CENTER FOR COMPUTER RESEARCH IN MUSIC AND ACOUSTICS
MAY 1989

Department of Music
Report No. STAN-M-60

DMIX:

AN ENVIRONMENT FOR COMPOSITION

Daniel V. Oppenheim

CCRMA
DEPARTMENT OF MUSIC
Stanford University
Stanford, California 94305

DMIX:

AN ENVIRONMENT FOR COMPOSITION

Daniel V. Oppenheim

Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music, Stanford University
Stanford, CA 94305
Dan%CCRMA-F4@Sail.Stanford.edu

Abstract

The integration of real-time editing, highly interactive graphics, functional programming and alphanumeric coding are important features of Dmix -- an object oriented environment for Computer Aided Composition. Fast turn-around times encourage compositional experimentation. Music objects can become the tools which modify other music objects, thus improving the correlation between a desired musical result and the means for obtaining it.

Foreword

> *the 'problems' of Computer Music are no longer that of technology but rather of our ability to control it* (Mathews, 1989).

Providing the means to easily express musical ideas, to experiment with them, and to organize them into a musical composition seems the essence of Computer Aided Composition. It is becoming generally recognized that the ability to provide such means is, nowadays, less a factor of synthesis capability (technology) and more that of environment design, or more specifically—the user interface. Designing a 'good' interface is by no means trivial (Oppehneim 1986). In fact, it seems safe to assume that no two composers would select the same environment, interface, or tool for handling a given musical problem. This choice is not only personal but also differs as progress on a composition is made.

BRIEF OVERVIEW

Dmix is an object-oriented framework for creating, editing and interacting with music, and is implemented in Smalltalk-80 (see Pope 1989, 1987). Its main motivation was to design an easy-to-use and yet flexible environment that has a uniform user-interface, that is easily extendible, and that is independent of any synthesis hardware (see Oppenheim 1986, 1987). If hooked to a real-time synthesizer, editing can be carried out in real time with the provision of instantaneous audio and visual feed-back (see Mathews 1970, Buxton 1980). Dmix can also be used effectively as a high-level user-interface to non real-time

software-synthesis packages or in conjunction with environments such as PLA and SAMBOX (Schottstaedt 1984).

A major goal in designing the user interface was to enable composers to work with minimum interruption of the creative process—once a musical idea has formed it should be easy to find a way to implement it; during implementation the composer should not have to spend time writing code or consulting operating manuals. Dmix offers a rich variety of tools for creating, editing and modifying music objects. At any point the composer may choose to work via text editors, graphic editors, real-time editors, applicative (functional) programming, or conventional alphanumeric programming. All tools are equally available to the user at all times and may be used in any sequence or in conjunction with each other. Each tool is simple in conception and therefore easy to use. Tools are easily extendable and can quickly be tailored to comply with specific needs. Beyond that, all of Smalltalk's standard features are readily available: objects may be created, inspected and modified by writing Smalltalk code.

A unique feature in Dmix is the ability to create tools directly from music objects and vice versa (see example 3). This enables composers to think of tools and of ways to use them in more familiar musical terms that may help correlate between a desired musical effect and the means to produce it. This is also significant in that it offers alternative ways to deal with important musical concepts such as motivic treatment and development.

Editors

Classes for editing include TextEditors, GraphicEditors, RealTimeEditors, and HierarchyEditors (note that class names are capitalized in the Smalltalk convention). Whenever an edit view is opened, a new music object is created and the original is kept unchanged. The user can play, inspect or open different editors on any of the two versions and compare them. Versions of the new object can be saved as work progresses; a final save will replace the edited version with the original throughout the system.

TextEditors benefit from every music-object's ability to save and create itself from an ASCII stream. The user may switch between absolute or relative time-tagging and all standard text-editing facilities are available.

