CENTER FOR COMPUTER RESEARCH IN MUSIC AND ACOUSTICS
DECEMBER 1990

Department of Music
Report No. STAN-M-69

MODELING MUSIC NOTATION: A THREE-DIMENSIONAL APPROACH

Glendon Ross Diener

CCRMA
DEPARTMENT OF MUSIC
Stanford University
Stanford, California 94305

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF MUSIC
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Abstract

This dissertation introduces a model for computer-assisted composition. The model describes a music notation system embedded in a visual programming environment. To the system, the images and conventions of particular notational styles take the form of data encapsulated as object specifications in an object-oriented programming language. Objects in the system display a specifiable graphics image. They are characterized as much by this image as they are by their state and behavior.

The model describes a consistent visual metaphor in which two-dimensional musical scores are seen as three-dimensional constructs made of piles of objects. Equivalent to tree structures, these piles specify hierarchical relationships defining temporal ordering among their component objects. Manipulation of these piles is equivalent to manipulation of their temporal ordering.

The model is claimed to be an excellent basis for conveniently expressing a variety of notational styles. A software system, dubbed *Nutation*, has been developed as an existence proof of this claim. Nutation is capable of running data sets defining a variety of notational styles, including a working subset of common music notation.

To the memory of my father

Acknowledgements

I would like to thank my advisors, Drs. Chowning, Mathews, Smith, and Dannenberg, for the interest they have shown in my work, and for giving generously of their time, their wisdom, and their enthusiasm.

This work benefited enormously from the help of my wife, Marcia Derr, in matters both technical and editorial, and in helping me see it through to the end.

Thanks to all my colleagues at CCRMA for making the last four and a half years an exciting and stimulating educational experience. In particular, I would like to thank David Jaffe, Doug Keislar, David Mellinger, and Atau Tanaka for their many helpful suggestions.

I would like to gratefully acknowledge the Social Sciences and Humanities Research Council of Canada for their support of my research at Stanford University.

Contents

List of Figures

Introduction

This dissertation is about music notation by computer. Most researchers in this area have, to date, concentrated on music printing and, in particular, the *music setting* problem. Their goal has been to transform instrumental gestures and hand-written manuscripts into high-quality printed output with the aid of a computer. By contrast, my research focuses on what will be called the *compositional* use of notation, and the use of the computer itself as a tool in the compositional process.

For centuries, the tool *par excellence* for working out musical ideas in the visual realm has been the pencil and manuscript paper. This dissertation uses contemporary technology to offer an alternative. In this scenario, the computer is used as a musical instrument: an instrument which is "played" through the direct, on-screen manipulation of music notation.

The fundamental contribution of this dissertation is the *three-dimensional, transparent glyph*(3TG) model of the notated score. This model has implications for both the data structure and the user interface of music notation programs. Much of the dissertation is concerned with developing these implications.

Chapter 1 is entitled "Notation, Composition, and Computers", and discusses what these three entities have to do with each other. The compositional use of notation is defined, and its relation to other research is explored. Chapter 2 contains the core ideas of the dissertation, for it is here that the 3TG model is introduced. The next two chapters are devoted to working out the implications of the model. In Chapter 3, an underlying data structure suggested by the model, called the *TTree*, is presented, while Chapter 4 discusses implications of the model for a hypothetical user interface. Chapter 5 describes an actual computer implementation based upon

the model. Finally, Chapter 6 summarizes the contributions of the dissertation and describes opportunities for further research. The logical flow of the text, then, has the following organization:

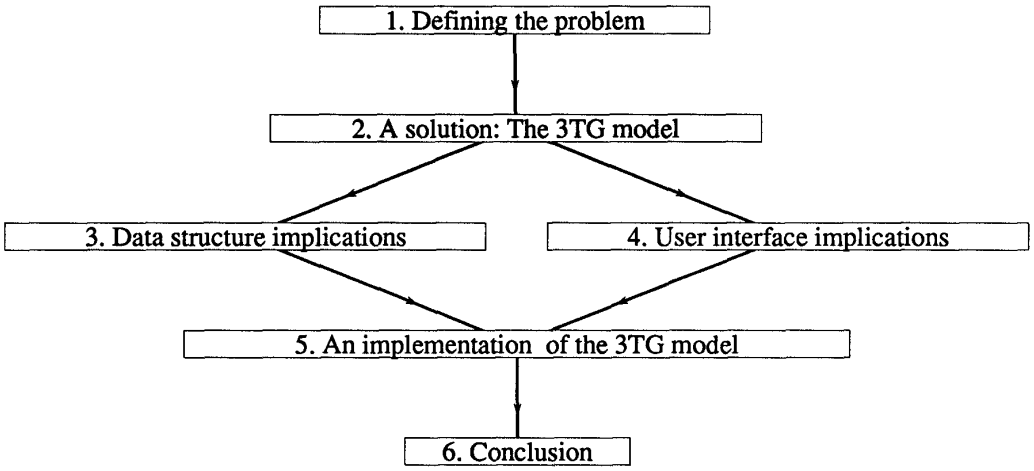


Figure 0.1: The logical flow of the text.

Much of the work on the TTree data structure was developed in [Die85]. Portions of the material in this dissertation originally appeared in [Die88], [Die89a], [Die89b], and [Die90].

visual symbolic expression, or *music notation system*, forms an integral part of their musical world.

Civilizations have been writing down music for at least 3700 years, and even a cursory account of the development of music notation could easily fill many volumes.[1] Throughout this long history, the only constant has been that of change itself. An examination of this history would reveal that music notation not only reflects changes in musical style, but also serves as a driving force behind the evolution of the very music being notated. In the words of Lewis Rowell:

> The history of musical scripts is at once a testament to human inventiveness and a demonstration of music's essential fragility. Each advance in our ability to notate music has had enormous consequences for the tradition we seek to represent; notation has been a destabilizing influence upon musical style. [Row82, page 35]

This evolutionary process, of course, is still going on, and today's computerized music notation programs are simply another step in the procession. Whether or not this step turns out to be a significant one may well depend on how developers of music notation software view their task. If their only goal is to automate the chores of music copying and printing, then their products, while they may eventually put a whole profession out of business (if they have not already done so), are unlikely to have any appreciable effect on the evolution of either notation or the music being notated. Computer technology, however, gives both developers and end users an enormous potential to experiment with original, unusual systems of notation. Unfortunately, the more innovative a notation system is, the less likely it is to gain acceptance within a musical culture steeped in its own traditions. As a preface to his analysis of numerous notational reforms proposed over the past three centuries, Gardner Read points out:

[1] "In 1974, after several years of decoding work, a Hurrian love-song dating from c. 1800 B.C. was at last transcribed and performed. This song caused a major revision of our ideas about the antiquity of musical notation, for it ante-dated the previously known oldest notation by more than a millennium." [Ras82, page ix]

Chapter 1

Notation, Composition, and Computers

1.1 Introduction

Musicians often take the subject of *music notation* by computer to be synonymous with *music printing* by computer. This is hardly surprising, as even the most cursory examination of music notation software past and present would serve to instill this belief. Music notation, however, has many uses in musical cultures, and music printing by no means serves all of them. This chapter identifies what will be called the *compositional* use of notation—a use which, up until now, has been all but neglected by developers of music notation software. An examination of a number of representative systems from the viewpoint of the notational uses they support underlines the need for a new approach to computerized notation systems, and helps to show how a compositional system would differ from one whose prime focus is music printing.

1.2 Notation

Many musical activities are carried out without the use of visual symbolic systems. Music is the art of sound in time, and improvisational styles and folk traditions flourish without any form of written notation. Yet for many cultures, some form of

> Almost without exception each proposal made its small ripple in the waters of Western music and thereafter quietly sank beneath those waters, some reforms barely surviving the lifetime of their creators. [Rea87b, page 1]

In the present study, the task of designing music notation software is viewed as that of adapting technology to facilitate that incremental change which constitutes the natural evolution of artistic style. No proposals for notational reform are to be found here. Instead, the goal is to create a design which can fully support the syntax of many widely understood notational traditions while allowing unlimited experimentation and expansion from these traditional bases. If this goal can be achieved, then the step to computerized notation systems for music may indeed turn out to be a significant one for our musical culture.

1.3 Composition

Music notation has served and will continue to serve a wide variety of purposes in musical cultures. This dissertation focuses on the *compositional* use of notation, and the way in which computers can support that usage. The term *composition*, when used in the context of notation, is intended in the sense of its Latin etymology *com-* + *ponere* = to put together. This implies an activity in which elements from a vocabulary of symbols are judiciously put together to create a representation of sound. Because these symbols are visual, they enable the composer to plan the emerging composition free from the linear, sequential constraint of time. The compositional use of notation covers a broad spectrum of musical activities, embracing not only the creation of original works of musical art, but such activities as preparing material for music education, arranging, orchestration, and the creation of musical scores intended as *objets d'art* in their own right.

The compositional use of notation, as defined here, contrasts most strongly with what might be termed its *archival* usage: an activity in which musics resulting from creative processes which have *already taken place* are translated into visual, symbolic representations to supplement human memory. This archival usage has been a

substantial contributer to the development of musical cultural and continues to be so today, even if its role has been somewhat diminished by the advent of recording technology.

Another use of notation which is distinct from both compositional and archival uses arises from the manner in which a great deal of music is performed. For many musical styles, music is a kind of partnership between composer and performer in which the composer specifies, through music notation, a set of instructions for the performer to realize instrumentally or vocally for an audience. Typically these instructions only partially determine the final shape of the work, leaving substantial interpretive latitude to the performer. In this scenario, notation functions as a form of communication between composer and performer, hence this usage could be characterized as "notation for communication." As Gardner Read comments:

> As the written language by which the composer communicates with an audience through a performer, it is a study in human relationship as well— to be judged by the effectiveness with which it communicates what to do, when to do it, and how to do it." [Rea87a]

Musical analysis also makes extensive use of notation. The graphic presentations of Schenkerian analyses, for example, testify to the power of purely visual, symbolic codes to convey information about musical structure. Like the archival use of notation, this analytic usage differs from the compositional usage in taking the results of creative processes which have *already taken place* as its starting point. Finally, the religious and metaphysical use of notation should not be left out, particularly if the notational practices of Oriental cultures are to be accounted for.[2]

There are, of course, many other uses of music notation, just as there are other ways to categorize these uses. Many of these categorization schemes are easily reducible to the categories presented here. To illustrate, consider the following list of usages proposed by Hugo Cole (referring primarily to Occidental cultures over the last one thousand years):

[2]The introduction to Kaufmann's study, [Kau67], contrasts the use of music notation in Oriental and Occidental cultures .

1. To allow the writer to invent new music, and to calculate effects in advance and at leisure.

2. To provide an exact timetable, so that independent parts may be closely co-ordinated.

3. To provide the performer with an artificial memory.

4. To describe the sounds of performed music for purposes of analysis or study (as, for instance in the notations of folk music collectors).
[Col74, page 9]

In terms of the categories of use just described, the first two of Cole's *uses* are clearly compositional, the third is archival, while the last is analytic. The point of this section, however, is not to single out what categorization scheme is the most appropriate for a given purpose. Rather, it is to identify the distinct usage of music notation in which the notation system itself serves as a creative tool helping musicians shape and form musical ideas. And as the next section shows, this *compositional* use of notation has been, up until now, all but overlooked by developers of music notation software.

1.4 Computers

This section turns its attention to music notation programs themselves in order to discover the underlying categories of notational usage to which they are adapted. The systems described here have been chosen for the diversity of approaches they represent rather than in an attempt to give a complete survey of the field. An excellent history of computerized music notation programs to 1984 can be found in Byrd's dissertation [Byr84]. Music notation software for the then new Apple Macintosh computer is discussed by Yavelow [Yav85]. Since that time, the flood of commercial software on the market would make any comprehensive attempt to describe it obsolete almost overnight. A good source of information about available systems can be found in the yearly publication *Computing in Musicology: A Directory of Research* published by

the *Center for Computer Assisted Research in the Humanities*. In their 1989 issue, they claim to have the names of about 75 music notation software developers on file. The issue briefly describes 55 recent systems which have shown a "demonstrated capability for handling classical music" [HSF89, page 44].

The earliest program dealing with music notation known to me is that of Hiller and Baker [JB65]. Input to this system was through a typewriter modified to include a music typeface and a paper tape punch and reader. Scores entered through the typewriter were simultaneously punched onto paper tape for subsequent input to an ILLIAC computer. The computer was programmed to do margin justification and part extraction on its internal representation of the score, then output the result back onto paper tape. This output was then fed back into the typewriter's tape reader to produce printed copy.

The notational usage implicit in this process is clearly not compositional. Indeed, it would be hard to imagine its creators conceiving of compositional usages given the limited user-interface technology available at that time. With the creative foresight of pioneers, however, they did consider the role of the computer in the compositional process, even if they rejected its possibility:

> Preparation of the Composer's Manuscript. This, of necessity, is a slow laborious hand operation since it is so intimately bound up with the compositional process itself. Composers' habits are so idiosyncratic and so variable that any real attempt to apply automation hardly seems feasible. [JB65, page 131]

The research of Hiller and Baker, then, was aimed at the archival and communicative uses of notation rather than the compositional. This is certainly not surprising, for the stated goal of the project was the automation of music printing. Theirs was the first of a long series of notation programs concentrating almost exclusively on music printing to such an extent that the two terms—music notation and music printing—came to be synonymous in many musician's minds whenever computer software for music was discussed. Music printing, however, is not really an accurate term for the role of the software in such programs, and more properly belongs to the process

of applying ink to paper. Instead, the term "music setting" is often found in the literature.

Many early music setting programs attempted to remove the human being from the process as much as possible. David Gomberg summed up the position thus:

> [Leland] Smith takes a different view and interlaces human and computer activities. By permitting explicit interaction he has quickly produced first-level results and some of what he has accomplished is quite encouraging. Nonetheless, not having excluded human activity from the start, he will likely find it difficult to do so later. A very basic reason for removing human activities from the system, is that these activities require knowledge and skill in many ways comparable to those of an engraver or autographer. They will probably not be lost entirely, but the number of persons involved in skilled crafts generally, and music setting particularly, is declining quite rapidly. [Gom77, page 69]

It has been over a decade since Gomberg wrote these words, yet no successful notation system which totally removes human activities from the music setting process has been forthcoming. Perhaps early researchers simply underestimated the complexity of the problem.[3] One researcher who devoted a good deal of time towards the analysis of this complexity was Donald Byrd, who wrote:

> Fully Automatic High-Quality Music Notation is...in general impossible without human-level intelligence. Since current work in artificial intelligence is far from this level, one must compromise, and three ways are possible: compromise in degree of automation, in quality of output, or in generality of input. [Byr86, page 147]

Byrd's own dissertation was entitled "Music Notation by Computer" [Byr84]. He makes it clear, however, that this title is somewhat misleading, and that he chose it

[3] In light of current thinking about the role of computers in society, one is tempted to say of Gomberg's approach that "not having *included* human activity from the start, he will likely find it difficult to do so later."

instead of the more accurate "Music Setting by Computer" in order to avoid even more confusion over the less-well-known phrase "music setting".

As witnessed by the preceding quotation, Leland Smith's MSS program [Smi73] included rather than excluded the user from the music setting process. Users were kept constantly in touch with the image of the score they were entering through immediate graphic feedback on the computer screen. Smith considered that the various editing tools provided by MSS were among its most important features. Thus, even if it was not an explicitly stated goal, the use of the system for composition was certainly a possibility. A testament to the feasibility of Smith's approach has been its durability and success: a microcomputer-based descendant of the original MSS is currently marketed under the name Score [Pas88], and is in use by numerous musicians and professional publishing companies today.

