A Structure for Representing, Displaying, And Editing Music

Roger B. Dannenberg
Computer Science Department and
Center for Art and Technology
Carnegie-Mellon University
Pittsburgh, PA 15213 U.S.A.

Abstract

An extensible data structure has been implemented for the representation of music. The structure is intended for use in a display-oriented music editor and has several features that support interactive editing. The structure consists of a set of events, each with a list of attribute-value pairs. In addition, events can have multiple views, and an inheritance mechanism allows sharing of data between views. Multiple hierarchies can be encoded within this structure. A version system facilitates an efficient undo operation and incremental redisplay. At the same time, versions allow client programs that update the structure to be written in isolation from programs that present the structure on a graphics display.

1. Introduction

A screen-oriented interface that allows music notation to be entered, modified, and displayed in traditional notation is high on the list of resources desired by many composers and researchers working in the field. Such an editor can ease the task of producing printed scores and copying parts. In addition, it can provide a foundation for educational programs to teach music theory, sight-singing, and orchestration. A sufficiently flexible editor can do much more. It can serve as an interface to various forms of software and hardware synthesis, it can provide a standardized representation for music so that output from one program can serve as input to another, and it can be extended to support new notations without the major software effort required to implement a new notation from the ground up.

I am working on such an editor as part of an advanced computer music system called the Musician's Workbench. At this time, a data structure for supporting the Musician's Workbench editor has been implemented, and a very simple editor has been constructed for testing and evaluation. The data structure is important because it provides all facilities for representing data and most of the mechanisms for interactive editing and redisplay. The data structure itself does *not* specify how music is to be encoded, so a fair amount of flexibility is obtained in this area. I have made every effort to avoid making decisions at the data structure level that would preclude notational or representational choices at a higher level.

The principal terms and features of the data structure are outlined in the following paragraphs.

Scores. A piece of music is represented by a structure called a *score*.

Events. A piece of music, or *score* is a set of *events*: data objects which may be created, deleted, or modified.

Properties. An event has a list of associated *properties*, each of which has an *attribute* or name, and a *value*.

Hierarchies. Events can be related in multiple hierarchical structures. For example, an event may represent a note which is a member of several overlapping phrases. The note may also be a member of a chord. These structural relationships can be represented explicitly as instances of hierarchies.

Views. An event can participate in any number of *views*, and events may have view-specific properties. For example, the first violin part of an orchestral score would

Dannenberg, "A Structure for Representing, Displaying and Editing Music," in Proceedings of the 1986 International Computer Music Conference, Den Haag, Netherlands, October 20-24, 1986. San Francisco: International Computer Music Association, 1986. pp. 153-60.

be represented as a view. A piano reduction could be another view.

Incremental Update. A view can be associated with one or more regions of a computer display. When visible events of a view are modified, mechanisms provided by the data structure allow view software to modify the display incrementally. This is accomplished by automatic bookkeeping that records what has changed and what views are affected by the change.

Undo. The data structure uses a history mechanism to enable previous versions of the data structure to be restored. Undo can be applied iteratively to undo a sequence of changes (called Undo-more) and recursively to restore a change that was undone (called Undo-less). This facility is completely integrated with the incremental update mechanism so that changes can be undone rapidly.

Permanent Storage. Scores can be written to a file and read back into memory. Circular data structures and shared structures are properly handled. Assuming music editing exhibits locality of reference, then saving and restoring a score will, as a side effect, rearrange its placement in virtual memory so as to optimize paging performance.

Event Sets. Events can be collected into sets (implemented as hierarchies). Abstract operations exist for creating and manipulating event sets. For example, one operation creates a set containing every event such that the value of a given property is greater than a given amount. Set union and intersection are also implemented.

Editing Operations. A collection of abstract editing operations is provided. For example, one operation takes every event in a given set and increments the value of a given property by a given amount.

The use of property lists rather than some predetermined, fixed data structure allows new information to be added to the structure quite easily. For example, a composer could edit a composition in common music notation, but attach additional timbral information to each note. This information would be ignored by the standard display routines, but could be accessed and used by a synthesis program. Properties might also be used to store performance information such as actual durations as opposed to theoretical durations computed according to beats and tempo. This approach differs from earlier systems that either are restricted to traditional notation [3, 6], are designed to represent typographical information only [1, 5], or have simpler structure representation facilities [4, 2].