GraphicEditors are flexible work spaces that can modify events via the mouse, alphanumeric coding, applicative programming, or real-time editing. Mouse-oriented editing allows inserting events, deleting, fitting parameters to lines and curves, and so on. Music Events can be grouped in Selections that enable higher level editing actions. Selections can be dragged, stretched, copied into other views, etc. Selections can also be modified by blocks of code from the editor's CodeDictionary (to be described, see example 2). More powerful actions can be carried out by applying Modifiers to Selections, as will be described (see example 3).

Real-time editing is accomplished by connecting external input devices, such as joy-sticks, faders or Midi controllers, to specific parameters of the music events (see example 1). During an edit session a copy of each music object is made just before it's playback time, its parameters are computed in relation to the input device's position, the new object is played and the EditView displays the new

parameters over the original. Both audio and visual feedback are instantaneous and the user may adjust the input device accordingly. The original music objects remain unchanged and new EventLists are automatically created for the edited objects and for the input-updates.

HierarchyEditors manipulate the hierarchical structure of a composition and provide a high level view of the music (see example 4). They are especially useful in managing large compositions, where each section is composed of many layers. Each section could be displayed and treated as a single object, or opened up to display all of its components to allow for detailed editing. They may be used in a top-down or bottom-up approach, or in any combination of the two.

Extendibility via CodeDictionaries

The ability to perform high level operations is both the advantage and drawback of any tool. Problems begin when musical situations arise that are slightly different than those for which it was intended. In such cases the user is compelled to either compromise his initial concept or to interrupt the musical activity in order to design the necessary tool. In Dmix this problem is overcome by using CodeDictionaries.

CodeDictionaries provide the power and flexibility of general programming with minimal interruption of musical activity. They store blocks of compiled Smalltalk code and are used by tools to manage the code that determines their action (see the LegacyDictionary in Diener 1989). For example, a Filter can be in *bandPass* or *bandReject* modes, depending upon the block selected by the user. Inspecting the dictionary pops up a window in which the user can modify existing code or write new code that is immediately compiled and ready for execution (see example 2). The real advantages of CodeDictionaries become apparent during a work-session as they can be used within the musical context at hand and they enable the easy modification and extension of any tool's functionality while it is being used.

Modifiers

Modifiers are tools that are typically applied to notelists in order to modify some parameter as a function of time (see Mathews 1970, Backus 1978, Harrison 1985, Kopec 1985, Dannenberg 1986). For example, a Function could set, offset, or scale amplitudes in a notelist (see example 3). Other Modifiers include Tables, Filters, Quantizers, and ModifierInterpolators.

Each Modifier represents data (i.e. brake points in the case of the Function) and some mathematical operations (*, /, +, =, etc.). Data can be derived from music objects and then edited with text or graphic editors. The algorithms that implement the mathematical operations are stored in CodeDictionaries so that they may be accessed and modified.

Music Objects

Dmix merely manipulates music objects, and it does so by sending them messages. Music is therefore represented by the collection of objects and messages that produce it. A single abstract class - MusicEvent - represents all music objects. It has two abstract subclasses: TerminalEvent and EventList. A

TerminalEvent could represent a single note, one sample, an entire sound file, a parameter update, a Smalltalk message, a Midi event, a DSP synthesis patch, etc. EventLists are collections that contain TerminalEvents and/or other EventLists. Protocol in these three classes defines generic playing methods, ensures that all music objects will respond to Editors and Modifiers in a uniform way, and provides several formats for saving and restoring objects from disk.

There is no fixed notion of a 'score' in Dmix and the user is free in constructing a model for his music. If synthesis is carried out by a Music-V-like package the user may want to model notelists and think of their collection as a score; if Midi is used the user is able to emulate Midi tracks; a more complex tree structure could also be used (such as 'TTree', see Diener 1989) —Dmix is unbiased. There is protocol in EventList for managing arbitrarily complex hierarchies, parsing and enumeration.