Mockingbird [Max81],[MO83], is the product of one of the most influential research projects in the music notation field ever attempted. This influence is all the more remarkable when one considers that neither the general public nor the computer music research community ever had the opportunity to use the system, for it was developed on experimental high-performance, proprietary hardware and software at Xerox corporation's Palo Alto Research Center (PARC). Rather than attempting to deal with a broad range of notational tasks, Mockingbird limited its functionality to the notation of standard piano music. Maxwell and Ornstein described the purpose of their system as follows:

> Mockingbird is a composer's amanuensis, a computer program designed to aid a composer with the capture, editing, and printing of musical ideas. The purpose of Mockingbird is not to invent new music or to suggest variations to the composer, but simply to aid him in recording his own ideas by speeding up the notating process. Mockingbird is not a publisher's aid, although it does print music, nor is it a performer's aid, although it can play; it is strictly focused on the composer's need for a powerful scribe. [MO83, page 1]

This quotation shows that although Mockingbird was conceived as a composer's tool,

it focused on archival uses of notation rather than compositional uses. Nevertheless, because the system included on-screen editing facilities which were both innovative and powerful for their time (it was probably the first music notation program to use direct manipulation[4]), compositional usage was certainly a possibility. As the following quote shows, however, the direct manipulation of notational material on the computer screen was seen as nothing more than a less powerful alternative to synthesizer keyboard entry:

> There are two ways to enter music in Mockingbird: by playing it on the synthesizer keyboard directly or by adding notes to the score manually. In the first, Mockingbird observes the synthesizer keystrokes and records their occurrence and duration. This allows the composer to enter music quickly and naturally in a form much like an old fashioned pianoroll. In the second, the composer adds notes by pointing at a position in the score with the mouse and commanding the editor to deposit a note of a specified value. This second method is less facile than the first, but is useful for making small corrections to the score. [Max81, page 11]

When the Apple Macintosh computer made its debut, it brought much of the user interface technology which had made Mockingbird possible into the hands of the general public. So rapid was the machine's impact on the music community that it prompted Christopher Yavelow to write:

> In eighteen short months, the Apple Macintosh computer has revolutionized the way "the rest of us" interact with computers, a fact that has surely not escaped the attention of most of our readers. The impact of the Macintosh upon the computer music world is just beginning to be felt, but already the ramifications are proving to be profound. [Yav85, page 52]

In 1985, Yavelow was able to review seven Macintosh-based software packages for music which were commercially available at that time (the number had grown to almost forty just a year later). A good deal of this software featured on-screen music

[4]Direct manipulation systems are discussed in Chapter 4.

notation of one form or another, and much of it permitted real-time playback through the machine's built-in eight-bit sound synthesis capabilities. One program in particular, "Professional Composer" [Mar86], came close to providing the kind of on-screen manipulation and flexibility needed to rival the pencil and manuscript paper as compositional tools. Nevertheless, the program was first and foremost a printing program, for it made no provisions for user extensibility or anything beyond the most standard aspects of common music notation. Yavelow summed it up nicely when defending the program against criticism of its playback facilities:

> "People who criticize the program for not providing more flexible playback do not have the remotest idea what the program is all about, that is, editing and printing music and not playing back music" [Yav85, page 64].

1.5 A Computerized Notation System for Composition

Among the notation systems discussed here, only rarely has the compositional use of notation been a possibility, and never was it found to be among the developer's stated goals. Why have developers overlooked this important aspect of music notation, and concentrated so much of their efforts on music printing? One reason is surely need. After all, the pencil and paper have been the tools *par excellence* for the compositional use of notation for centuries, and they are both cheap and abundant. High-quality music printing, however, before music printing programs became readily available, was a very expensive and time-consuming process. Software developers may well have concentrated their efforts in this area because this is where they perceived the greatest need for development. Another reason concerns the relatively large computer resources necessary for the task. Any program dedicated to the compositional use of notation would likely differ from the systems described previously in the following ways:

- Since on-screen entry and manipulation would be the primary, if not the only means of interacting with the system, its user interface would have to be exceptionally sophisticated.
- Musicians often compose with an instrument at hand in order to provide them with immediate aural feedback of their compositional ideas. If they are to be fully served by a computerized notation system for composition, that system ought to provide immediate, high-quality audio playback of their notational inventions.
- Pencil and paper are unlimited with respect to number of symbols, their size, placement, musical meaning, and so on. A computerized notation system for composition would have to be similarly unlimited, and provide for full user customization and extensibility.

Until very recently, machines powerful enough to support the graphics and real-time sound synthesis required for such a system have not been widely available. I believe, however, that the time has now arrived, and a computerized notation system for composition is a possibility. This dissertation is devoted to the design of just such a system. Of course, the compositional, archival, and communicative uses of notation are by no means mutually exclusive realms, and the compositional system which will be presented here could well be embedded in music printing programs, if only to facilitate minor editing tasks. This dissertation, however, focuses exclusively on compositional uses of notation. In short, it focuses on providing a framework for working out musical ideas in the visual realm.

1.6 Summary

This chapter has identified the need for a new direction in computerized music notation systems. The uses of notation in musical cultures were discussed, with emphasis on the evolutionary nature of notation's historical development. Several categories of notational usage were identified. Among these uses, the *compositional* use seems to have been all but overlooked by software developers. A number of representative

notation systems were analyzed in support of this claim. The feasibility of building a computerized music notation system for composition using contemporary technology was discussed. In the next chapter, a model for building such a system will be presented.

Chapter 2

Three-Dimensional, Transparent Glyphs

2.1 Introduction

This chapter introduces the three-dimensional, transparent glyph (3TG) model of computerized notation programs for composition. The model describes a consistent visual metaphor: two-dimensional musical scores are modeled as three-dimensional constructs made of piles of objects. Equivalent to tree structures, these piles specify hierarchical relationships defining both structure and temporal ordering among their component objects. The model provides a means for conveniently representing a wide variety of notational styles.

2.2 Models

Before describing the 3TG model itself, it will be useful to focus on the use of the term *model*, and on the motivation behind describing a system for the compositional use of notation in this way. Minsky has defined a model as "any structure that a person can use to simulate or anticipate the behavior of something else" [Min86, page 330]. Thomas Green tells us that "One structure models another...when the elements of the two structures are different but the relations between the elements are substantially

the same–although usually, the model contains only a carefully chosen subset of the relations in the target structure" [Gre90, page 5].

This chapter will introduce a wholly conceptual, *mental* design model of a music notation system. This model functions as a conceptual bridge between a source structure: music notation, and a target structure: a computer system for music notation. Researchers have long recognized that mental models play a vital role in helping users accomplish their goals when working with the computer systems these models describe. At the same time, they allow systems to be described completely abstractly, free of reference to any particular computer hardware or software.

In any computer system involving user interaction, it is possible to identify a number of potentially distinct mental models. Consider, for example, the user's conceptual model: the conceptualization that the user holds about the workings of the system. This conceptualization generally has little if anything to do with how the computer *actually* works, and has been called the *user illusion* by Alan Kay: "..the simplified myth everyone builds to explain (and make guesses about) the system's actions and what should be done next" [Kay84]. From the psychological viewpoint, Yvonne Waern distinguishes two fundamental ways in which the user forms this mental model [Wae90]. In the "bottom-up" approach, the model is formed solely on the basis of direct experience with the system. By contrast, in the "top-down" approach, the model is constructed on the basis of prior knowledge of similar tasks or systems. Waern tell us that "Most modern psychological theories of learning build on the assumption that learning mainly uses the top-down approach" [Wae90, page 75].

Another identifiable mental model distinct from the user's conceptual model is the design model: the designer's conceptualization of the system as implemented in the system itself. Common sense would seem to indicate that the design model should be as congruent as possible with the hypothesized conceptual model of the uninitiated user. In particular, a top-down learning strategy can be greatly facilitated if the design model is based on some familiar aspect of the user's knowledge and experience. The design model must be constructed with great care, however, for subtle differences between the designer's and the user's knowledge and experience can actually interfere with the user's success in accomplishing the goals of the system.

Ideally, the design model will include functionality unavailable in the source structure itself. To illustrate, this chapter is being typed using software whose interface implements a model of a desktop strewn with papers. Just like real papers, these "soft" pages can be moved around, piled, thrown away, and so on. Unlike real papers, however, the papers on this desktop display text which can be reformatted, spell-checked, scrolled, etc., all at the touch of a button. Without such extended functionality, there would be little point in using a computer in the first place.

This chapter presents the design model for a music notation system based on a familiar aspect of most users' reality: piles of rectangular objects of uniform thickness. For the most part, these objects behave just as they do in the physical world, and thus provide users with the opportunity to build their own conceptual models of the system using a top-down approach. At the same time, the model describes functionality which goes far beyond that of the source structure—music notation—which it models.

2.3 Three-dimensional, Transparent Glyphs

The focus of this chapter, and indeed the focus of this entire dissertation, is the *three-dimensional, transparent glyph* (3TG) model of the notated score. In this model, visual elements of musical scores are represented as special objects called glyphs. These glyphs present a two-dimensional rectangular surface displaying an image. Any image at all can be drawn in a glyph. Typically, glyphs will display such items as notes, clefs, staves, and other familiar symbols of music notation, but the choice of image is totally at the discretion of the user. Furthermore, a glyph might just as well display no image at all. The only requirement is that the image be drawn within a two-dimensional, rectangular area. Figure 2.1 shows five glyphs with the perimeters of their rectangular areas drawn in dotted lines.

Typically, the rectangular surface on which a glyph's image is drawn will be completely transparent, and only the image itself will be visible, as shown in Figure 2.2. This allows complex images to be built up by superimposing several glyphs on top of each other. This transparency, however, is also at the discretion of the user, and a glyph can be given any desired degree of transparency.

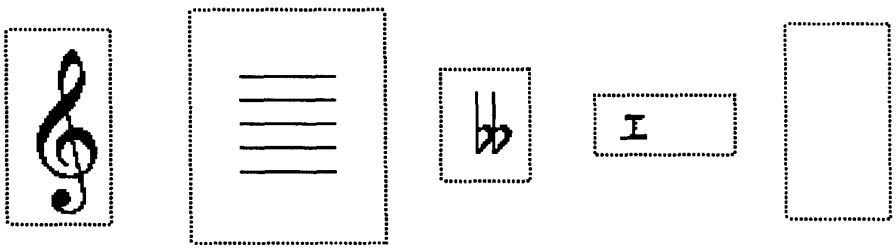


Figure 2.1: Five glyphs. The perimeter of each glyph's rectangular area is indicated by dotted lines which are not part of the glyph itself. Notice that this area may be larger than the image shown in the glyph, and that a glyph may have no image at all.

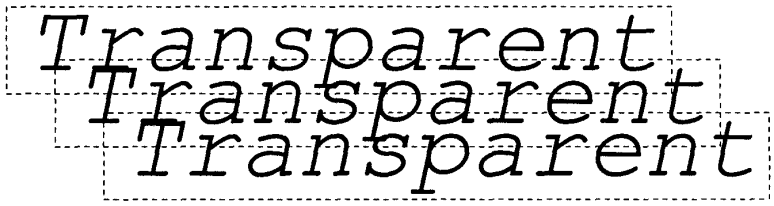


Figure 2.2: Three glyphs superimposed to show their transparency. As in the previous figure, the perimeter of each glyph's rectangular area is indicated by dotted lines which are not actually a part of the glyph.

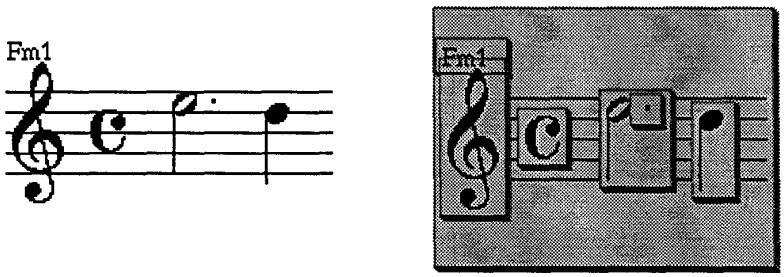


Figure 2.3: Two views of a score fragment. In the rightmost view, the glyphs have been given thickness and made opaque in order to emphasize their three-dimensional structure.

Although glyphs present a two-dimensional surface for drawing, conceptually they are three-dimensional boxes of unit thickness, and may be piled on top of or alongside of each other in a virtual three-dimensional space. It is this three-dimensional piling behavior that makes glyphs truly distinct from the graphical objects found in many computerized drawing programs. Since all glyphs have the same (unit) thickness, the three-dimensional space they occupy might more properly be referred to as two-and-a-half-dimensional, or two continuous dimensions and one discrete one. In the interests of avoiding the jargonistic phrase *two-and-a-half-dimensional transparent glyphs*, however, the three-dimensional designation will be used here. Figure 2.3 shows a portion of a common music notation score alongside a representation in which its glyphs are made opaque and given thickness in order to emphasize their three-dimensional structure.

To recapitulate, the 3TG model presents a world populated by piles of rectangular glyphs of unit thickness. Each glyph may be likened to a rectangular sheet of plastic transparency whose surface can contain drawing. When piled on top of each other, these glyphs allow complex images to be built up without ever losing the structural identity of the individual glyphs which make up the total image. The following sections elaborate upon the consequences of adopting this model for a music notation system.

2.4 Hierarchies

The designers of the Mockingbird system discussed in Chapter 1 opted for a data structure consisting of a flat, one-dimensional sequence of events. In their words:

> Our first design was a structured, hierarchical data structure that closely matched the formal structure we saw in music. But we ran into numerous problems with it because it didn't match the needs of an editor. After much discussion, we settled on an unstructured, sequential data structure. This surprised us, because in the beginning we had thought that the hierarchical design was the obvious choice. However, experience has convinced us that the sequential design is vastly superior.[MO83, page 15]

Why is it that Maxwell and Ornstein found a hierarchical data structure to be so unsatisfactory? Perhaps it is in their notion of hierarchy itself:

> There are many possible hierarchical data structures that might be used to represent music. We use the term loosely to describe a class of data structures that implicitly incorporate musical structure in the design. Thus one might imagine a data structure that had a separate part for each measure or for each voice. A "sequential" data structure, on the other hand, is simply a sequence of undifferentiated entities. No attempt is made to incorporate musical structure into the design. Instead, it is up to the algorithms to determine the structure from the entities.[MO83, page 16]

Unfortunately, they do not tell us why a hierarchical data structure has to implicitly incorporate any particular musical structure. In direct contrast to their findings, a major theme of this dissertation is that hierarchies are ideal data structures for music notation programs, provided they are not limited to some arbitrary number of hierarchical levels or set of shapes, and provided they are able to define temporal ordering on the nodes which constitute them. One hierarchical structure which fulfills both these provisos is the TTree,[1] which will be analyzed in detail in the next

[1]Pronounced tee-tree. The leading "T" stands for "temporal" or "time".

chapter. The binary equivalent of an ordered tree, the TTree associates time delays with each of its right-going arcs. Both the TTree and its multi-branching equivalent are unlimited as to hierarchical levels and shapes, but the TTree has the added bonus of providing a very efficient and convenient temporal ordering on its nodes (whence the initial letter T). This convenience results from the fact that sibling nodes of the ordered tree appear as a *list of lists* in the TTree. Thus time can be thought of as flowing *along* the branches of the tree, rather than somehow *between* them.

The primary motivation behind the three-dimensional piling behavior of glyphs is the expression of just such a hierarchy, for the glyph pile itself is simply the visual analog of a tree structure, with the base of the pile corresponding to the root of the tree. From the user's point of view, piling one glyph onto another is equivalent to attaching that glyph as the descendant of the other in a tree structure. To illustrate, referring to the fragment of common music notation shown in Figure 2.3, the root of the tree would be a glyph corresponding to the paper this score fragment is printed upon. The first generation consists of the glyph displaying the staff. The second generation includes the glyphs displaying the the instrument's name (in this case, "Fm1"), the treble clef, the time signature, and the half and quarter notes. Finally, the third generation would consist solely of the dot, attached in the tree as a direct descendant of the half note. Note that although the borders of the glyphs displaying the instrument and the treble clef overlap, neither is a descendant of the other, for they are siblings occupying the second generation of the tree.

It is interesting to compare this means of expressing hierarchical relationships by piling objects on top of each other to a closely related concept which has been in use in graphic systems since at least as far back as 1975: the idea of putting objects *inside* of each other, or *physical containment*.[2] Although containment also expresses hierarchical relationships, the piling model presented here is much more akin to the way many musicians are accustomed to thinking about notation. We are taught, for example, to put staves *on* pages, to put notes *on* staves, to put dots *on* notes, and so on. Whether consciously or not, musicians have been building up such visual hierarchies for centuries, hence the 3TG model is itself a very close match to the

[2]See, for example, David Canfield Smith's visual programming language *Pygmalion*[Smi75].

source structure—music notation—which it models.

2.5 Structural Organization Versus Graphic Generality

Striking the delicate balance between *structural organization* on the one hand and *graphical generality* on the other is a major issue in the design of music notation systems. This problem, described by Donald Byrd as the "fundamental tradeoff" between *semantics* and *graphics* [Byr86, page 145], is readily understood by imagining musical versions of the draw and paint programs available on many small computers. A musical draw program could facilitate high-level editing and performance operations by means of the data structure analogs of notes, staves, parts, and the like, but as a consequence would limit its visual universe to some finite collection of pre-defined symbols. By contrast, a musical paint program, by imposing no further organization on its data than that of a two-dimensional array of pixels, would gain graphical generality at the expense of its ability to perform musically meaningful operations on that data.

The 3TG model, however, imposes no restrictions whatever on the image which a glyph can display. At the same time, arbitrarily deep hierarchical data structures can be built up without sacrificing any of this graphic potential. It is as if one were able to run many paint programs at the same time, and arrange the windows from these programs into arbitrary hierarchical structures. Structural organization and graphic generality, then, are by no means incompatible objectives, and the 3TG model provides a synthesis of the two perspectives.