All of this functionality is implemented in C and runs on Sun, VaxStation, and IBM-RT workstations. The current music editor provides a "piano roll" notation where pitch is indicated by position on a grand staff, and time is indicated by horizontal position. A note is indicated by drawing a heavy horizontal line from the note's starting time to its ending time at the appropriate vertical position corresponding to the note's pitch.

The next stage of development is the creation of a set of music fonts and the development of a representation for traditional Common Music Notation within the existing data structure. This work is currently in progress.

1.1. An Outline of What Follows

In order to understand the present design, it is necessary to consider the problems the design is meant to overcome. There are a number of problems, so the resulting design is fairly complex. In the next section, I hope to show that these problems are important and to convince the reader that they require special treatment in a score-editing program. Section 3 describes the major features of the design as seen by the implementer of editing commands, and Section 4 extends the design with a description of the view mechanism. Then, Section 5 shows how the structure can support multiple hierarchies. Versions form the basis of undo and incremental redisplay operations, and are described in Section 6. Section 7 shows how the design is intended to be used, and Section 8 evaluates the design in terms of the problems it is intended to solve and looks at other problems that have arisen. Finally, conclusions are presented in Section 10.

2. Motivation

The present design is intended to address five critical problems in the implementation of a display-oriented editor. First, mechanisms must be provided for updating the display to reflect the represented information. Second, it should be possible to control the granularity of redisplay, not necessarily changing the display after every lowest-level change to the information. Third, information access and modification should be computationally efficient. Fourth, it should be efficient to "undo" a sequence of operations when the user decides he has made a mistake. Finally, the structure should support abstraction in the form of multiple hierarchies.

2.1. Display Consistency

The first problem is that of keeping the display consistent with the data. In many cases, it is computationally unfeasible to recompute the entire display after every modification to the underlying structure. Therefore, the display must be updated incrementally.

2.2. Granularity

The second problem is that it is often desirable to control the granularity of display updates. For example, suppose the user wants to move a box diagonally, and suppose that the position of the box is represented by a horizontal and a vertical coordinate. Furthermore, suppose that the user's move command is implemented by assigning a new horizontal value followed by a new vertical value. A straightforward implementation might attempt to redisplay the box at its new horizontal position and then redisplay it at its final position. This could require more computation than a single redisplay and might result in confusing and esthetically undesirable changes to the display.

2.3. Efficiency

The third problem is to support efficient modification of information. I want the music editing system to be extensible so that users can perform arbitrary computations on scores within the context of a nice interface. Therefore, it is important that these computationally intensive tasks do not suffer a heavy performance penalty because of display mechanisms. For example, changing a note duration in memory may only take a microsecond, but updating a view of that duration may take many milliseconds. In order to support applications that require intensive computation, I would like to pay the display penalty only in proportion to the amount of redisplaying that actually takes place.

2.4. Undo

The fourth problem is the provision of an Undo command that restores a previous state of the edited information. While checkpointing techniques are useful for recovering from catastrophic errors, it is nice to undo changes in time proportional to the size of the change rather than proportional to the total amount of information. Furthermore, the undo mechanism should not require commands to explicitly save information when modifying information. This would greatly complicate command implementation.

2.5. Hierarchy

The fifth problem is that the structure must support multiple hierarchies. The problem is mentioned here to emphasize the importance of this aspect of the structure. In our terminology, a hierarchy represents a class of relationships. For example, a beam hierarchy is one that describes the relationships of beams to notes. A *hierarchy instance* can be thought of as one node in the hierarchy; for example, a single beam might be represented as an instance of the beam hierarchy. A *hierarchy instance* is said to contain members; for example, a beam contains notes and perhaps other beams. The concept of *multiple* hierarchies means that hierarchy membership does not necessarily form the well-nested structure of a single hierarchy. For example, two phrases may share a note, and a beam hierarchy may be contained only partially within a slur. These examples illustrate the need for a very flexible hierarchy representation system.

