helping musicians work out their ideas in the visual realm much as they would with pencil and manuscript paper. Immediate, high-quality audio playback is provided.

Musicians program Notation in two ways: visually—through direct on-screen manipulation of musical symbols, and textually—through the creation and modification of class definitions in the system's embedded object-oriented programming language. Because of these programming capabilities, unlimited extensions to both western and non-western notation are possible, as is the creation of completely original notational systems.

2. User Interface
The basic element of Notation's user interface is the Glyph: a transparent, rectangular area of the computer screen which displays arbitrary graphics defined in the PostScript language [Ade90]. Glyphs can be dragged, copied, resized, and deleted through mouse interaction. In addition, they can be piled on top of each other. Figure 1a shows a score fragment created by first piling a staff glyph onto a page, piling clef, note, and rest glyphs onto the staff glyph, then piling a dot glyph onto the note glyph. Figure 1b shows a raised view of Figure 1a, with all glyphs made opaque to reveal their three-dimensional pile structure.

All user actions which effect an individual glyph effect any glyphs which are piled onto it. In this way entire glyph piles can be dragged, copied, resized, or deleted through mouse interaction.

Glyph piles may be frozen together, preventing the individual glyphs which make up the pile
from being inadvertently unplied until the pile is unfrozen.

Whenever one glyph is piled atop another, it is visually clipped to the other's rectangular boundary. Clipping can be used to hide portions of glyph piles, which do not become entirely visible until the user drags them off of their base glyph. In Figure 2a, this technique is used to hide details of the score fragment of Figure 1 on the right edge of the score title, "Opus I". Figure 2b shows a raised view of 2a with clipping turned off to reveal the hidden fragment.

4. Class Hierarchy

Nutation is built upon the Objective-C (ObjC) programming language [Cox86]. Following the object model of Smalltalk-80 [GR83], every ObjC object is an instance of some class. Classes define the set of methods (procedures) which their instances can respond to. In addition, they define storage locations, called instance variables, for each of their instances. All instances of the same class have the same number and types of instance variables, though the values these instance variables take on may well vary from instance to instance.

In ObjC, classes themselves are organized as a tree structure, with every class inheriting the functionality of those classes below it (closer to the root) in the structure. Class Object forms the root of this structure, so Object's methods are inherited by every other class defined in an ObjC program.

Nutation's glyphs are implemented as ObjC objects, though in Nutation, the root class (class Object) is replaced by class Glyph. Instances of class Glyph respond to a basic set of messages (method invocations) for building, copying, and storing TTrees and for displaying their images on the computer screen.

Several useful Glyph subclasses are provided by the system. The root node of every TTree (or, to put it another way, the base glyph of any glyph pile) is an instance of class Root. This class provides the animation functions needed for dragging and clipping glyph's images in response to mouse interaction.

Figure 4 shows that portion of Nutation's class hierarchy used in the score fragment of Figure 1, with
classes depicted as building blocks. As the figure indicates, all classes are built on class Glyph. Class Note (which defines methods for specifying pitch) is built upon class Rest, class Rest (which defines methods for specifying duration) is built upon class Sonata, while class Sonata (which defines methods for displaying characters from Adobe System Incorporated's popular Sonata font) is built directly upon class Glyph.

![Figure 4: A portion of Notation's class hierarchy.](image)

PostScript is itself a programming language, and glyphs display themselves by generating PostScript language code for imaging on the computer screen. Several classes exist for optimizing this display process, including a class which caches the generated PostScript, making it unnecessary to recompute this code for every redraw operation. When greater efficiency is required, the imaged PostScript can be cached as a bitmap, allowing the imaging process itself to be circumvented during redraw operations.

5. Instance Hierarchy

As we have seen, Notation implements the nodes of its TTrees as objects. The TTree is, in effect, a hierarchy of instances of class Glyph and its subclasses. An important characteristic of all tree structures is that there is exactly one simple path between any two nodes in the tree. Since objects communicate with each other by passing messages, this means that every glyph in a TTree has a unique, well-defined way of messaging every other glyph in the tree. The TTrees becomes, in effect, a channel for inter-object communication.