2.6 Glyphs as Objects

The importance of user "programmability", even in the most flexible of interactive systems, is a theme which has been echoed over and over by software visionaries over the years. In the words of Alan Kay:

> Does this mean that what might be called a driver-education approach to computer literacy is all most people will ever need—that one need only learn how to "drive" applications programs and need never learn to program? Certainly not. Users must be able to tailor a system to their wants. Anything less would be as absurd as requiring essays to be formed out of paragraphs that have already been written." [Kay84, page 57]

In the field of music composition by computer, many musicians have come to exactly this same conclusion. As Bill Schottstaedt writes, "...our experience demonstrates that composers find a programming language far more congenial than a data-entry system when trying to write computer music" [Sch83, page 20]. Yet a general purpose programming language is not enough: some means must be found for focusing the musician's energies more on the task of composition than on the task of programming. In the 3TG model, this can be accomplished by implementing glyphs as objects in a visually-based, object-oriented programming language.

Most object-oriented programming languages provide some facility for sharing code among different objects.[3] In a *single-inheritance, class-based* object-oriented system, the *class* of an object contains *methods* (procedures) which determine how any object which is an *instance* (member) of that class will behave in response to the corresponding *messages* (procedure calls). Classes themselves can be arranged into *inheritance hierarchies*. When an object receives a message for which its class has no corresponding method, that method can be searched for one level up the inheritance hierarchy in the object's *superclass*, then in the object's super-superclass, and so on, until the corresponding method is found. If no corresponding message is found, an error condition can be raised. This process is called *inheritance*, and it allows methods which are common to a whole group of classes to be "factored out" and pushed as far up the inheritance hierarchy as necessary, eliminating the need to repeat that code for each of the classes involved.

When glyphs are implemented as objects, the root class of the inheritance hierarchy (hereafter called class *Glyph*) will include the necessary functionality to support their piling behavior and the display of their images. All glyphs, no matter what

[3] See [Cox86] for an introduction to basic concepts of object-oriented programming.

their purpose, will inherit this functionality. The musician wishing to define some specific behavior for a class of glyphs need only specify how that class differs from its superclass. A *Note* class, for example, might contain methods which determine the pitch of its instances. *Subclasses* of the *Note* class, such as *QuarterNote* or *HalfNote*, would be responsible for determining their instances' durations.

As objects, glyphs can communicate with each other by sending messages. And because a tree structure provides a well-defined path between all of its nodes, the hierarchy, which glyph piles define, can function as a network for channeling these messages from glyph to glyph. Note that this tree structure is a hierarchy of *instances*, not to be confused with the *inheritance* hierarchy described earlier, which is composed of classes, rather than instances of classes.[4]

This message-channeling scheme immediately suggests a method for causing glyph piles to perform themselves as music. To illustrate, assume we have a pile of glyphs representing a fragment of common music notation such as that shown in Figure 2.3. When the root of the tree (pile) is sent a *play* message, this message eventually trickles down through the tree until it bottoms out at the leaves (generally, some kind of "note" glyph). Along the way, the flow of the message can be controlled by the glyphs it passes through. A "paper" glyph, for example, might relay the message to its subglyphs (usually "staff" glyphs) successively from the top of the page to the bottom. Each "staff" subglyph, upon receipt of the "play" message, responds in turn by relaying the message from its leftmost subglyph to its rightmost. These subglyphs, generally "note" glyphs, respond by instructing lower-level synthesis functions to generate sound samples. The mechanism ensures that the score is performed in the correct top-to-bottom, left-to-right order, and allows for the accumulation of such global state as current key and time signature, dynamic level, etc., in a manner analogous to the way in which a human performer might read the score.

Note that the model's hierarchical scheme is by no means fixed. The number of hierarchical levels and their interrelationships constitute data which can be crafted, or reprogrammed, by the musician as desired. The model prescribes a convenient

[4]This class-instance distinction is blurred in certain object-oriented programming languages. In Smalltalk-80, for example, classes are themselves objects, and as such are instances of other classes, called *metaclasses*.

framework for building such structures and for associating them with graphic representations. Given the generality of this architecture, it is easy to extend the system to embrace non-standard notational schemes by defining new glyph classes. In fact, the model gives musicians the power to define their own notational worlds: worlds which may bear little or no resemblance to common notational practice.

2.7 Glyph Sets

If the 3TG metaphor is implemented according to the object-oriented scheme just outlined, then individual notational systems can be represented as sets of subclasses of class *Glyph*. Implementations of the model would be capable of supporting any number of notational worlds, any or all of which may potentially coexist within the same score. Chapter 5 of this dissertation describes just such an implementation: it presents a program, dubbed *Nutation*, and three distinct sets of Glyph subclasses implementing a piano-roll notation, common music notation, and a tablature notation system for voice and the samisen (a lute-like instrument) from the island of Okinawa.

The world of music notation by computer would be very different if programs such as this were to reach maturity and become standard. Musicians with the programming skills to create new glyph sets would be able to design their own notational systems, perhaps using the computer as an intelligent performer for scores written in those systems. Archives of these glyph sets could be kept so that they could be shared with others, including those who preferred not to program the system itself. In such a world, the evolutionary nature of music notation and its relationship to composition, as discussed in Chapter 1, could be greatly enhanced.

2.8 Summary

This chapter has introduced the central idea of this paper: the three-dimensional, transparent glyph (3TG) model of computerized music notation programs. By presenting the concept as a model, reference to any particular hardware or software implementation was avoided. At the same time, such a description forms a mental

model of great benefit to the potential user. After claiming the superiority of hierarchical data structures over purely flat, one-dimensional structures for music notation programs, the hierarchical data structure implicit in the glyph's three-dimensional piling behavior was identified. The 3TG model was shown to provide a synthesis between notation programs' need for structural organization on the one hand and graphic generality on the other. The importance of user programmability for computerized compositional systems was stressed, and an outline of a single-inheritance, class-based object-oriented programming language supporting the 3TG model was presented. Finally, the concept of glyph sets and the implication they might have for the future of the compositional use of notation was discussed.

Chapter 3

The TTree

3.1 Introduction

The three-dimensional, transparent glyph (3TG) model introduced in Chapter 2 strongly suggests some form of tree structure for its implementation. There are, of course, many ways to implement trees, each exhibiting characteristic advantages and disadvantages for a given problem domain. Further implementation choices result from adopting the object-oriented programming paradigm, where the principle of *data abstraction* groups data together with the operations to be carried out upon that data.

This chapter introduces the concept of the *TTree*: a simple and elegant structure uniting temporal organization with the hierarchical representation of musical scores in an object-oriented system. The TTree is shown to be an ideal choice for implementing the 3TG model.

Throughout the chapter, a basic knowledge of computer data structures and object-oriented programming concepts is assumed. Wherever possible, formal definitions have been relegated to the footnotes.

3.2 Design Goals

Before informed decisions about data structure design can be made, a set of relevant design goals must be formulated. The six "ground rules" for graphic software system design suggested by Newman and Sproull—simplicity, consistency, completeness, robustness, performance, and economy [NS73, page 80]—form a convenient starting point. These goals are no less applicable to software combining music with graphics, although the added computational load of audio processing may well make system performance the dominant concern. For music composition and synthesis (MCS), moreover, with its emphasis on creativity, the importance of system performance is well-known. As Rodet and Cointe point out:

> Why is MCS one of the most interesting fields for testing advanced technology for computing? In MCS, perhaps even more than in artificial intelligence, programs have to be tested, modified, and rewritten very often. This modification must be effected quickly and easily; otherwise, the continuity between musical ideas and results gets lost.[RC84, page 32].

System performance, then, must not disrupt the continuity between musical ideas and results. Any delay between an operation carried out in the visual realm and the audition of that operation's result must be kept to a minimum. This suggests that it is better to provide a single, unified data structure for both playing and displaying than to provide separate, quasi-independent structures requiring complex conversion functions between the two domains. With a single structure, operations in the visual realm are, effectively, operations in the auditory realm. At the same time, with only a single representation, it is impossible for audio and visual representations to become inconsistent with one another.

An integrated data structure, however, must also satisfy the goal of completeness. For music composition software, this means simultaneously meeting the needs of graphic score representation and real-time audio playback. More specifically, for an effective implementation of the 3TG model, a structure must be found which integrates temporal aspects of score performance with the model's graphical, tree-structured representation. This is the role of the TTree. Before describing this

structure, however, a few well-known characteristics of trees will be reviewed in order to establish a working vocabulary.

3.3 Trees

There are many ways to implement trees in a computer.[1] A fundamental problem confronting all implementations is the fact that an arbitrary number of subtrees may be attached to any given node. Clearly, the task of allocating storage for these subtrees would be greatly simplified if this number were constant. Applications, however, are seldom so obliging. Fortunately, due to what has been called the *natural correspondence* between forests and binary trees [Knu73, page 333], it is always possible to represent a multi-branching tree by its corresponding binary one.[2] Informally, a binary tree is formed from its corresponding tree by linking together the subnodes of each root, then deleting all but the leftmost root-subnode branch, as shown in Figure 3.1.

Because the binary correspondent never has more than two edges emanating from any of its nodes, it is much more convenient to implement than its multi-branching

[1] Stating the well-known recursive definition, the term *tree* designates a finite set of one or more nodes T such that

1. One particular node, $root(T) \in T$, is designated as the *root* of T ;
2. The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$. Each of these sets is in turn a tree, and together they are called the *subtrees* of the root.

Further, as with most computer applications, the order of the subtrees $T_1, \ldots, T_n$ is taken as significant. In graph theory, then, this usage would be referred to as a *finite, labeled, rooted* and *ordered* tree.

[2] A *binary tree* B can be defined formally as a finite set of nodes which is either empty or contains a root and two disjoint binary trees B_1 and B_2, called its left subtree and right subtree respectively. Whereas the ordering of subtrees in a tree is significant, it is the *position* of the subtrees which is significant for the binary tree. It makes no sense, for example, to have a tree with a 2nd subtree and not a 1st subtree, though it is perfectly acceptable to have a binary tree with a right subtree but no left subtree. Formally, the *binary correspondent* of a *forest* of trees $F = (T_1, \ldots, T_n)$, denoted $B(F)$, is defined as:

1. if $n = 0$ empty, else
2. $root(T_1)$ is its root; $B_1 = B(T_{1_1}, T_{1_2}, \ldots, T_{1_m})$ is its left subtree, where $T_{1_1}, T_{1_2}, \ldots, T_{1_m}$ are the subtrees of $root(T_1)$, and $B_2 = B(T_2, \ldots, T_n)$ is its right subtree.

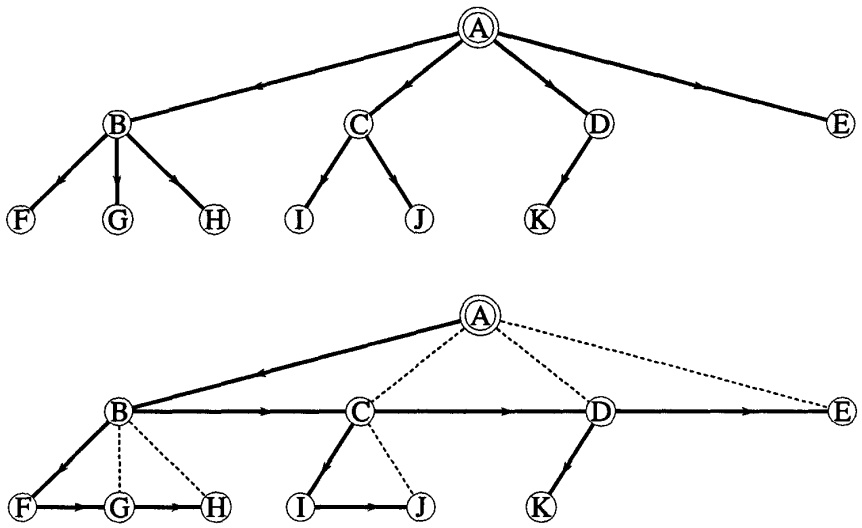


Figure 3.1: An ordered tree and its corresponding binary tree. The root node of each tree is indicated by a double circle. Links removed from the tree to form its binary correspondent have been drawn with dotted lines to show the underlying process involved.

counterpart. For this reason, computer implementations of trees almost always make use of the corresponding binary representation. In this dissertation, the term "binary tree" will be used instead of the longer "binary correspondent of a tree," while the term "ordered tree" will designate the corresponding tree.[3]

From the viewpoint of memory usage, the most parsimonious schemes for representing binary trees make use of sequential memory locations for node storage. These representations, however, can be very difficult to alter after they have been set up, and may be unsuited to applications where interactive restructuring of the tree is the rule rather than the exception, as with the 3TG model. Accordingly, a linked representation, with the left and right subtrees of a node represented by pointers (memory addresses) will be adopted here.

[3]The binary correspondent of a *tree*, as opposed to a *forest of trees*, follows directly by letting $F = (T_1)$ in Footnote 2. In this case, note that the right subtree of $root(T_1)$ will always be empty.

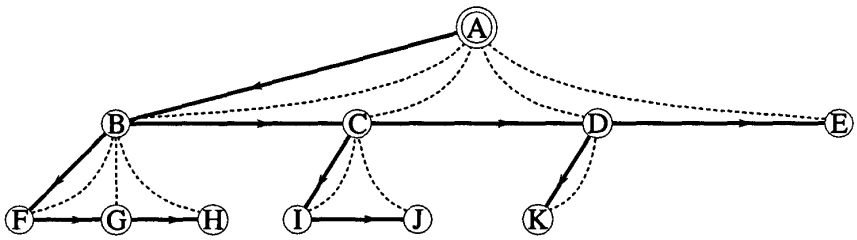


Figure 3.2: Ancestor links. This binary tree depicts ancestor links as curved, dotted lines.

Because pointers are unidirectional, it can be very inconvenient for nodes to refer to their ancestors. For this reason, in applications where "upward" references within the tree will be required, a third pointer is often used. These "ancestor" pointers, which link nodes with their direct ancestors, are depicted as curved, dotted lines in Figure 3.2. Note that such pointers refer to each object's ancestor in the corresponding ordered tree, which is not necessarily the object's binary tree ancestor. To illustrate, in Figure 3.2, the node labeled "H" has, as its ancestor, the node labeled "B", despite the fact that its binary tree ancestor is actually "G".

Traversal is the name given to the task of visiting the nodes of the tree in some prescribed order such that each node is visited exactly once. Traversal algorithms are extremely important for implementations of the 3TG model, because they are the means by which graphic information is accessed from the structure for communication to the display hardware, and by which audio information is accessed for communication to the synthesis hardware.

There are three common methods of traversing a binary tree. All are described recursively according to the formula "traverse the left subtree, then traverse the right subtree." The methods differ as to when the root of each subtree is visited: *preorder* traversal visits the root before traversing the left subtree, *inorder* visits the root after traversing the left subtree, while *postorder* visits the root only after the left and right subtrees have been traversed. Referring to Figure 3.2, preorder visits the nodes in the order ABFGHCIJDKE, inorder yields FGHBIJCKDEA, while postorder yields

HGFJIKEDCBA.

3.4 Structural Levels

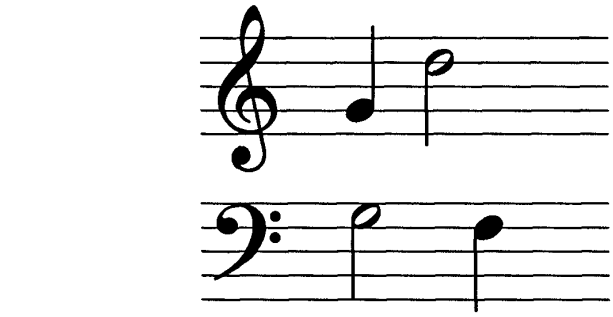
Readers might argue that the binary tree is a useful implementation strategy, but has little value beyond the realm of computational efficiency. In many situations, however, the binary representation is actually closer to intuitive notions about hierarchy than is its ordered counterpart. For a given application, if the connections among siblings have equal or greater conceptual primacy than the parent-sibling connection, then the binary tree is likely a superior model in terms of both conception and implementation.

Such situations abound in music. Schenkerian theory [FG82], for example, places much more emphasis on the notion of "structural level" than on the relationship between the elements of one structural level and another. It is as though the tree structure were viewed in cross-section, emphasizing relationships between branches over relationships along branches. The binary tree, of course, models this situation precisely, for siblings are directly connected in a list, in contrast to the indirect connection, through their common ancestor, provided by the ordered tree.