3. The Score Data Structure

My solution to the problems described in the previous section is based upon a fairly elaborate data structure called a Score, which has several levels of interface. The highest level is seen by *clients*, that is, programs that manipulate score information, normally in response to a user command. To a client, the Score appears as a set of entities called *events*. Each event contains a set of attribute-value pairs called *properties*. In the current implementation, attributes are Lisp-like atoms, and a value can be either an atom, an integer, a floating-point number, a reference to another event, a list of values, or a special Null value. In addition, an "indirect" value is supported so that many properties can be bound to one variable. Changing the variable modifies all corresponding properties. This additional mechanism is completely integrated with Undo and display update facilities, but further details are beyond the scope of this paper.

The client can perform the following operations:

put_value (event, attribute, mode, value);

value := get_value (event, attribute, mode);

event := next_event (event);

event := score_event (event);

The first two operations allow properties to be written and read. (The *mode* parameter will be explained below.) The third operation allows the programmer to iterate through all events. Events are linked in a circular list, with a distinguished event serving as a list header. This header event is a convenient place to store properties that apply to the score as a whole. It is appropriate to refer to this event as "the score" since a reference to it gives access to the entire score structure. For reasons we will learn later, it is often necessary to locate the score given one of its events. The fourth operation takes any event and locates the corresponding score, that is, the event at the head of the circular list of events. Figure 3-1 illustrates a score structure with four events (including the score) represented by circles, and a number of properties represented by boxes. The score event is distinguished by a double circle.

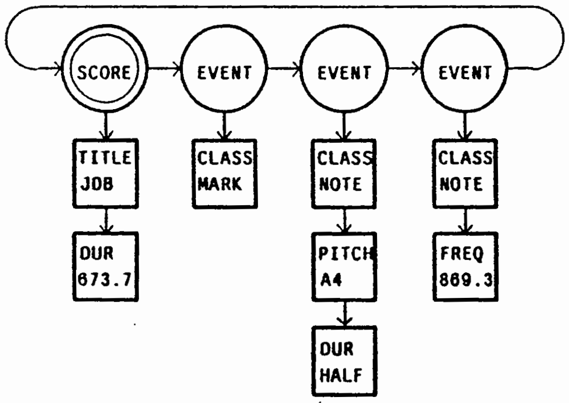

Figure 3-1: A score structure with events and properties.

4. Views

Before describing views, let me try to explain how they came about. One of my assumptions is that a small amount of manual positioning and touch-up will be necessary to achieve high-quality printed scores. This implies that there will be detailed typographic information stored in the data structure. The problem arises when an entity (a note, for example) appears in more than one presentation of the score. For example, a note might appear in a french horn part in F and in the full score in C. These two different presentations, or views, of the score may require conflicting sets of typographical information. To handle this, there must be a way to associate properties with a specific presentation. At the same time, some properties should be shared by all presentations in order to save space and encourage consistency. (Consistency is easier to maintain if a piece of information is stored in one and only one place.)

The data-structure that corresponds to a presentation is called a *view*. A view is a set of view events, some or all of which are related to score events. A separate property list is provided for each view event, and the resulting structure is illustrated in Figure 4-1. As indicated by the

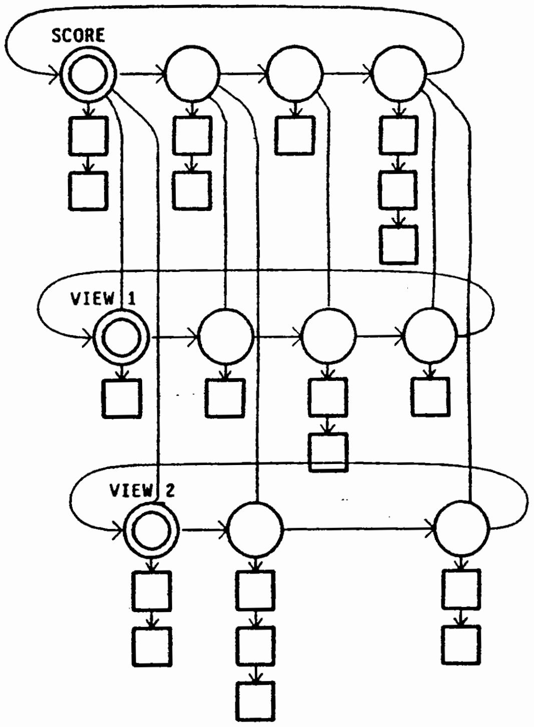

Figure 4-1: A score with two views.

figure, each view is similar to the score structure shown earlier in Figure 3-1, except view events may be linked to a corresponding score event, as illustrated by the vertical arcs between view events and score events. In our implementation, the score event corresponding to a view event is accessed by reading the EVENT property of the view event. Starting from a score event, we can find the corresponding view events by getting the VIEWS property from the score event. The value (if not Null) will be a list of events. A score may have any number of views.