MUSIC EXAMPLES

Example 1: Interactive Real-Time Editing

The Bach prelude in C major is a Midi sequence generated in Smalltalk and will be used in several examples. Figure 1 shows two EditViews open on the same prelude. The bottom view is monitoring pitch (midi key number) and duration in a piano-roll notation. The top view is monitoring velocity; note that all notes have equal velocity. Each view was assigned to a separate Input device for simultaneous real-time editing. The black continuous lines were drawn in real-time and indicate the new parameters. During editing a copy of each music object is made just before it's playback time, it's parameters are updated according to the current position of the Input device, the copy is played and the EditView displays the new parameter value over the original. Both audio and visual feedback are instantaneous and the user may adjust the input device accordingly.

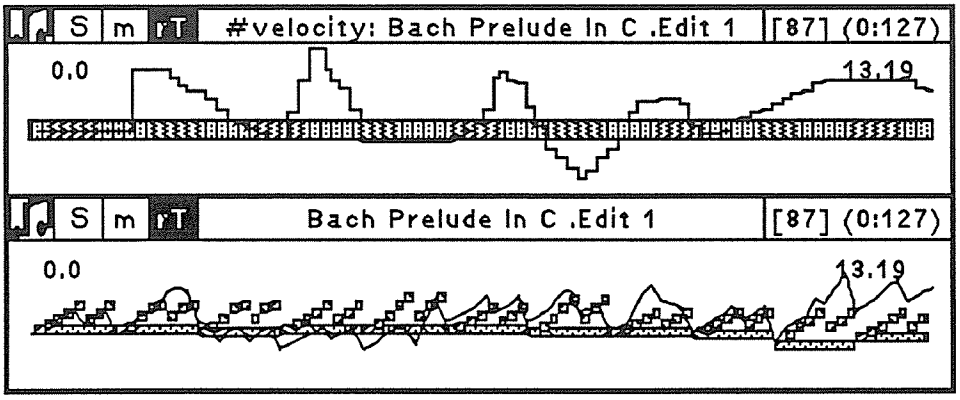

Figure 1: Real-Time scaling.

It should be noted that scaling empty EditViews will create new events with parameters that are analogous to the gestural input. A useful example of this feature for Midi could be the creation of a system-exclusive stream for controlling an existing Midi-track. This can enable continuous control of synthesis parameters not readily available through standard Midi techniques.

Example 2: using CodeDictionaries

This example demonstrates the ability of applying a block of code to a Selection of music events within a GraphicEditor. The top edit view displays the selected notes enclosed in a rectangle. Selecting the **do Block** option popped open a CodeDictionaryInspector in which a new block named **crazy** was written. The entire code for **crazy** can be seen above the EditView. The black notes in the bottom view are the result of applying **crazy** to the selected notes: their onset-times were shifted back two seconds, they were transposed two octaves up and their duration was lengthened by a factor of three. The remaining notes, in gray, are unchanged.