Another example is afforded by the temporal structure of common music notation. In the upper tree of Figure 3.3, notational elements of a score fragment are represented as nodes in a tree. The tree's root is the paper the score is printed on; the next structural level contains the staves; finally, the clefs, notes, rests, and so on, form the bottom level. Now, whereas the pitch structure of the work is primarily determined by relationships *along* the branches, the temporal structure is determined by relationships *between* branches. Thus, in the lower, binary tree of Figure 3.3, time flows *along* the branches, rather than somehow *between* them, as with the ordered tree.

3.5 TTrees

Before a binary tree score representation can be performed by machine, its temporal structure must be made explicit. This is the role of the TTree, where the leading "T"



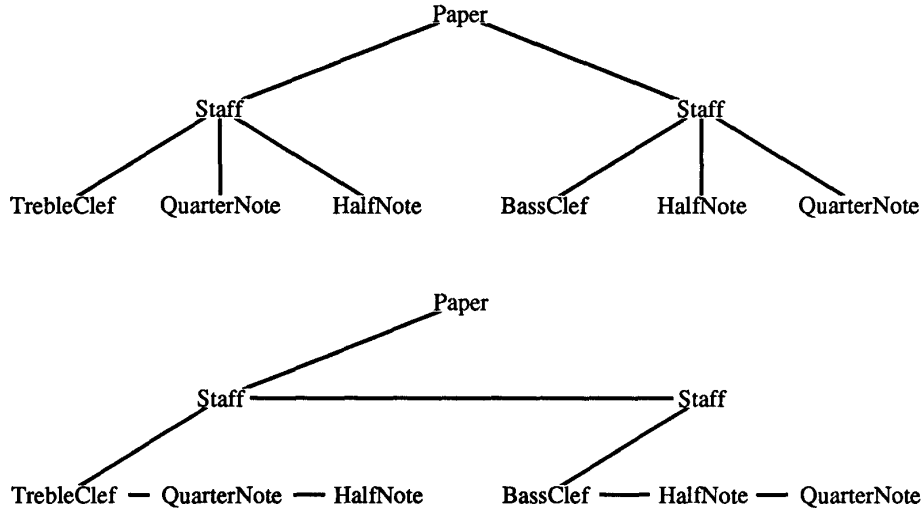


Figure 3.3: Ordered tree versus binary tree. This fragment of a common music notation score is represented first as an ordered tree and then by the corresponding binary tree. Braces, barlines, and time signatures have been omitted for simplicity.

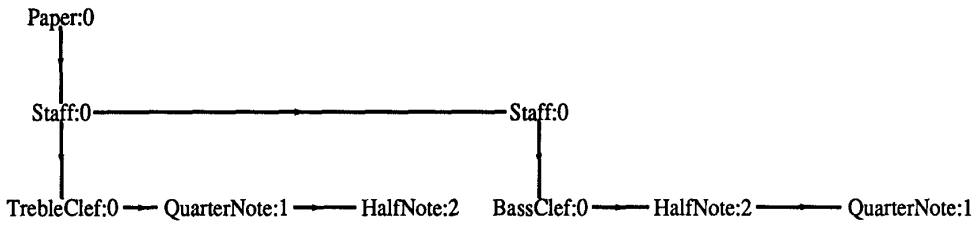


Figure 3.4: A TTree. The figure reproduces the binary tree of Figure 3.3, with horizontal arcs labeled by time delays, and with arcs entering each node from above drawn vertically.

stands for "temporal" or "time". A TTree is simply the *binary correspondent of an ordered tree, with time delays associated with its right-going arcs.*

In Figure 3.4, the binary tree of Figure 3.3 has been converted to a TTree by labeling its nodes with integers representing time delay in beats. At the same time, the graph has been shifted so that arcs entering each node from above are aligned vertically. Shifting the graph in this way emphasizes its time structure: nodes aligned along common vertical axes are temporally simultaneous. Note that the time delay associated with each node affects only horizontal relationships. Thus, referring to Figure 3.4, the node labeled *Paper* together with the nodes *Staff* and *TrebleClef* directly underneath it are temporally simultaneous. Moreover, because *TrebleClef* has a time delay of 0, the leftmost *QuarterNote* also occurs at the same time. This *QuarterNote's* time delay value, however, delays the occurrence of the leftmost *HalfNote* by one unit. Similarly, because the leftmost *Staff* has a time delay of 0, the rightmost *Staff*, together with *BassClef* and the rightmost *HalfNote* are simultaneous with *Paper*, while the rightmost *QuarterNote* is delayed by two units.

There is an interesting natural language interpretation of the TTree which can help to clarify the way in which it organizes time. In this interpretation, vertical arcs are referred to as "is" arcs, horizontal ones are referred to as "then" arcs, while delay values are referred to as "wait" values. To illustrate, Figure 3.5 shows the partially complete layout of a typical school fugue form as a TTree. The structure of the

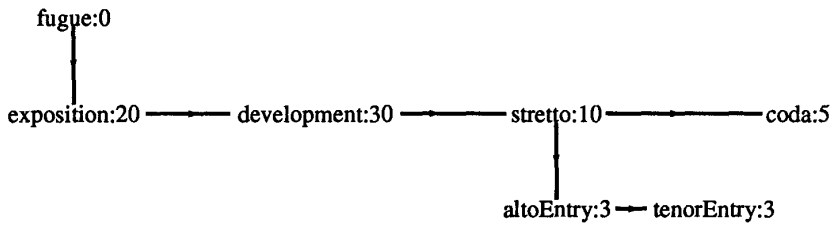


Figure 3.5: A partially complete school fugue form represented as a TTree.

fugue can be read directly from the tree. One can say "*fugue* is *exposition* wait 20 then *development* wait 30 then *stretto* wait 10 then *coda*", and "*stretto* is *altoEntry* wait 3 then *tenorEntry*". Note that the time delay associated with the node labeled *fugue* is redundant. The time delays associated with nodes *tenorEntry* and *coda* are similarly redundant, though they will become meaningful if further horizontal arcs are attached to them at some later stage.

It is important to note that a TTree's delay values have nothing whatsoever to do with duration. The *altoEntry* of Figure 3.5, for example, could last anywhere from milliseconds to millennia. If the *stretto* is a true musical stretto, then the duration of *altoEntry* must be greater than 3 units, though this information is in no way implied by the TTree, which limits its role to indicating the relative begin times of the nodes it is made of.

The glyphs of the 3TG model introduced in Chapter 2 map directly into the nodes of a TTree. TTrees are the key to implementing the model: they provide, in a single structure, both the temporal information needed for score performance and the hierarchical information needed to support the model's tree-structured graphical representation.

3.6 TTrees and Grammars

This section provides an analysis of TTrees in the context of formal language theory. The reader unfamiliar with the theory may safely skip this material without loss of continuity. The formalism adopted here is taken from Hopcroft and Ullman [HU79].

The relationship between formal grammars and data structures for computer music was first investigated by the Structured Sound Synthesis Project (SSSP) [BRBM77]. The purpose of the present description is twofold. First, it provides an alternate analysis of the structure within the framework of a well-defined mathematical formalism. Second, by showing the relationship between TTrees and grammars, a basis is formed for the development of automatic score generating programs. Though such development is beyond the scope of this dissertation, the possibilities are extremely compelling. As generative tools, for example, such programs become tools for algorithmic composition. Conversely, analytic tools could be built which use existing compiler technology to convert one-dimensional note list representations into graphic scores.[4]

Formal language theory shows how certain, possibly infinite, sets of strings (the *language* generated by a given grammar) can be derived from a single distinguished symbol (the *start symbol* of the grammar), through the successive application of a series of rules (the grammar's *rewriting* or *production* rules). For clarity, these derivations are often displayed as ordered trees. A derivation, however, could just as well be displayed as the binary equivalent of an ordered tree and, by extension, as a TTree, provided all of the production rules used in the derivation are of a certain form. To this end, define a T grammar G_T as any grammar of the following form:

[4]There is a fairly extensive body of research dealing with the use of grammars for composition and analysis. In the study [Die85], in-depth analyses of several representative projects are discussed.

$G_T = (V, T, P, S)$, where
V is a finite set of symbols called variables,
T is a finite set of symbols called terminals,
P is a finite set of production rules of the form:
$A \rightarrow X_1 X_2 \cdots X_m$, $m \geq 1$, $A, X_i \in V$
or $A \rightarrow a$, $A \in V, a \in T$,
$S \in V$ is the grammar's start symbol.

T grammars, then, are simply context-free grammars with the exception that their productions are restricted to a special normal form. Similar to Chomsky normal, the right-hand sides of all productions in a T grammar must contain either a single terminal symbol or one or more non-terminal symbols.[5]

A TTree can be viewed as a binary tree corresponding to a (non-binary) derivation tree in a T grammar. Thus, the application of a T grammar production of the form $A \Rightarrow X_1 X_2 \cdots X_i$ will have a vertical arc from A to X_1, X_1 will have a horizontal arc to X_2, X_2 a horizontal arc to X_3, and so on. Thus:

$$\begin{array}{l} A \\ \downarrow \\ X_1 \rightarrow X_2 \rightarrow X_3 \rightarrow \ldots \end{array}$$

Note that this procedure results in the *binary equivalent* of a derivation tree.

T grammars restrict their productions in order to permit this form of tree structure without violating certain conventions concerning derivation trees. Specifically, if productions of the form $A \rightarrow x_1 x_2 \cdots x_m, A \in V$ and $x_i \in T, m \geq 2$ were not prohibited, then terminals could appear as interior nodes, whereas they normally only occur in leaf positions. Observe that no restriction in the grammar's generative capacity is brought about by this restriction, for any context-free language not containing the null string is generated by a grammar of this form. In fact, the grammar might well be termed "semi-Chomsky normal," for the above observation is often proved as an

[5]In Chomsky normal form, either a single terminal symbol or exactly 2 non-terminal symbols may appear on the right-hand side of every production.

intermediate result when proving the generative equivalence of Chomsky normal form grammars and context-free grammars.[6]

The primary purpose of any TTree defined by a T grammar derivation is to give temporal meaning to the symbols constituting the tree. To this end, a numeric 'wait' value, $\mathcal{W}(X)$ is associated with every variable $X \in V$. The *time* of any symbol $\nu \in V \cup T$ in a TTree, relative to its root, is the sum of all values $\mathcal{W}(X)$ of the variables X having horizontal branches on the path from S (the start symbol) to ν. More formally, if x_1 and x_n are two symbols in a TTree such that x_1 is an ancestor of x_n, and if the path from x_1 to x_n is described by the sequence of symbols $x_1, x_2, \ldots, x_n$, then the time from x_1 to x_n is defined as follows:

$$\mathcal{T}(x_1, x_n) = \sum_{i=1}^{n-1} \mathcal{V}(x_i, x_{i+1}), \qquad \mathcal{V}(x, y) = \begin{cases} \mathcal{W}(x) & \text{if } x \text{ branches horizontally to } y \\ 0 & \text{otherwise} \end{cases}$$

To recapitulate, there is a convenient mapping between TTrees and formal grammars: TTrees can be seen as (binary) derivation trees corresponding to derivations in a T grammar. Although T grammars restrict their productions to a certain normal form, they are generatively equivalent to context-free grammars. Associating numeric "wait" values with the variables of a T grammar allows temporal information to be embedded in the derivation tree itself.

In a recent paper, I describe a compositional environment which parallels these formal language concepts precisely [Die89b]. By contrast, the TTree model presented in this chapter has no notion of disjoint sets of variables and terminals: it is as if all the nodes of the tree were members of the variable alphabet. The formal grammar correspondence can easily be established, however, by adding terminals t and productions $T \rightarrow t$ for all variables $T \in V$.

3.7 Message Channeling

In object-oriented systems, the nodes of binary trees may be conveniently represented as objects. Instance variables of these objects, when bound to other nodes in the tree,

[6] See, for example, [HU79, pp 92-93]

implement the arcs which define the tree's structure. An important characteristic of all tree structures is that there is exactly one simple path between any two nodes in a tree.[7] For an object-oriented system, where communication between objects is done by message-passing, this means that every object in a binary tree will have a unique, well-defined way of messaging every other object.[8] The tree becomes, in effect, a channel for inter-object communication.

In order to illustrate this point, some form of algorithmic language will be required. In this and subsequent examples, the syntax of Smalltalk-80 will be adopted.[9] Following the natural language interpretation introduced in Section 3.5, assume that every object in a binary tree has instance variables named *is* and *then* which are either nil (empty) or contain references to the respective vertically and horizontally situated objects in the tree. The corresponding messages *is* and *then* cause these references to be returned. Furthermore, assume that each object contains an instance variable *ancestor*, which it returns in response to the message *ancestor*. Thus, referring to Figure 3.5, *fugue* can send the message *aMessage* to *altoEntry* by executing "*self is then then is aMessage*", while *stretto* would communicate with *development* by executing "*self ancestor is then aMessage.*"[10]

In the object-oriented programming literature, structures such as these have been referred to as "instance hierarchies" as distinct from "class hierarchies." Class hierarchies, through the mechanism of inheritance, are the principle means by which many object-oriented languages allow objects to share behavior. Behavior sharing within instance hierarchies, however, can be a very powerful tool. In a TTree, such sharing can be accomplished by having every object which does not understand a

[7] In graph theory, a path is *simple* if all its nodes and edges are distinct.

[8] This paper uses the terms "message-passing" and "messaging" in the sense of Smalltalk-80 or Loops: "...a synchronous form of message-passing in which the sender must wait for a reply from the receiver before continuing." [Weg87, page 512]

[9] For the sake of precision, it is often better to adopt an actual programming language for describing algorithms rather than some subset of English. Smalltalk-80 [GR83], as one of the oldest and best-known object-oriented programming languages, has a simple and consistent syntax which is ideal for this purpose.

[10] Of course, if the structure contained explicit pointers backwards along the horizontal axis (i.e. true, "binary tree" ancestor pointers), this message could be simply "*self predecessor aMessage.*"

given message simply "delegate" that message to its ancestor.[11] Messages, then, will always travel upwards through the tree towards the root until they arrive at an object that can respond to them.

The implementation described in Chapter 5 makes extensive use of this form of delegation. To illustrate, consider the case of a "note" glyph trying to send the message *aMessage* to its "staff" glyph. Now, it may not be the case that notes sit directly on top of staves in a score: there could be, for example, several layers of "part" glyphs, "measure" glyphs, and so on, between the two. But if we arrange for staves, and staves alone, to return *self* in response to the message *findStaff*, then a note can simply execute "*self findStaff aMessage*". Since the objects lying between the note and the stave don't understand *findStaff*, they will pass it on successively up the ancestor chain until it reaches the staff, and the communication is accomplished.

Several useful categories for describing behavior-sharing mechanisms in object-oriented programming languages are introduced in *The Treaty of Orlando* [SLU89]. Adopting the terminology of this paper, the form of delegation presented here can be described as:

- Dynamic: sharing patterns are determined when an object actually receives a message.
- Explicit: the programmer explicitly directs the patterns of sharing by direct manipulation of glyphs through the 3TG model.
- Per object: idiosyncratic behavior can be attached to the individual object.

When merged with the static, implicit, and per group behavior-sharing mechanisms of class-based inheritance, the scheme provides a powerful programming system for building scores with the 3TG model.

[11] The term "delegation" refers to the process of determining behavior sharing strategies dynamically (during program execution). Various forms of inheritance and delegation are described in Wegner's article [Weg87], while Stein has argued for their equivalence [Ste87].

3.8 The TTree as Declarative Description

Like "object-oriented", the adjective "declarative" is often applied to programming languages and descriptions. Ullman defines a *declarative* language as:

> ...a language in which one can express what one wants, without explaining exactly how the desired result is to be computed. A language that is not declarative is *procedural*. "Declarative" and "procedural" are relative terms, but it is generally accepted that ordinary languages, like Pascal, C, Lisp, and the like, are procedural, with the term "declarative" used for languages that require less specificity regarding the required sequence of steps than do languages in this class. [Ull88, page 21]

Declarative languages attempt to liberate programmers from the necessity of specifying "how" results are obtained, allowing them to focus on "what" results they wish to obtain. Presumably, this makes the programming process easier and, all other things being equal, users will prefer declarative languages over procedural ones. Nevertheless, most music representation languages to date have tended towards procedural descriptions. This has led Dannenberg to remark:

> In spite of these precedents, my position is that algorithmic or procedural descriptions are a bad approach to the representation problem because of the difficulty of manipulating these descriptions.[Dan89b, page 73]

The message channeling capabilities of the TTree, described in Section 3.7, have a strong declarative flavor, for they largely free the programmer from having to specify the mechanism involved in passing messages among nodes of the tree. Users can manipulate the structure freely, leaving the mechanics of how messages pass among its component objects to the underlying execution mechanism. This simplifies manipulation tasks not only for the user, but for other programs and processes as well. The TTree, then, has the potential to unite many of the advantages of declarative language descriptions with object-oriented and programming technology.