Now we can put information that is common to all views on the score event, and information that is particular to a single view can be stored on the corresponding view event. The *mode* parameter in the *get__value* and *put_value* operations is used to specify where properties are stored and accessed as follows: If *mode* is Local, only the view event is modified or accessed. If *mode* is Any, and the operation is *get_value*, then the view event's property is searched first. If no property is found with the desired attribute, and if there is a corresponding score event, then the score event's properties are searched. If *mode* is Any, the operation is *put_value*, and a corresponding score event exists, then the property is placed on the score event and any existing property with the given attribute is removed from the view event. On the other hand, if no score event exists, the view event is updated as if *mode* were Local.

5. Hierarchy

In order to provide a general structure for music representation, events can be related in hierarchies. A hierarchy is represented in terms of events and properties. For example, one might want to represent a specific slur as a hierarchy containing the notes (events) under the slur. To do this, an event (call it *slur-event*) is created with the property [CLASS: SLUR] and the property [SLUR: *slur-event*] is added to each note under the slur. In the implementation, *slur-event* is a reference to (memory address of) the *slur-event* structure. This approach is flexible in that multiple hierarchies can be created, events can be members of several hierarchies, and hierarchies do not have to be nested. Hierarchies have the nice property that an event can quickly find the hierarchies of which it is a member. Figure 5-1 illustrates two hierarchies which are not themselves hierarchically related. Operations are provided to simplify the manipulation of hierarchies. The principal operations are:

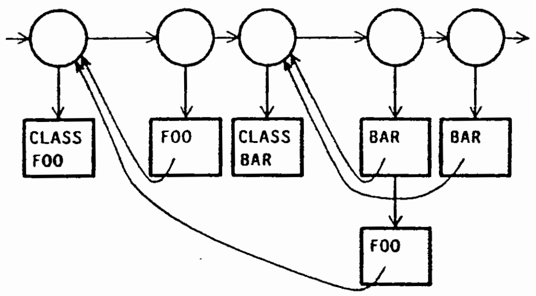

Figure 5-1: Two abstract hierarchies and their representation as events.

Create a new hierarchy event, inserted after *previous event*:

create_hierarchy(previous_event, class);

Test to see if an event is a member of a hierarchy instance:

member_hierarchy(event, class, instance);

Remove an event from a hierarchy:

rem_from_hierarchy(view, event, mode, class, instance);

Remove an event from the score (or delete a hierarchy instance):

rem_from_score(event, score);

6. Versions

The incremental redisplay and undo facilities are based on the idea of versions. Each cycle of command entry, data modification, and redisplay is associated with a new version number. Although invisible to most clients, the score data structure includes a version number for each property, so a property is really a triple: [attribute, version, value]. The current version number is maintained in the global variable *current-version* and the operation

set_version(version)

is invoked to change the current version. Conceptually, *put_value* works by inserting a new property with the current version number at the head of the list. Old properties are never removed or modified, even if they have the same attribute. The operation *get_value* works by scanning the property list from head to tail for the first property with a matching attribute and a version that is less than or equal to the current version. To avoid complications, if *put_value* is invoked at a given current version, then it cannot be invoked later with a lesser (earlier) version. In other words, modifications are made with non-decreasing versions. In intuitive terms, "you can't change history."

6.1. Undo

Given the version mechanisms described above, it is relatively straightforward to implement an undo facility. Since previous versions are accessible, we can undo the last change as follows: First, locate properties that are the result of changes to be undone. These properties will have a particular version tag if we want to undo only the latest version. Second, for each property to be undone, find the previous value for the given attribute and perform a *put_value* operation of that value at the highest version.

Note that we can even undo an undo operation with no additional support. The following example illustrates this process. Suppose we have just completed the construction of version 10. To undo version 10, returning the structure to its state in version 9, we begin by finding attributes that were changed, reading their values at version 9, and writing these values at version 11. Version 11 will now have the same values as version 9. Now, suppose the user decides to undo his undo command. This is accomplished by reading version 10 and writing the values at version 12.