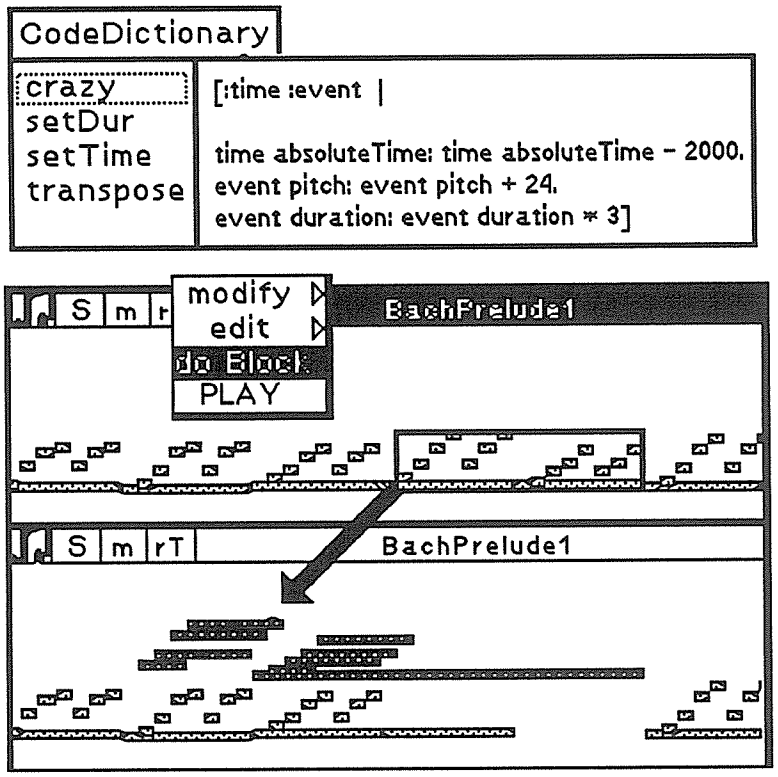

Figure 2: Applying a block of code onto a Selection of notes from the Bach prelude.

Example 3: Functional Programming with Modifiers

This example demonstrates the ability to apply Modifiers to music objects (see Mathews 1970, Backus 1978, Harrison 1985, Kopec 1985, Dannenberg 1986). Since both music objects and Modifiers are merely representations of data, each can be created with data derived from the other. Figure 3 is an example of mapping the rhythm from a Midi recording named **Jazzy Tune** onto the Bach prelude. A Function is first created from the **Jazzy Tune** and then applied to the prelude. The entire process was completed in less than a minute.

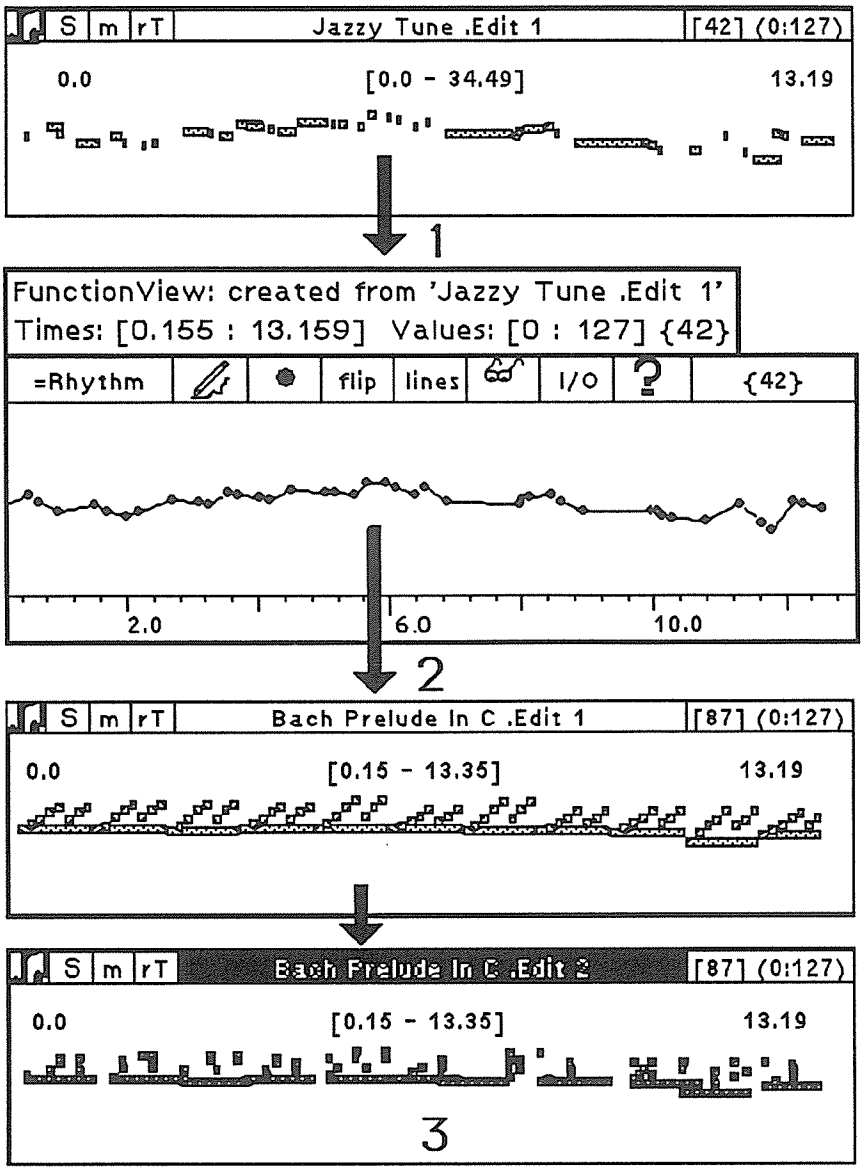

Figure 3: Mapping rhythm from a **Jazzy tune** onto the Bach prelude.