3.9 TTree Traversal

As mentioned in Section 3.3, traversal algorithms are critical for implementations of the 3TG metaphor, for they are the basis by which the TTree is both displayed and played. This section introduces a number of traversal algorithms in the context of object-oriented programming. The algorithms are presented as methods of some class whose instances are nodes in a TTree. While the terminology used is that of the TTree, these methods make no reference to the TTree's unique temporal structure (this discussion is reserved for Section 3.10). Consequently, they are just as applicable to binary trees as they are to TTrees.

Unless otherwise stated, all the following traversal methods are activated by messaging the root node of a binary tree with the method name. Although only preorder traversal methods are shown, most of these methods can be easily converted to inorder by simply moving the *self visit* message expression textually past the *is traverse* message expression.

Preorder traversal of a binary tree can be accomplished recursively as follows:[12]

```
traverse
    self visit.
    is notNil
        ifTrue:[is traverse].
    then notNil
        ifTrue:[then traverse]
```

Method 1. Preorder traversal of a TTree.

[12]Briefly, a Smalltalk-80 message send is written simply as *receiver message*, or, if the message takes an argument, as *receiver message: argument*. Variables local to a method are declared by enclosing them between vertical bars, as in | *local* |. Assignment is designated by a left-pointing arrow,←, while the uparrow symbol, ↑, precedes the name of the object to be returned from a method. The boolean infix operator, "==", yields true if and only if both its operands are exactly the same object. Finally, code enclosed in square brackets, such as [*receiver message: argument*] is called a *block* in Smalltalk-80: its execution is deferred until it is sent the message *value*. Blocks are often passed as arguments in messages to instances of the Boolean subclasses *True* and *False*. To illustrate, the *ifTrue:* method of class *False* never evaluates its block argument, while the corresponding method in class *True* always evaluates it (by sending it the message *value*).

Note the two uses of the message *notNil* in Method 1. These messages function as conditionals for testing the existence of *is* and *then* objects before these objects are themselves messaged recursively. They implement the termination conditions for the two recursive descents. They can easily be eliminated, however, by filling all empty instance variable slots in each object in the tree with *sentinels*: special objects used to mark the tree's boundaries. No matter what messages they receive, sentinels simply return, effectively terminating any recursive descent without the need for a conditional. Thus Method 1 can be simplified to:

```
traverse
    self visit.
    is traverse.
    then traverse
```

Method 2. Preorder traversal of a TTree with sentinels.

Observe that if there are n objects in a binary tree, then exactly $n + 1$ *is* and *then* instance variable slots in the nodes of the tree will be nil.[13] One single sentinel object can be used over and over to fill all these slots. The technique removes $2n$ conditionals from any traversal of the tree, while introducing $n + 1$ extra message sends. Whether or not this results in a computational saving, however, will depend on the particular implementation language. In Smalltalk-80, for example, where each conditional itself involves 2 message sends, the use of sentinels can result in significantly faster traversal. In languages where conditionals involve no message sends, however, the use of sentinels may actually increase the time it takes to do a complete traversal.

Note that Method 2 is a tail-recursive method, that is, the last executed statement is a call to itself. In order to save stack space, tail-recursion may be replaced by a reassignment of the message parameters followed by iteration. Since the *traverse* message has, apparently, no parameters, the process should be straightforward. In reality,

[13]To see this, note that a binary tree with n nodes will have $2n$ slots for *is* and *then* instance variables. Every object in the binary tree except the root will have exactly one instance variable–either an *is* or a *then*–referring to it, thus there will be $n - 1$ such references. Consequently, $2n - (n - 1) = n + 1$ of these instance variables will be nil.

however, most object-oriented languages always pass along one invisible parameter with every message send, namely the identity of the object to whom the message is being sent. The variable *self* is bound to this value before the corresponding method is executed. This means that *self* must itself be reassigned in order to eliminate the tail-recursion, as in the following:

```
traverse
    [self isSentinel]
    whileFalse:
        [self visit.
          is traverse.
          self ← then]
```

Method 3. Preorder traversal with sentinels and tail-recursion elimination.

This method assumes that non-sentinel objects answer *false* to the message *isSentinel*, while sentinels answer *true*. Unfortunately, the method is not correct Smalltalk-80, for that language, like many object-oriented languages, prohibits the explicit assignment of the variable *self*. The following method, which takes the root of the tree as an argument, has no assignments to *self*:

```
traverse: aRoot
    | aNode |
    aNode ← aRoot.
    [aNode isSentinel]
    whileFalse:
        [aNode visit.
          self traverse:  aNode is.
          aNode ← aNode then]
```

Method 4. Preorder traversal with sentinels and tail-recursion elimination, and with no assignments to *self*.

One drawback of all the traversal algorithms discussed so far is that they behave slightly differently if their recursive descent begins at the root than if it begins at some other object in the binary tree. If we apply Method 2, for example, to the node labeled *stretto* in Figure 3.5, then the *coda* will be visited in addition to the subtree rooted in *stretto*, which in many cases will not have the desired effect. Instead, an "auxiliary" function can be used:

```
traverse
    self visit.
    is auxTraverse

auxTraverse
    self visit.
    is auxTraverse.
    then auxTraverse
```

Method 5. Preorder traversal using an auxiliary function.

This section concludes with an iterative, *right-threaded* [Knu73, page 323] preorder TTree traversal algorithm. In right-threaded binary tree traversal, nil right-subtree pointers in nodes of the tree are set to point to the inorder successors of these nodes, resulting in efficient algorithms for both preorder and inorder traversal. These algorithms traverse subtrees without the necessity of any "auxiliary" functions as in Method 5, and, by eliminating recursion, provide great savings in stack space.

In Section 3.7, "ancestor" links were described which connected the nodes of a TTree to their ancestors in the corresponding ordered tree. It is not hard to see that for any node in a TTree with a nil *then* pointer, the node pointed to by such an ancestor link is precisely its inorder successor. This gives rise to the following method for preorder traversal:

```
traverse
   | aNode |
   aNode ← self.
   [true] whileTrue:
   [ aNode visit.
     [ aNode is isSentinel]
     ifFalse:[ aNode ← aNode is].
     ifTrue:
     [ [ aNode then isSentinel]
      ifFalse: [ aNode ← aNode then].
      ifTrue:
      [ [ aNode then isSentinel]
       whileTrue:
       [ aNode ← aNode ancestor.
        [ aNode == self]
        ifTrue:[ ↑self].
        ifFalse:[ aNode ← aNode then].
  ]]]]
```

Method 6. Preorder traversal using sentinels and threaded traversal.

The traversal algorithms of this section provide a means for displaying scores in the 3TG model. If, for example, every glyph of a TTree responds to the *visit* message by drawing its image onto the screen, then a complete tree traversal will completely redraw the score. In the next section, these same algorithms form the basis of simple and efficient methods for score performance.

3.10 Playing TTrees

This section extends the algorithms of Section 3.9 to make use of the TTree's unique ability to situate its nodes in time. There is, of course, much more to playing a score

than simply locating musical events in time. As we shall see, however, this is simply a question of granularity: the TTree can generate discrete sound samples as easily as it can generate high-level musical events requiring event-schedulers and synthesis hardware for their final realization.

The present discussion adopts the convention that "playing" or "performing" a node in the TTree is accomplished by sending that node one or more *play* messages at a predetermined time. In response to this message, the node's corresponding *play* method is responsible for actuating the available sound synthesis machinery. Playing a TTree, then, is the process of delivering these play messages to its nodes in accordance with the delay values associated with each node.

The most straightforward way of playing a TTree is to precompute absolute times, relative to the tree's root, for all the nodes in the tree. Any of the traversal methods given in Section 3.9 can then be used to write out a note list [DJ85, pages 14-17] or a series of time-tagged instructions for some event-scheduling process as described in [AK86] or [Dan89a]. For any given node, its absolute time will be the the sum of all the delay values along the path from the root to itself, not including its own delay value. Regardless of whether preorder or inorder traversal is performed, the nodes will probably have to be sorted into ascending order of begin time, for many note list processors require their input to be in this form.

Despite its simplicity, this approach contradicts an important goal enunciated in Section 3.2, for it is little more than a conversion function for creating a quasi-independent data structure for TTree performance. The following method avoids creating such structures by embedding scheduling control within the methods of the node objects themselves:

```
tick
    self play.
    is tick.
    self wait ifFalse:
        [then tick]

wait
    delay = 0 ifTrue:
        [↑ false].
    delay ← delay - 1.
    ↑true
```

Method 7. Sampling algorithm for TTree performance.

Although the methods introduced in this section use the recursive descent structure of Method 2, any of the methods of Section 3.9 may be similarly adopted. Method 7, called the "sampling" algorithm, assumes that some process will send a series of *tick* messages to the TTree's root at a fixed sampling rate. The *tick* message itself propagates recursively down the *is* branches of the tree, but can only propagate along *then* branches when a node's *wait* method returns *false*. As successive *tick* messages arrive at a given node, that node's *delay* value will eventually time out, causing the *wait* method to return *false*, and allowing the *tick* message to flow not only downwards but along the node's vertical *then* branch as well. As in Method 2, this method uses sentinels to terminate the recursive descent. Figure 3.6 shows a score fragment together with snapshots of the corresponding TTree as it evolves in response to the five *tick* messages needed to perform it in its entirety.

In Method 7, the *wait* method operates by simply decrementing a counter called *delay*, returning *false* only when the counter has reached zero. Any temporal operation which returns *true* or *false*, however, can be used as a *wait* method. Processes like "wait until my 'is' branch has finished playing" or "wait until the ballerina assumes her arabesque downstage right" exemplify some of the possibilities.

If the *play* methods of the nodes in a TTree generate individual sound samples

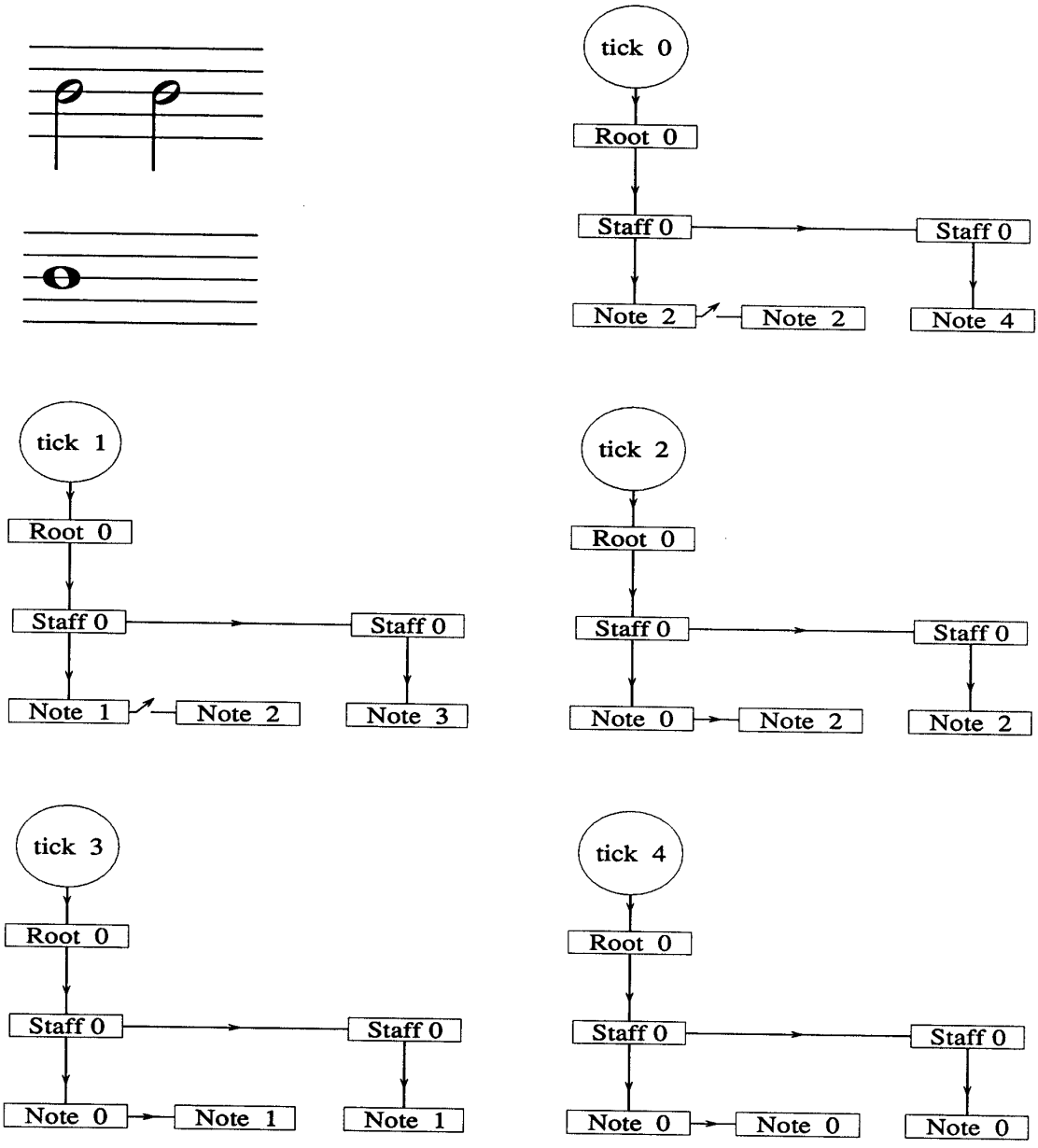


Figure 3.6: The sampling algorithm. The figure shows "snapshots" of the sampling algorithm playing a score fragment. The fragment is deliberately left incomplete for simplicity. As successive *tick* messages course through the TTree, the delay value of the first half note eventually times out, allowing the second half note to receive the *tick* message.

with each message, than they will typically require thousands of *play* messages to complete their tasks. On the other hand, if these methods generate, say, MIDI events, then they may need no more than a single message. In either case, however, they will eventually begin receiving messages they have no use for. Further, as more and more *then* branches open up, the effective size of the tree being traversed grows and grows. As a result, traversal takes progressively longer and longer to execute, with many nodes receiving the message long after they have played themselves out.

Many of these redundant messages can be eliminated, however, by revising the *tick* method to answer *true* whenever more play messages are needed, and *false* otherwise:

```
tick
    selfNeedsMore ifTrue:
        [selfNeedsMore ← self play].
    isNeedsMore ifTrue:
        [isNeedsMore ← is tick].
    thenNeedsMore ifTrue:
        [self wait ifFalse:
            [thenNeedsMore ← then tick]
        ].
    ↑selfNeedsMore | isNeedsMore | thenNeedsMore
```

Method 8. Sampling algorithm for TTree performance with message blocking.

Method 8 assumes that the TTree's sentinels respond to *tick* with *false*. The method uses three boolean-valued instance variables: *selfNeedsMore*,*isNeedsMore*, and *thenNeedsMore*. These variables, which must all be initialized to *true* before playing begins, prevent the tick message from propagating to nodes that no longer need it. As an added bonus, the root node of the TTree will return *false* in response to *tick* only when the entire tree has played, effectively informing the original message sender that the performance has finished. No such termination information is provided by Method 7.

Method 8 implements "message-blocking", because it blocks tick messages from

```
tick
    selfNeedsMore ifTrue:
        [selfNeedsMore ← self play].
    isNeedsMore ← is tick.
    self wait ifFalse:
        [thenNeedsMore ← then tick].
    isNeedsMore ifFalse:
        [is ← is then].
    thenNeedsMore ifFalse:
        [then ← then then].
    ↑selfNeedsMore | (is isSentinel not) |
        (then isSentinel not)
```

Method 9. Sampling algorithm for TTree performance with subtree deletion.

The efficiency of Method 9 is gained at quite a price: once the TTree has been performed in its entirety, only its root node will be left intact! This deficiency, however, can be remedied for the cost of two new instance variables. When initialized to each node's *is* and *then* pointers before playing, these variables can take the place of the original pointers during performance, leaving all original linkages intact.

As a final refinement of TTree performance algorithms, note that for certain trees, most *tick* message traversals will function only to count off delay values, and will cause no *play* messages at all to be sent. Such "ineffective" traversals can be skipped if, at a particular time in the tree's performance, the delay until the next *play* message will be sent is known in advance. Then, through a simple modification of the *wait* method of Method 7, such traversals can be eliminated:

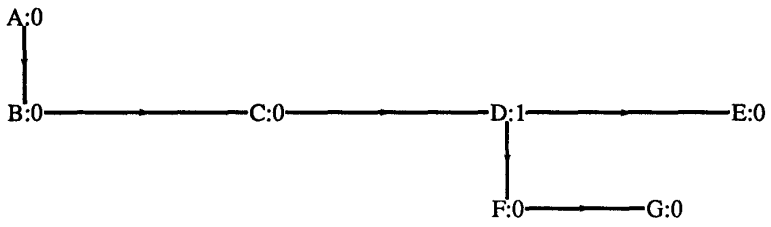


Figure 3.7: A TTree demonstrating message-blocking.

traveling downwards or to the right in the tree. To illustrate, assume that every node in the tree of Figure 3.7 returns *false* after receiving but a single *play* message. The first tick will reach all nodes except E. The second tick message will reach node E, but will be blocked from reaching nodes F and G, because D's *isNeedsMore* instance variable will be *false*. But consider the path this second tick message must take to reach E. It must travel from node A through nodes B,C, and D. A More refined approach would eliminate these intermediate nodes, shortening the traversal path by forging an arc directly from A to E.