6.2. Incremental Redisplay

As mentioned earlier, versions also facilitate redisplay. By comparing the previous version to the present one, a redisplay program can determine what has changed, thereby making minimal changes to the display. A simple but effective technique is to erase the image of any event that has changed, using the previous version of the event to determine what to erase, and then redrawing the event using the current version. To support redisplay more completely, a property with attribute MODIFIED maintains a list of attributes of properties that have changed since the previous version. Thus, if the PITCH and ACCENT properties of an event were modified, then the MODIFIED property would have the value (PITCH ACCENT). The MODIFIED attribute is always omitted from the list. If any attribute of a score event is modified, the MODIFIED property is updated if necessary, and in addition, EVENT is inserted into the MODIFIED list of each corresponding view event. Note that corresponding view events are quickly located by reading the VIEWS property and that EVENT only need be added to MODIFIED properties of the view the first time that the score event is modified. A redisplay routine can find out what events have changed by looking for events with the MODIFIED properties. The nature of the change can be roughly ascertained by looking at the value of the MODIFIED property, and the exact nature of the change can be determined by looking at the previous version of each property whose attribute is on the MODIFIED list. In cases where the score event changed, it will be necessary to locate the score event (by calling *get_value(event,* EVENT, Local)) and then looking at the MODIFIED list of the score event and the previous version of the score event.

7. Application

The typical application program that uses the Score structure is an editor, and the editor operates in a cycle of four phases. Each cycle corresponds to a version of the structure. In the first phase, the user enters a command. In the second phase, the command is interpreted and, as a result, data is modified through calls to *put_value* and *new_event*. Events that are visible and modified will have the attributes of modified properties put on the MODIFIED list. Note that the MODIFIED list is maintained as a side-effect of *put_value*, and in this way, support for redisplay is decoupled from editing operations.

In the third phase, a display update routine is called for each view. The display update routine finds the changed events either by traversing a list of modified events or by traversing the events looking for *modified* flags set by *put_value*. When a changed event is found, the update routine can redraw its representation of the event or perform some more sophisticated incremental update. Complete information about what has changed can be accessed quickly through the MODIFIED list and by reading the previous version.

An elegant example is the redisplay routine used in the current prototype, which graphs notes as horizontal bars. The vertical axis represents pitch and the horizontal axis represents time. When a note changes, the redisplay routine decrements the version to be used by access routines. It then sets the window manager's paint color to white and draws the note. The effect is to "white out" the old image of the note. Then, the access version is incremented (to the most recent version), the color is changed to black, and the note is drawn again. This repaints the note, making it visible and reflecting its new properties. Common practice notation will require more elaborate redisplay procedures, but at least the data structure provides all the information necessary. A likely scenario is that some medium-sized chunk of display, say one measure, will become the unit of redisplay.

The fourth phase is the cleanup phase in which MODIFIED lists are deallocated and the version is incremented. All of Phase 4 is an operation of the Score structure and is not something an application programmer must write or even fully understand.

8. Evaluation

One might now ask, how has the Score structure made programming easier? Let me return to the set of problems outlined in Section 2 and discuss how they are solved by the Score structure.

The first problem is keeping the display consistent with the data. The Score structure supports incremental display updates by providing the redisplay routines with information about what changes were made. This information is maintained by the structure access routines, so editing operations by the client need not include any code concerning the display. This is an important software engineering consideration, and it leads to modular structure in which editing operations are cleanly separated from display maintenance.

The second problem is controlling the granularity of display updates. This is accomplished in part by not constraining how the display is updated. The main control over granularity is through the version mechanism. In Phase 2, the application program is free to make arbitrarily many updates to the structure without any display changes. Only when the application advances to Phase 3 is any display updating performed.

The third problem is efficiency. There is not space here for a complete analysis, but it is worth noting that the overhead of accessing data in this structure is never more than a small fixed cost per operation. Mechanisms have been implemented to prune the structure of old versions with very little overhead, although a detailed explanation has been omitted from this paper.