Methods can be designed so that these messages which invoke them propagate through the entire TTree. Notation's performance mechanism uses this feature to ensure that it performs the glyphs of a score in their proper time sequence. During a performance, the message tick is repeatedly sent to the root of the score's TTree representation. Whenever a glyph receives the tick message, it delays it immediately to any glyph directly below it. It only relays it to the right, however, if its note delay value has "tuned out". Note glyphs respond to the tick message by instructing the underlying synthesis hardware to realize the note.

Figure 5 illustrates how this scheme works. Since staffs have a time delay of zero, the two staffs in the figure are performed simultaneously. The second staff has a glyph called Pan. Because it does no drawing, Pan doesn't appear in the score fragment (although it is there, right underneath the whole note, and can be dragged by mouse interaction). Its function is to ensure that the tenor voice and the bass voice form two temporally simultaneous polyphonic strands.

![Figure 5: Notation's performance mechanism.](image)

The TTree in Figure 5 shows that the note middle C is not piled on top of its ledger line. Instead, the two glyph contexts at the same hierarchical level. If the note were piled on top of its ledger, its stem would have been clipped to the ledger's rectangular boundary.
Time delays in a TTTree need not be simple numeric values. Glyphs delay passing the tick message to the right as long as their wait method returns True. For most classes, this wait method simply decrements an instance variable every time it is invoked, and returns True only if that variable is greater than zero. Glyph subclasses, however, can define their own version of the wait method, allowing the time delay to be determined by any desired algorithm, perhaps in response to user interaction during live performance.

Whenever a glyph doesn’t understand a message, it forwards that message to the glyph’s ancestor in the TTTree. For example, if the whole note in Figure 5 needs to message its Staff glyph, it can send itself the message toStaff. Assuming Nuation is programmed so that only Staff glyphs implement toStaff, then the message will be forwarded back along the ancestor chain (shown by dotted lines in the figure) to Part, and then finally to Staff, where the corresponding method will be executed.

Nuation sistematically generates a number of messages in response to user interaction. Such messages as wasPiled, mouseDragged, and didResize allow individual Glyph classes to customize how they respond to the corresponding user actions. To illustrate, the ledgerline in Figure 5 was not placed there by the user. Instead, the Staff glyph created it in response to a wasPiled message, which is generated by the system whenever the user places one glyph on another.

6. Programming Environment

For many applications, Nuation can be programmed entirely in the visual, graphic domain by dragging and piling glyphs on the computer screen through mouse interaction. This alters the underlying TTTree data structure, redefining the message channels among glyphs, and thereby causin the TTTree to respond differently to various messages. New glyphs themselves may be created by clicking on their names in a hierarchical class browser. Alternatively, pallets of glyphs can be built up and used as source material for composition.

The process of defining new Glyph subclasses is done textually. Nuation provides customized editors for this purpose, and features dynamic compilation and linking of ObjC, so Glyph subclasses can be built entirely interactively. Immediate execution of ObjC statements is provided, giving programmatic control of glyph hierarchies. TTtrees can be saved to disk for later retrieval. Classes needed to support the glyphs in a TTtree are loaded on demand when such files are retrieved from disk.

7. State of the project

Nuation has been fully implemented on the NeXT computer. Real-time sound output of its scores is provided through NeXT’s music kit [JB89]—a library of ObjC classes which implement sound synthesis on the machine’s digital signal processing hardware. A fairly complete subset of common musical notation glyphs has been implemented, as has a simple piano roll editor. Recently, the system has been used as the editor for optical music recognition software developed at McGill University [Fu52]. Nuation is currently available as free, public domain software.

References

[Fu92] Ichiro Fujinaga, Bo Aplonhce, Bruce Pen- nycook and Glesdon Diener. Interactive Optical Music Recognition. (In these proceeedings).