The following method accomplishes path-shortening through "subtree deletion": the process of removing arbitrary subtrees from the tree once they no longer require *tick* messages. Each node does its subtree deletion work only after both its *is* and *then* subtrees have been traversed. If either subtree returns *false* in response to *tick*, then it no longer requires *tick* messages, and can be removed from the tree.

```
wait
    delay ← delay - (conductor tickLength).
    delay = 0 ifTrue:
        [↑ false].
    conductor nextPlayIn:  delay.
    ↑true
```

Method 10. Modified wait method for eliminating "ineffective" ticks.

Rather than assuming some fixed message rate, Method 10 relies on a *conductor* object sending *tick* messages to the tree at prescheduled times. With each traversal, nodes subtract the value of "*conductor tickLength*" (the time since the last *tick* was sent) from their *delay* instance variable, passing the new value to *conductor* through the message "*nextPlayIn: delay.*" After every traversal, *conductor* schedules the next *tick* message as the minimum of all the delays it received.

In his essay *Composing with Computers* [Loy89], Gareth Loy distinguishes two basic models of event scheduling for music. Strict functional models do their work by polling at fixed intervals. Event-oriented models, on the other hand, with no notion of sample rate, are controlled only by the frequency of musical events. Both have their advantages and disadvantages, and successful systems have been built using both approaches. As we have seen, the TTree structure supports both models, for the sampling algorithms clearly belongs to the functional category, while Method 10 describes an event-oriented approach.

3.11 Summary

Chapter 2 introduced the metaphor of three-dimensional, transparent glyphs (3TG) as a way to organize graphically-displayed musical scores in computerized compositional environments. This chapter has presented a data structure which not only supports the displaying of such scores, but the playing of them as well. Called a *TTree*, the structure is the binary correspondent of an ordered tree with time-delays associated

with its right-going arcs. Like binary trees, TTrees are often better conceptual models of musical structure than their multi-branching correspondents, and may well be more efficient in terms of computer implementation.

The connection between TTrees and formal grammars suggests the possibility of using the structure for both compositional algorithms and automatic score-generating programs. In an object-oriented system, TTrees provide both a well-organized instance hierarchy greatly facilitating object-to-object communication, and a powerful behavior-sharing mechanism. These features result in a declarative-like data description which contrasts with the procedural descriptions of many music-representation schemes.

A variety of TTree traversal algorithms with differing properties can be implemented within the object-oriented paradigm. Simple extensions to these algorithms provide efficient and elegant methods for TTree performance.

Chapter 4

User Interface Design

4.1 Introduction

Chapter 2 of this paper introduced the three-dimensional transparent glyph model of the notated score. In this chapter, the unique style of musician-machine interaction implicit in the model is explored. Issues concerning the design of a prototypical user interface based upon the model are discussed. This discussion forms the basis of the user interface to the music notation program described in the following chapter.

4.2 Scope of the Discussion

Decisions about user interface design are made from different perspectives by different people. All too often, such decisions are based solely on the personal taste, experience, and intuition of the software developer. Fortunately, in today's commercial software development world, design decisions are frequently guided by preestablished standards. Many computer manufacturers, for example, in an effort to achieve consistent user interfaces across applications, publish such references as "Human Interface Guidelines: The Apple Desktop Interface" [App88], or "The NeXT User Interface" [NeX89a]. Unfortunately, only rarely are such guidelines formulated according to factually or experimentally verified principles, in spite of the ever-growing body of results obtained from user interface research.

The current focus of the human-computer interaction research community can be divided into three categories.[1] First, software engineers are exploring the technologies of object-oriented, distributed, and knowledge-based systems for creating highly interactive, visual computer environments. Second, human factors experts are attempting to map out the basic cognitive processes underlying our interaction with computers. Finally, a growing number of researchers are exploring such topics as workgroup computing, participatory design, and the role of the computer in society.

For a compositional environment based on the 3TG model, how can these research results help in the design of a user interface? During the earliest stages of design, this research is probably at its least prescriptive. As Day points out,

> ... the psychological and human-factors literature is rarely specific enough to describe precisely how a user interface should be designed. This is especially true when developing new technologies or designing complex user interfaces. Therefore, a human factors specialist designing a user interface relies on established knowledge and experience for the first "draft" design. This design is then tested for usability.[Day89, page 4]

The importance of iterative usability testing [GL85] and other empirical measures of interface design cannot be overestimated. Such testing, however, cannot be carried out without an implemented, prototypical system to carry it out upon. To this end, the present chapter consigns the task of usability testing to further research, and focuses exclusively on the design of a first "draft" system. Chapter 5 describes the implementation of just such a system, based on the discussion of this chapter together with the TTree data structure presented in Chapter 3.

4.3 Conceptual Integrity

The first draft of a user interface, then, must rely heavily on "established knowledge and experience." One theme in particular, the concept of *conceptual integrity*, will underlie all of the discussion of this chapter. To quote from Frederick Brooks:

[1] Grudin's paper discusses the historical evolution of the changing focus of user interface research and development [Gru90].

> I will contend that conceptual integrity is *the* most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.[Bro75, pg. 42]

What Brooks articulated in 1975 has become, in various formulations, a familiar refrain in user interface design. One way of achieving a conceptually integral user interface is to base all decisions about its design on a single, pervasive metaphor. If this metaphor is powerful enough, many of the details of the interface will follow naturally from it. In this way, rather than obliging users to rummage through voluminous technical documentation, they can *predict* how certain actions are achieved by direct inference on the metaphor. At the same time, a single metaphorical design basis helps to foster a consistent interface. And as Peter Polson has demonstrated, " ...consistent user interfaces mediate positive transfer within an application across tasks...These results should surprise no one; the important fact is their magnitude" [Pol88, page 103].

It can be a creatively challenging task to find a single metaphor that can wrap an entire application domain into one conceptual package, particularly if that domain is highly abstract or unfamiliar. Fortunately, the 3TG model introduced in Chapter 3 is both concrete and familiar. Stacking objects into piles, for example, is a common, everyday experience, and although musicians do not usually conceptualize scores as three-dimensional piles of symbols, the concept of putting staves *on* pages, notes *on* staves, dots *on* notes, and so on, is familiar to all those acquainted with traditional western notational practice. For this reason, the 3TG model makes an excellent metaphor for the user interface to a music notation program. By basing all decisions about interface design directly on this metaphor, a conceptually integrated compositional environment for music notation can be built.

4.4 Direct Manipulation

Schneiderman, in [Shn83], portrays *direct manipulation* systems as follows:

- Continuous representation of the object of interest.
- Physical actions (movement and selection by mouse, joystick, touch screen, etc.) or labeled button press instead of complex syntax.
- Rapid, incremental, reversible operations whose impact on the object of interest is immediately visible.
- Layered or spiral approach to learning that permits usage with minimal knowledge.

In effect, direct-manipulation interfaces permit users to operate directly on computer representations of objects themselves, rather than indirectly through some intermediate language. These interfaces are known to decrease the demands on user's short- and long-term memory [Jac89]. At the same time, most users find them easier to master and more enjoyable to use than the corresponding indirect-manipulation interfaces. Such "fostering of enjoyment" among users is a key characteristic of any system boasting the enhancement of creativity among its goals.

The 3TG model readily lends itself to a direct-manipulation style of interaction. This interaction must include a facility for removing a given glyph from one pile and placing it on another. This operation is easily effected using a graphics input device such as a mouse, light pen, or joy stick, and a single "drag" button.[2] The input device is used to position a cursor over the glyph to be moved. Then, with the drag button held down, this glyph can be dragged to its new location. If two drag buttons rather than one are used, the second button can be used instead of the first to indicate that a *clone* (deep copy[3]) of the glyph should be made, and that it, rather than the original, is to be dragged to the new location.

This manipulation metaphor will be referred to as *clone-and-drag* in order to distinguish it from the more prevalent *cut-and-paste* metaphor implemented by many

[2]A graphics input device such as a mouse which allows both itself and a button or buttons to be operated with a single hand may be preferable to one requiring two hands. In a musician's studio there may well be other tasks—playing a keyboard, operating the controls on an amplifier, waving a conducting baton—which the musician wishes to perform concurrently with commanding the computer.

[3]The the article [Die89b] for a discussion of deep versus shallow copying in computer music systems.

graphically-based user interfaces today. With cut-and-paste, the object of interest is first *selected* by the user. Then, either the command *cut* or *copy* is executed, generally by choosing the command from a menu. This typically causes either the object itself or a copy of the object to be placed in some invisible buffer. Next, the desired destination location for the object is indicated, and the command *paste* is executed to copy the object from the invisible buffer into the destination.

Note that this characterization of the cut-and-paste metaphor does not really fit Shneiderman's portrayal of a direct manipulation interface, for the object of interest is not continuously represented during the operation: after the cut or copy command, the object becomes invisible until the paste command is issued. As a result, users must retain an image of the object in their memories in order to use the metaphor effectively. If this memory image is lost, the user risks inadvertently replacing the object in the invisible buffer with some other object. Some systems implement the buffer as a stack of objects, alleviating the problem of inadvertent replacement at the cost of greater demands on the user's memory store.

By contrast, the clone-and-drag metaphor fits Shneiderman's portrayal precisely. The object of interest is continuously represented during the operation, freeing users from the necessity of keeping an image of the object in their own memory. The four discrete physical actions necessary to complete a cut-and-paste operation—select object, cut, identify destination, paste—are replaced, in clone-and-drag, by two continuous ones: select and drag to destination. Further, users can view the effect a clone-and-drag operation will have on a representation *before* they commit to it by releasing the drag button. For these reasons, clone-and-drag operations are often found in computer drawing programs, and they make excellent manipulation methods for repositioning glyphs in a 3TG-based compositional environment.

4.5 Windows

After reviewing several research projects concerning window system design, Shneiderman concludes that the advantages and disadvantages of various design approaches are still too poorly understood to be of much guidance to the developer [Shn87].

Nevertheless, design choices will have to be made, for the pervasiveness of windowing systems on computers today makes it hard to imagine a graphic user interface without one. For many development projects, the decision to use some form of *user interface management system*[4] will obviate the need to do much in the way of basic window design. Such management systems, however, still give the developer enough latitude to seriously affect the usability of the interface. Consequently, designers of successful window-based applications must invest a good deal of thought into the ways in which their applications might map into their chosen windowing environment.

A likely scenario for a 3TG window-based system assigns to each window of a potentially multi-window application the task of displaying a single hierarchy of glyphs. In this scheme, the background of each such window represents the root glyph of the hierarchy. Glyphs can be dragged not only from one pile to another, but from one hierarchy (window) to another as well.

If glyphs can be dragged from one window to another, then a unique method for deleting glyphs can be implemented. In this method, deletion is triggered when a glyph is dragged out of its window and released at some location outside of all of the application's windows. Adopting this scheme simplifies the user interface by eliminating the need to dedicate a menu item or other control scheme to the task.

Window-to-window mobility also provides a convenient means for delivering sets of glyphs to the user. In place of some specialized palette scheme with its requisite set of commands, windows may be "pre-loaded" with sets of individual glyphs. Users drag glyphs from these pre-loaded windows into their "working" windows as needed. Except for the fact that such windows have been pre-loaded, they are identical to all windows in the system. As a result, no new commands are needed to manipulate them, thus the number of commands the user must master is kept to a minimum.

[4]A user interface management system [BC89, page 207] is a software support package, often provided by the computer vendor, which defines much of the look-and-feel of the applications which use it.

4.6 Grouping

There are many potential circumstances in which a group of glyphs would be best treated as a unit rather than as a set of individual objects. As a simple example, when a complex glyph pile has been built up, the user may wish to guard against the possibility of inadvertently destroying the structure through some accidental clone-and-move operation. For cases such as these, a scheme in which piles may be "glued" or "frozen" together may suffice: when frozen, the individual glyphs which make up the pile cannot be repositioned with respect to the pile's foundation (root) glyph.

One way in which the interface to a 3TG-based compositional environment can be greatly simplified is to ensure that any operation which affects a single glyph also effects every glyph piled upon it, whether the pile is frozen or not. Thus, when a glyph is dragged, all glyphs piled upon it are dragged as well. Similarly, when a glyph is cloned, all glyphs piled upon it are cloned as well. This simplicity is gained at a cost, however, for if users wish to operate on a glyph without affecting the glyphs piled upon it, they must first move these piled glyphs off of their foundation before performing the operation. A parallel set of operations operating only on single glyphs rather than on glyph piles could be implemented, though this might unduly increase the complexity of the interface, sacrificing a good measure of conceptual integrity in the process.

The glyph pile, then, can be used as a unit which implicitly specifies the *scope* of certain operations. Buxton et. al. define *scope* as "the sphere of action of any activity" [BPRB81]. Their article distinguishes a variety of approaches to scope in interactive score editors, and suggests that for the *demonstrative* approach ("transpose this note, that note, and that note"), graphic manipulation is the most appropriate tool. The demonstrative approach is easily implemented in a direct-manipulation system using what might well be termed "scope-by-pile." Unfortunately, it will not always be the case that every glyph in some desired scope will belong to a common pile. Since such situations are likely to be very common, an appropriate mechanism must be found which is still metaphorically related to the 3TG model.

Many graphic user interfaces implement "multiple selection": a mechanism by

which more than one object can be selected, and thus operated upon, at a time. The idea of scope-by-pile is itself a form of multiple selection, but a severly restricted one for two reasons. First, only glyphs which are connected to a common subglyph can be selected. Second, if two glyphs which are connected by a common subglyph are selected, then so are any other glyphs sharing the same subglyph, including that subglyph itself. Nevertheless, these restrictions can be removed by allowing operations to effect more than one pile at a time, effectively expanding the notion of scope-by-pile into scope-by-set-of-piles.

There is another method by which the shortcomings of the scope-by-pile method can be circumvented, a method which is perhaps closer in spirit to the 3TG model than is the scope-by-set-of-piles concept. This method introduces the concept of the *tray*: a glyph which functions as the missing common subglyph under two or more other glyphs. When a glyph is selected in some "special" way (by double-clicking with a mouse, for example), a tray glyph can be created and attached *under* the selected glyph. Subsequent special-selections cause the original tray to grow in size to support each subsequent specially-selected glyph. Further, every such glyph can be automatically "frozen" to the tray. When the multiple selection is complete, users manipulate the tray just as they would any other glyph, with the exception that when the tray is "unfrozen", it deletes itself, leaving behind all the glyphs sitting on top of it.

The tray concept is more general than the scope-by-set-of-piles mechanism described earlier, for multiple trays can easily coexist within the same application at any given time. It is as if the interface supported multiple multiple-selections. Unfortunately, due to their unusual selection and deletion behavior, trays do contribute to the conceptual complexity of the interface. Nevertheless, since trays have a concrete, metaphorical relationship to the real world, that complexity may be easy for the user to surmount. Clearly, only empirical investigation can determine whether this is in fact the case.

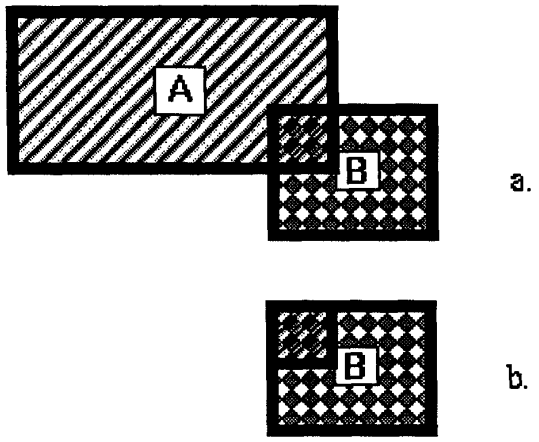


Figure 4.1: Clipping. The figure shows two possibilities for displaying a glyph which overhangs another.

4.7 Clipping

For any implementation of the 3TG model, there will be several possibilities for dealing with the situation of one glyph piled on top of another such that its edges overhang the glyph it is piled upon. Referring to Figure 4.1, when glyph A is piled on glyph B, should the portions of A which do not intersect B be visible, as in 4.1a, or should A be clipped to the border of B, as in 4.1b?