Step 1: A Function is created from the notes displayed in **Jazzy Tune**. Begin times and pitches are read as X-Y coordinates and stored in the Function's Table.

Step 2: The Function is applied to the Bach prelude in **=Rhythm** mode. This mode is one of several stored in the Function's CodeDictionary and effects only the time-domain. This code may be edited by clicking the **=Rhythm** button.

Step 3: As a result, begin-times of notes in the prelude are shifted to match the closest begin-time in the **Jazzy tune**.

Note that the selected mode in this example, **=Rhythm**, has effected only the time-domain. Other modes could also modify the pitch-domain, or both. Applying the Function to an EditView on a parameter other than pitch would modify that parameter. A Function is also capable of scaling and transforming the data in it's Table into any time-value window coordinates. Applying it to another Function would produce a new, more complex, Function.

Example 4: Hierarchy Editors

HierarchyEditors open on EventLists that contain other EventLists, and can manipulate the hierarchical structure of a composition (see figure 4). In this example, **Composition** consists of two themes. **Theme1** is the product of two layers: **DSP flute** and **Midi piano**. **Theme2** is made of **glass sounds**, **wind** and **vocals**.

The HierarchyEditor distinguishes between EventLists that contain TerminalEvents, i.e. the layers, and those that contain yet other EventLists, i.e. the themes and composition. A layer is considered a *child* and colored gray whereas each theme is a *father* and colored black (note that this distinction results only from the way that music objects are being used; in fact, there is nothing to prevent a *father* from being inserted into one of his own *children*). The time relationship between objects can be adjusted by dragging their display; music objets can be inserted anywhere in the tree allowing the addition of new layers, themes or levels; any element can be spawned and edited independently—if it is a *child* then a regular EditView will open.

A 'hide' feature enables the hiding and redisplaying of any *father's children*. This allows the composer to group elements that form a higher level idea and treat them as a single object. Any edit action performed on a *father* will also effect its *children*, hidden or not. It is always possible to redisplay hidden *children* in order to rework them.