The fourth problem is the provision of an Undo operation. The facility discussed in Section 6.1 has several interesting properties. First, Undo is completely independent of the client; that is, no special code need be written by the application programmer to support the Undo operation. Secondly, the Undo mechanism can be applied to itself with no extra work to provide an "undo undo" operation. In fact, the "undo undo" operation can also be undone, and so on. Third, the Undo operation is completely compatible with incremental redisplay. Performing an Undo has the same effect on the Score structure as calling *put_value* to make the necessary changes. Therefore, the MODIFIED lists and version number are changed as usual, and a display update routine can be called as usual. No special provisions are needed in the display routine to handle Undo.

The fifth problem concerned the representation of hierarchies and was discussed in Section 5. The Score structure solves all of the problems posed in Section 2; however, no data structure is perfect, and even the Score structure has some limitations and undesirable properties. These are discussed in the next section.

9. Remaining Problems

To give the reader a balanced assessment of the Score structure, we will consider some potential liabilities of its use. First, there is the obvious point that the structure organization itself may be inappropriate. For example, random access to events or associative lookup by properties is not directly supported. It is likely that for any given application a more compact representation could be found. Thus the Score structure has a restricted set of access methods and is not optimal in its memory requirements.

Another problem of the Score structure is that it only supports incremental update of discrete structures. There is a hidden assumption that the user only makes fairly coarse modifications to the structure.

Another problem with the Score structure is the increased access time due to the existence of multiple versions of properties. The real problem, if any, arises when *get_value* is called with an attribute that does not correspond to a property. The result is the Null value, but this can only be determined by traversing the entire property list, including all old versions of properties. A tradeoff between space and time can be made by making the following change in the implementation of *get_value*: Whenever an attribute for which no property exists is accessed, insert a new property with the attribute and the Null value at the head of the property list. The next time the same access is made, the property will be found immediately; thus, the average access time can be substantially improved in some cases.

10. Conclusions

The Score structure is designed to support a flexible and extensible music editor. It is essential that operations be separated from redisplay so that new operations can be added easily and so that redisplay can be performed once after a collection of low-level operations. The resulting structure performs these tasks well, and the mechanisms are cleanly integrated with a powerful Undo facility. The resulting system is quite flexible and efficient, given the demanding requirements.

The structure currently provides the foundation for the development of an advanced music score editing and typesetting system. A simple editor has been created to test and debug the structure. The editor demonstrates the use of versions, views, incremental display update, Undo operations, and the practicality of the Score data structure as a tool for building interactive, display-oriented editors.

An informal cooperative effort has been formed between the University of Washington, Brown University, and Carnegie Mellon University to continue development of a music editor using the present design as a foundation. Writing an editor is a big job, and I welcome the assistance of other interested individuals and organizations.

11. Acknowledgments

This design started as an extensive discussion with Dean Rubine and Paul McAvinney. The initial implementation was enabled by the loan of a workstation which is part of a campus computing system, Andrew, developed jointly by CMU and IBM. In the initial implementation, I wanted a flat structure without views and property inheritance. Rob Duisberg deserves credit for pointing out the flaws in my thinking. I would also like to thank the numerous people who have shown an interest in these ideas. Their questions and discussions have helped to improve not only the design but my presentation of it.

References

[1] Brinkman, Alexander P.
A Data Structure for Computer Analysis of Musical Scores.
In *Proceedings of the ICMC 1984*, pages 233-242. Computer Music Association, June, 1985.

[2] Buxton, W., R. Sniderman, W. Reeves, S. Patel, R. Baecker.
Evolution of the SSSP Score Editing Tools.
Computer Music Journal 3(4):14-25, 1979.

[3] Byrd, Donald Alvin.
Music Notation by Computer.
PhD thesis, Indiana University, 1984.

[4] Decker, Shawn L. and Gary S. Kendall.
A Unified Approach to the Editing of Time-Ordered Events.
In *Proceedings of the ICMC 1985*, pages 69-77. Computer Music Association, August, 1985.

[5] Minciacchi, Marco and Diego Minciacchi.
Music Editing and Graphics (MEG 1.00): A Personal Computer Based Operative System for Editing and Printing Musical Scores.
In *Proceedings of the ICMC 1984*, pages 257-272. Computer Music Association, June, 1985.

[6] Maxwell, John Turner III and Severo M. Ornstein.
Mockingbird: A Composer's Amanuensis.
Technical Report, Xerox PARC, 1983.