I believe that the clipping approach, as exemplified by Figure 4.1b, is preferable for several reasons, even though it contradicts the concrete, real-world characteristic of the "piles of objects" metaphor (in the real world, objects don't clip their borders to each other). Because of the transparency of glyphs, adopting a non-clipping approach leads to ambiguities about how the glyph hierarchy is structured. In Figure 4.1a, for example, it is impossible to decide on the basis of visual data alone whether A is on B, B is on A, or, for that matter, whether A and B actually coexist at the same hierarchical level and simply overlap visually. By contrast, Figure 4.1b makes it much more visually apparent that A is resting atop B.

Of course, it could be argued that in Figure 4.1b, it is not clear that anything at all

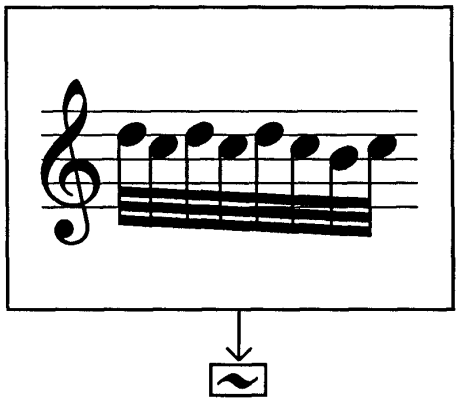


Figure 4.2: Clipping to suppress detail. A trill figure with the trill fully written out as a fragment of common music notation, then "mounted" on a trill symbol is shown. The arrow indicates that the fragment has been dragged over top of the symbol. It is aligned so that only a part of its area which contains no drawing is "visible" over the symbol—the rest is clipped. Both the fragment and the symbol are enclosed in rectangles to show the extent of their glyphs.

rests on B. Since users don't see all of A, the rectangle in B's upper left corner could simply be a part of B's image. Yet the fact that clipping makes most of A disappear can be used to advantage. As we shall see, the ability of clipping to suppress detail opens up a whole new set of possibilities for the 3TG model, as witnessed by the following examples.

In Figure 4.2, a trill figure is fully-written out in a fragment of common music notation, then frozen onto a glyph containing a trill symbol, where it is clipped to the glyph boundary of that symbol. Because a portion of the fragment containing no drawing at all is located over the symbol, the fragment is entirely invisible. Furthermore, because the fragment is frozen to the symbol, it is inaccessible until the unit is unfrozen. When unfrozen, the user can drag the fragment off of the symbol and alter it as desired.

Clipping can also be used to hide information not normally associated with a musical score. In Figure 4.3, the glyph pile at the left is shown on the right in

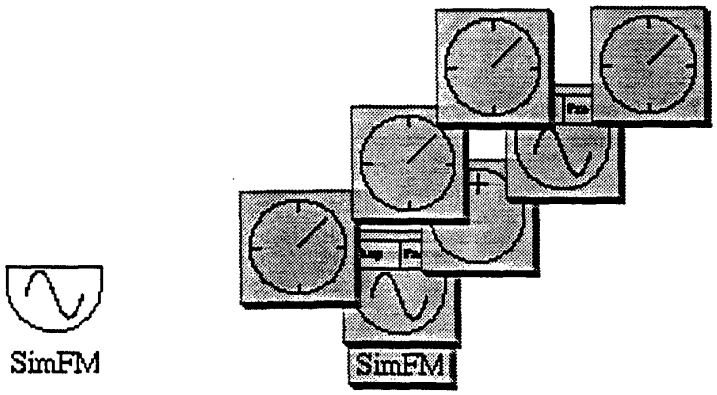


Figure 4.3: Two views of an FM instrument. The 3-d view on the right reveals structures which have been visually clipped from the view on the left.

an alternate representation where clipping is turned off and all glyphs are mounted on opaque platforms to emphasize their three-dimensionality. The figure shows a schematic diagram of a simple FM instrument [Cho85]. The schematic includes two glyphs representing sinusoidal oscillators (labeled by a sine wave cycle), an adder glyph (labeled by a circle with a plus sign near the top), and four constants (labeled by circles resembling the faces of clocks).

Finally, Figure 4.4 exemplifies how musical structure itself can be encapsulated by clipping. In this figure, a musical phrase is "hidden" on top of a glyph labeled "Soprano". Other structure-defining glyphs may be combined with it into larger and larger formal compositional units, until a complete composition is formed.

Clipping, then, carries with it an enormous potential for hiding visual detail from the user. Although it may not be the case that the 3TG model is the most convenient representation for all aspects of composition and synthesis, it is certainly conceivable that a composer's entire compositional world, from the lowest level of sample generation to the highest level of formal organization, could be united under a single metaphor and manipulated from a simple and consistent user interface.

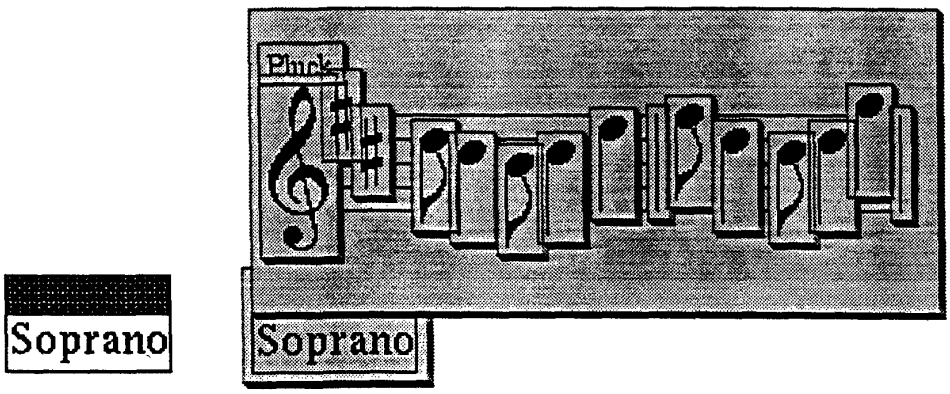


Figure 4.4: Two views of a structure-defining "phrase" glyph

4.8 Summary

This chapter has presented a discussion of user interface design for a compositional system based on the three-dimensional, transparent glyph (3TG) model introduced in Chapter 2. While recognizing the importance of iterative usability testing, the chapter has focused exclusively on the early, prototypical design phase of the interface in order to provide a foundation for implementing a running, testable prototype. Adopting the principle of conceptual integrity as a fundamental assumption, an interface has been described which relates all its functionality to the the metaphor implicit in the 3TG model of the musical score. A direct-manipulation system employing a "clone-and-move" strategy in place of the familiar "cut-and-paste" editing style was described. The use of a multi-window environment in which glyphs can be moved from window to window reduced the complexity of the interface by eliminating the need for any separate palette mechanism. Problems concerning the grouping and editorial scope of glyphs were discussed, and the concept of a *tray* was introduced as a possible solution. Clipping was shown to provide a powerful and unique method for suppressing visual detail.

Chapter 5

An Implementation of the 3TG Model

5.1 Introduction

This dissertation has, until now, diligently avoided reference to any particular implementation of the three-dimensional transparent glyph model of music notation. Indeed, one of the motivations behind describing the system in terms of a model was to avoid reference to any particular computer hardware or software. Nevertheless, the model is intended as a framework for implementation, and its formulation benefited enormously from the parallel development of a working computer system, dubbed *Nutation*. The present chapter describes Nutation in terms of the technical resources it is built upon, in terms of the underlying programming system it supports, and in terms of the interface it presents to the musician. Finally, three different styles of music notation implemented within the system—piano roll notation, common music notation, and Okinawan notation—are presented. A basic knowledge of object-oriented concepts and terminology on the part of the reader is assumed.[1]

[1] Cox's book contains a good introduction to object-oriented concepts and terminology [Cox86].

5.2 Goals

The implementation of a computer system based on the 3TG model was undertaken with several major research goals in mind. As discussed in Chapter 4, iterative usability testing of user interface design cannot be carried out without the existence of a working prototype to carry it out upon. The creation of just such a prototype, with the promise of further refinement of the model through empirical means at a later stage of development was a major goal of the implementation. At the same time, while certainly not constituting "proof" of any rigorous kind, the existence of a working system would lend credence to many of the claims about the model articulated in the foregoing chapters. The model was claimed to be:

- a new direction in computerized music notation systems.
- a model for the *compositional* use of notation.
- a means for conveniently representing a wide variety of notational styles.
- because of its similarity to the physical world, a mental model of great benefit to the potential user.
- the visual representation of an underlying hierarchy, which in turn was claimed to be an ideal data structure for music notation programs. When implemented as a TTree, the hierarchy was claimed to be both a good conceptual model of musical structure and a convenient data structure for both the displaying and the playing of scores efficiently in real time.
- conducive to an object-oriented, visual programming language implementation furnishing musicians with the opportunity to define their own notational worlds.
- strongly suggestive of a simple and conceptually integral direct-manipulation style user interface.

The creation of a fully-functioning notation system, ready for use by the computer-music community, was *not* a goal of this implementation. At an early stage of research,

such an undertaking was judged to be beyond the scope of this study.[2] Instead, development was limited to exploring those aspects of notation where the three-dimensional approach seemed especially appropriate, at the same time developing enough functionality to demonstrate the feasibility of a full-blown system.

5.3 Technical Resources

In the previous section, neither code portability nor machine independence were listed among the goals of the implementation. Freed from these constraints, the project was able to take advantage of the unique hardware and software facilities for real-time graphics and music synthesis provided by the NeXT™computer.[3]

The NeXT computer used for the project was among the earliest machines manufactured by the company, and included both the MC68030 central processor and the MC68882 floating-point coprocessor with 16 megabytes of random access memory, a 330 megabyte Winchester disk, and a 256 megabyte removable optical disk drive. In addition, the machine provided significant hardware support for sound generation through its DSP56001 digital signal processing chip clocked at 25 MHz, and its high-quality stereo digital-to-analog converter operating at 44100 samples per second with 16-bit quantization. With this setup, significantly complex, compact-disk quality stereo sound could be synthesized in real-time without the necessity for any external hardware other than amplifier and speakers.

The computer ran the Mach operating system [NeX89b] designed at Carnegie Mellon University, and supported full UNIX® 4.3 BSD [Wan88] compatibility. The principle implementation languages were Objective-C® [Cox86] and Allegro CL®

[2]Some idea of the size of the task can be gleaned from Donald Byrd's experience with his SMUT system. In his words: "My first estimate of how long it would take to have a usable program was on the order of a few months, not 14 years. Underestimates this extreme of the difficulty of programming CMN [Conventional Music Notation] seem to be rather common."[Byr84, page 114]

[3]All too often, the choice of hardware and software tools for computer music research has more to do with resource availability than with the exigencies of the task at hand. In this case, however, the availability of a NeXT computer—arguably the first machine to take support for music software development seriously— greatly facilitated the research.

(Common Lisp) [Fra]. A good deal of use was made of NeXT's programming *kits*: extensive, pre-compiled libraries of Objective-C classes. The *Application Kit* [NeX89c] provided a comprehensive user interface management system, while sound synthesis was done entirely through the *Music Kit* [JB89]. All graphics on the NeXT computer utilized the PostScript® system, including the basic PostScript language [Ado85] augmented by the Display PostScript® extensions [Ado89] and NeXT's own extensions to Display PostScript [NeX89d].

5.4 The System

5.4.1 The Programming Language

As discussed in Section 3.6, the implementation of glyphs as objects was an important facet of the 3TG model. From the programmer's point of view, Nutation is a computer environment for a visual programming language. Shu defines visual programming as "the use of meaningful graphic representations in the process of programming" [Shu88, page 9]. In Shu's categorization scheme, Nutation's language is closest to a "language for supporting visual interaction", for it supports visual representations and visual interactions, but the language itself is textual, not visual.

The language of Nutation follows the single-inheritance, class-based object model of Objective-C. Indeed, the language is "embedded" within Objective-C itself, while extending Objective-C in several significant ways. First, every object has a user-specifiable image which is displayed on the computer screen. Normally, this image is common to all instances of the same class. All instances of class *QuarterNote*, for example, display the same image, even though the size, or "magnification" of that image can vary from instance to instance. It is possible, however, for a class to specify that its instances have unique images. To illustrate, every instance of class *Beam* has a unique image determined dynamically by the notes associated with that instance. Without the capability for a unique image, every unique beam structure displayed by the program would have to correspond to a unique class.

The language of Nutation also differs from Objective-C in having the class *Glyph*,

rather than class *Object*, at the root of its inheritance hierarchy.[4] This *Glyph* class provides all the functionality one might expect of class *Object*, including methods for instance creation, copying, deletion, testing inheritance relationships, and so on. In addition, the class provides the functionality for the piling behavior defined by the model. Every instance of class *Glyph* has instance variables to support the TTree data structure described in Chapter 3. Instance methods provide for the interactive restructuring of the tree in response to messages generated by the user's actions on the NeXT computer's two-button mouse.

Whenever an instance of class *Glyph* receives a message for which no corresponding method can be found within its inheritance hierarchy, it responds by relaying the message to its ancestor in the TTree. This mechanism supports the form of delegation described in Section 4.7. The behavior of any given glyph, then, is determined not only by the class to which it belongs, but by the classes to which all its TTree ancestors belong as well.

Programming the system consists of first subclassing the *Glyph* class, specifying images for the new class, creating instances of the new class, then using the mouse to pile these instances on top of each other, creating the hierarchical TTree data structure in the process. Typically, users are relieved of all but the last of these steps, for the system provides collections of pre-defined and pre-created glyphs. Thus users are free to concentrate on the interactive, mouse-driven restructing of glyph piles. Nevertheless, the ability to create completely new *Glyph* subclasses or to modify existing ones is always a possibility.

One of the goals of the system is to allow the user to make small modifications to a class definition, test the results of that modification, modify again, and so on, without the necessity of compiling, linking, and restarting the entire program for every change. This encourages an incremental, exploratory style of program development. Unfortunately, when the program was developed, the NeXT computer provided no support for the "incremental linking" needed to create such a programming environment. The version of Common Lisp provided by Franz Inc., however, provided a

[4]Strictly speaking, the root of the inheritance hierarchy is still class *Object*, with class *Glyph* as an *Object* subclass. Conceptually, however, *Glyph* is at the root, for the system provides no way to create an instance of *Object* which is not an instance of *Glyph*.

partial solution to the problem.

Franz's Common Lisp system allowed compiled Objective-C to be incrementally linked into a running Lisp image, then executed directly from within that image. Accordingly, the bulk of the Nutation system (which consisted of compiled Objective-C class definitions) was linked into a Lisp image. Facilities were provided for editing, compiling, then linking *Glyph* subclasses into the image from within the image itself. To the user, this Lisp image was all but invisible, as incremental linking was virtually the only use made of the underlying Lisp system.

This method, however, had several unfortunate consequences. First, the size of the program grew substantially as a result, and taxed the NeXT computer's resources to the point of affecting performance. Second, it was no longer possible to use the debugging facilities provided for Objective-C, as they were never designed to deal with Objective-C embedded in Lisp. Finally, the incremental linking process itself was quite slow, making exploratory programming more of a frustration than a boon to creativity. For all these drawbacks, however, significant amounts of the program's development were carried out using the method which, for minor changes in the program's code, was considerably more convenient than relinking and rerunning the entire system.

5.4.2 The User Interface

Nutation implements a multi-window, menu-based system inheriting much of its look-and-feel from NeXT's *Application Kit* user interface management system. Though there may be many windows on the screen at the same time, every window falls into one of three window-types. This section describes these window-types, and introduces the user interface commands provided for manipulating them.

The first window-type to be described is the "glyph window", called simply "window". These windows are responsible for the display and manipulation of glyphs and glyph piles. Windows are created by menu selection, and are initialized by the system to contain a single glyph, called the *rootGlyph*.[5] Instances of class *RootGlyph*, these

[5]This chapter follows certain conventions in order to distinguish concepts from the names of classes and instances of those classes. Concepts use normal type, classes are indicated with *Slanted*

objects display a solid white rectangular image, and appear to be the "background" of the window they are displayed within.

Whenever a window is resized, its *rootGlyph* is also resized to ensure that it fills the entire window. These *rootGlyphs*, however, never shrink, they only grow. As Figure 5.1 shows, if a window is resized so that its borders are smaller than its *rootGlyph's* borders, then the window will display scroll bars along the sides and bottom, allowing portions of the *rootGlyph* outside of the window's boundary to be scrolled into view.