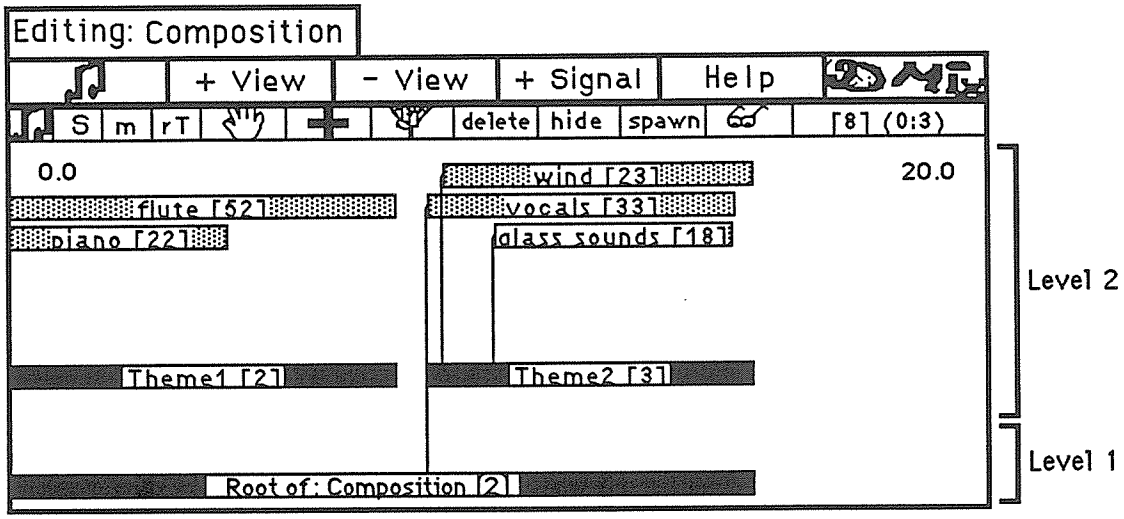

Figure 4: Hierarchy View.
Numbers in square brackets indicate the number of Events in each EventList

Conclusions and Future Prospects

Dmix provides a framework for composing that is independent of synthesis-hardware. During the past several months it readily adapted to diverse compositional activities ranging from algorithmic composition to interaction with a performer in real-time. Fast turn-around times, audio and visual feedback, and real-time editing capabilities, encourage compositional experimentation and help improve the correlation between a desired musical result and the means for obtaining it. A feature that seems particularly significant for composition is the ability to transform music objects into Modifiers that can, in turn, be applied to other music objects. This offers composers alternative ways of thinking about transforming musical materials and may lead to new approaches in dealing with musical concepts such as motivic treatment and development.

Dmix has just made it's first step into the world and many refinements are yet to be made. We would like to see the addition of components such as a more meaningful music/sound representation (see Oppenheim 1987), synthesis patch-editors, and sound file editing. However, it is hard to imagine that any single implementation could encapsulate such a wide range of functionality efficiently. We hope that a joint effort within the community might lead to a formulation of a 'standard' in music-representation that will allow the sharing of such resources between diverse music applications.

Availability

Dmix will be made available at no cost to interested parties. The author should be contacted for details.

Acknowledgements

It would be impossible to mention all the people at CCRMA who's constant encouragement accompanied this project. Guy Garnett sparked off the initial inspiration and helped construct the basic music classes. Lounette Dyer and Glendon Diener aided in implementing the CodeDictionary. Amnon Wolman and Richard Karpen were willing to be the first guinea-pigs.

References

Backus, J. 1978. "Can Programming Be Liberated from the von Neumann Style?," *Communications of the ACM* 21(8):613-641.

Buxton, W., Reeves, W., Fedrokow, G., Smith, K. and Baecker, R. 1980. "A Microprocessor-Based Conducting System," *Computer Music Journal* 4(1):8-21.

Dannenberg, R., McAvinner P. and Rubine. D. 1986. "Arctic: A Functional Language for Real-Time Systems," *Computer Music Journal* 10(4):67-78.

Diener, G. 1989. "TTree: A Tool for the Compositional Environment," *Computer Music Journal* 13(2):77-85.

Harrison, P. and Khoshnevisan, H. 1985. "Function Programming Using FP," *Byte* August 1985:219-232.

Kopec, G. 1985 "The Signal Representation Language SRL," *IEEE Transactions on Acoustics, Speech, and Signal Processing* 33(4):921-932.

Mathews, M. and Moore, F. 1970. "GROOVE - A Program to Compose, Store, and Edit Function of Time," *Communications of the ACM* 13(12):715-721.

Mathews, M. V. 1989. private communications.

Oppenheim, D. 1986 "The Need for Essential Improvements in the Machine-Composer interface used for the Composition of Electroacoustic Computer Music," *Proceedings of the ICMC*, the Hague.

Oppenheim, D. 1987. "The PGG Environment for Music Composition - a Proposal," *Proceedings of the ICMC*, Urbana Illinois.

Pope, S. 1987. "A Smalltalk-80 Based Music Toolkit," *Proceedings of the ICMC*, Urbana, Illinois.

Pope, S. 1989. "Machine Tongues XI: Object-Oriented Software Design," *Computer Music Journal* 13(2):9-22.

Schottstaedt, B. 1984. "PLA - A Tutorial and Reference Manual," CCRMA report No. STAN-M-24, Department of Music, Stanford University.