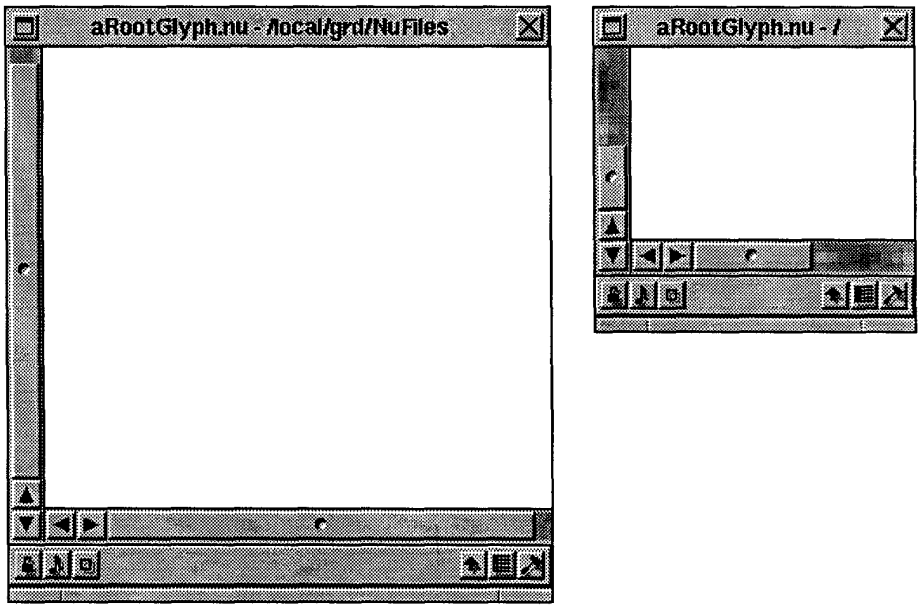


Figure 5.1: Nutation's glyph windows. The same window is shown before and after resizing. The scroll bars on the smaller window allow scrolling invisible parts of the *rootGlyph* into view.

The Nutation system provides a number of preprogrammed *Glyph* subclasses. These classes are all stored in individual disk files, and may be accessed by clicking

type and always begin with an upper case letter, while instances of classes are *slanted* but begin with a lower case letter. Thus "note" refers to the concept of a note, "*Note*" designates a class, and "*note*" refers to an instance of that class.

on the "new glyph" button (the second button from the right in the bottom, or "command" row of each of the windows in Figure 5.1), then choosing the desired class from a list. This instructs Nutation to create an instance of that class and pile it onto the *rootGlyph*.

A cursor tracks the location of the mouse on the computer screen. Whenever the cursor is positioned over a glyph and the left mouse button is held down, that glyph may be dragged to a new location. As long as the button is held down, the glyph is effectively detached from any glyph pile. As soon as the button is released, however, the system identifies that glyph (other than the one being dragged) which is directly underneath the cursor, and attaches the dragged glyph as one of its subglyphs. In this way, arbitrarily complex glyph piles can be built up.

Figure 5.2 illustrates Nutation's implementation of clipping. Whenever a glyph is placed on top of another, the uppermost glyph will be clipped to the boundaries of the glyph it is piled upon. As discussed in Section 5.7, this clipping allows for the deliberate suppression of detail from the image.

In some cases, the structure of a glyph pile may not be visually obvious. For this reason, a special version of the glyph window, invoked by the "raise" switch (third from the right in the command row), draws a different view of the glyph hierarchy. As Figure 5.3 illustrates, glyphs in a raised view do not clip, and they are drawn opaque and given thickness to emphasize their 3-dimensional structure.

Unfortunately, in this implementation, the raised view is a frozen entity: the glyphs it displays cannot be dragged, cloned, or manipulated in any way. A raised view with all the functionality of a glyph window would have been a very useful addition to the interface.

As an alternative to instance creation using the "new glyph" button, holding down the right mouse button instead of the left with the cursor positioned over an existing glyph will create a clone, or "deep copy" of that glyph. Although a window's *rootGlyph* cannot itself be dragged, it can be cloned, and the resulting copy can be dragged.

Glyphs are not confined to any particular window: they may be dragged out of one window and relocated in any other. When a glyph is dragged out of a window

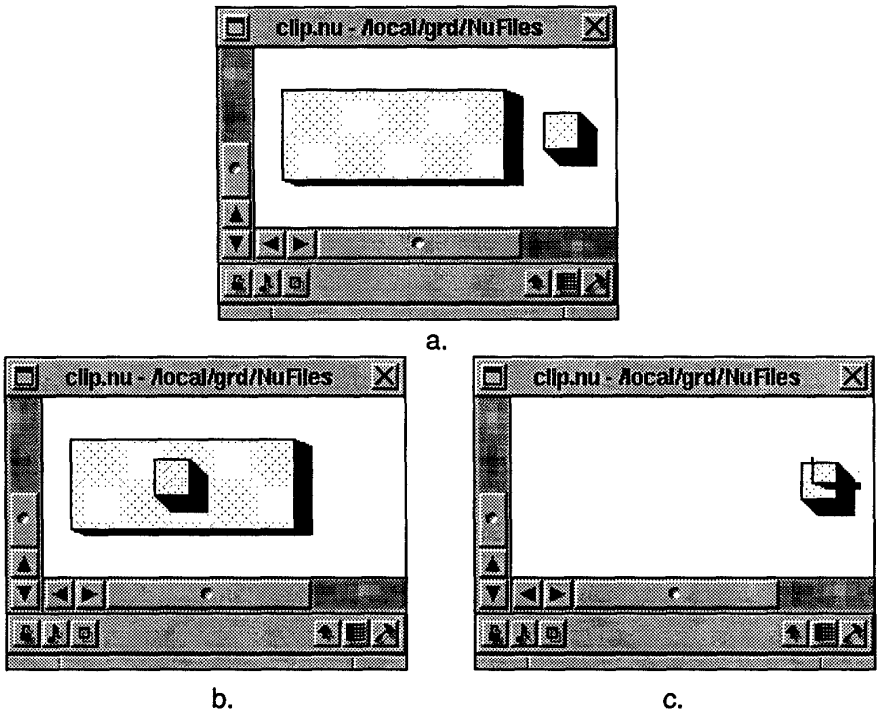


Figure 5.2: Clipping. This figure shows three Nutation windows. Window b displays the same two glyphs as window a, with the smaller glyph piled onto the larger. In window c, however, the larger glyph has been piled onto the smaller, causing its image to be visually clipped to the boundaries of the smaller.

and the mouse button released while it is outside of all glyph windows, that glyph is deleted from the system, and it disappears from the screen.

All of the above operations—dragging, cloning, and deletion—work identically whether they are applied to a single glyph or to a pile of glyphs. In fact, the only time that an operation affects but a single glyph is when there are no other glyphs piled on top of it. To temporarily protect the structure of a pile, it can be "frozen" together by activating the "freeze" button (third from the left in the command row). When frozen, the pile acts just as if it were an individual glyph, and none of its glyphs may be moved independently until the pile is "melted".

Any window, together with all its glyphs, may be saved to a disk file as a unit.

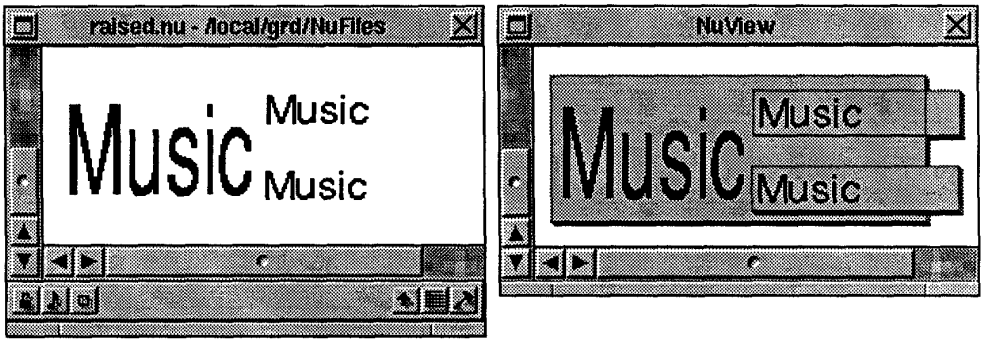


Figure 5.3: The raised view. The leftmost window above is shown on the right in an alternate "raised" display mode, with no clipping and with all glyphs made opaque and given "thickness" to emphasize their three-dimensional structure. The raised display reveals that the glyph in the upper right is not a subglyph of the large glyph—instead, these two glyphs coexist at the same hierarchical level. By contrast, the glyph in the lower right is clearly piled on top of the larger glyph.

Similarly, any window previously saved to disk may be loaded back into the system. Not only does this provide for the saving and retrieving of previous work, but it furnishes an alternative to the "new glyph" button for creating glyphs in the first place. Windows, such as that shown in Figure 5.4, can be "preloaded" with sets of glyphs as desired. Such windows function as pallets, providing a source for the cloning of glyphs as needed. A lock mechanism on the window (the leftmost button in the command row of Figure 5.4) ensures that only clones, and not originals, can be dragged off of such a window.

Section 5.6 discussed the question of grouping, and introduced the concept of "trays": glyphs which function temporarily as common direct ancestors to other glyphs so they can be manipulated as a unit. Figure 5.5 exemplifies the use of trays to move two glyphs simultaneously without affecting the glyph they are piled upon. When the tray is created, all the glyphs sitting upon it are "frozen" to it. Later, when a tray is "melted", it is deleted automatically. Trays function much like the multiple-selection mechanisms found in many user interfaces. Unlike multiple-selection, however, many trays may coexist at once, effectively providing multiple

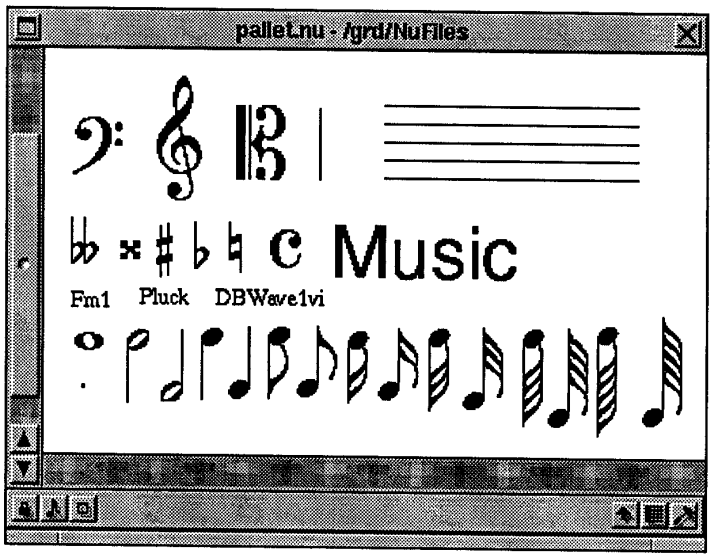


Figure 5.4: A pallet window. Pallets are ordinary glyph windows which have been "preloaded" to function as a source of glyphs for other windows through the cloning mechanism.

multiple-selections.

Every window always contains one particular glyph, called the "target" glyph, which is the object of a number of special commands. Whenever a glyph is moved, cloned, or simply selected by the mouse, it becomes the current target, and is distinguished on the display by being "shadowed", as exemplified by the staff glyph of Figure 5.6. As a special case, whenever the *rootGlyph* is the target, no shadow whatever will be displayed on the screen.

The target glyph is the recipient of a number of important messages; indeed, it is the "entry point" for messages to the glyph hierarchy. As an example, whenever a window's "play" button (second from the left in the command row) is pressed, a series of *tick* messages are sent to the target. Normally, part of the target's response to the message will be to relay *tick* to all those glyphs which are piled upon it, possibly activating the machine's synthesis hardware in the process. This response, however, is entirely determined by the corresponding *tick* method in the target's class definition,

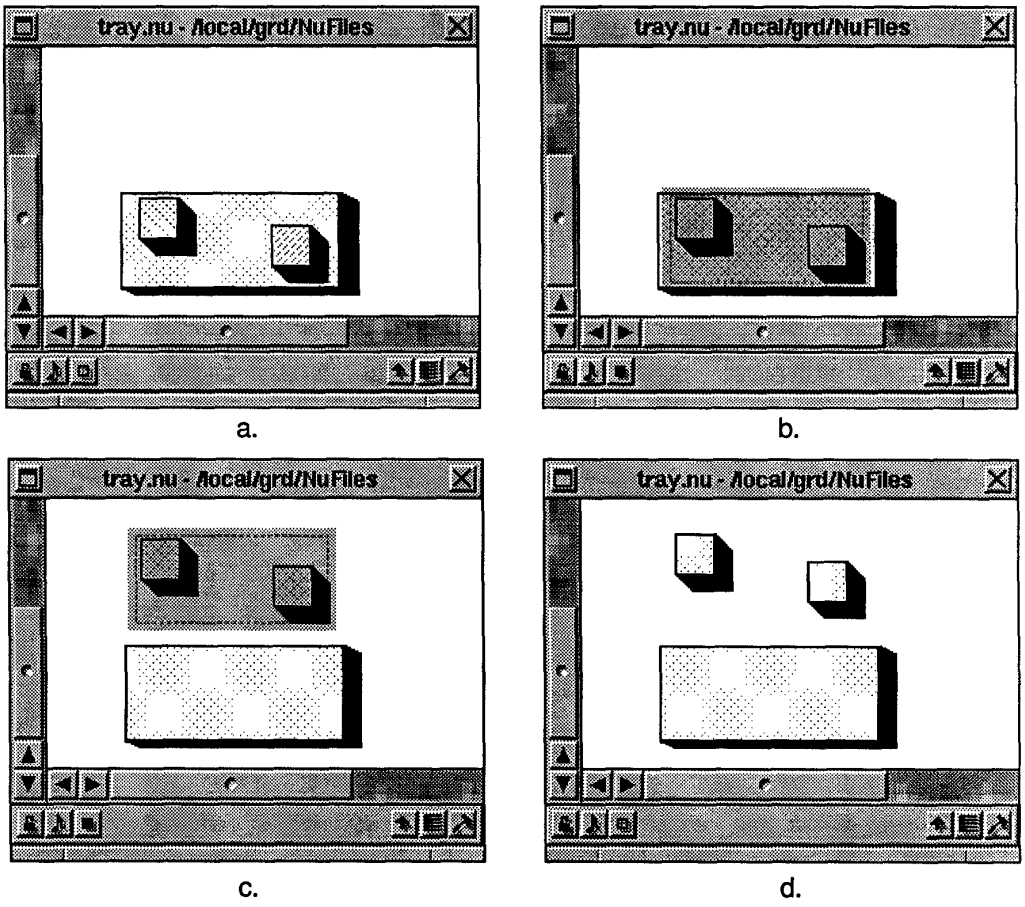


Figure 5.5: Trays. Suppose the user wishes to move the two small glyphs in window a off of the larger glyph, while preserving precisely the physical relationship between the two. To accomplish this, the user double clicks on the small glyphs, creating a new "tray" glyph and installing it as their common direct ancestor (window b). Note that the glyphs are "frozen" to the tray, as shown by the change to a solid rectangular icon in the "freeze" button. In window c, the tray unit has been moved off of the larger glyph. Finally, in window d, the tray glyph is melted, resulting in its automatic deletion.

and can be programmed by the user as desired.

Pressing the window's "browse" button (the rightmost button of the command row), sends a *browse* message to the target glyph. Typically, glyphs respond to this message by activating the second of Nutation's three window-types, called a browser window. As Figure 5.6 shows, browsers display two independent text editors. The top editor contains PostScript language code which defines the image the target glyph displays. Editing and saving this text will cause that image, together with the images of all other glyphs belonging to the same class, to change. The bottom text editor contains the method definitions defining the target glyph's behavior. When this text is changed, the Objective-C compiler is invoked to recompile the class, then the underlying Lisp subsystem is invoked to incrementally link the resulting compiled code back into the system.

The last of Nutation's window-types is called an inspector (Figure 5.7). Unlike the browser, which displays and changes the definitions of a glyph's class (i.e. its behavior and image), the inspector displays and changes the values of a glyph's instance variables (i.e. its state). Though there may be many browser windows on the screen at a time, there is only one inspector window. This window displays information about the current target glyph. Whenever a new target glyph is selected by mouse action, the information displayed by the inspector will change as well.

When the structure displayed in a glyph window becomes extremely complex, or when glyphs are particularly small, it can be difficult to pick out individual glyph using mouse action alone. For this reason, the inspector window provides facilities for walking through the hierarchy. Referring to Figure 5.7, the top row of buttons changes the current target, moving laterally through the hierarchy with the "then" and "precursor" buttons, moving toward the root via the "ancestor" button, and moving toward the leaves with the "is" button.

The second row of buttons on the inspector is used for resizing. Whenever the "Knobs" button is pressed, the target glyph is surrounded by a rectangle drawn with dashed lines, with three rectangular knobs around its perimeter, as shown in the glyph window of Figure 5.7. The leftmost knob only moves up and down—it is used for changing the height of the rectangle. The rightmost knob moves only left or right,

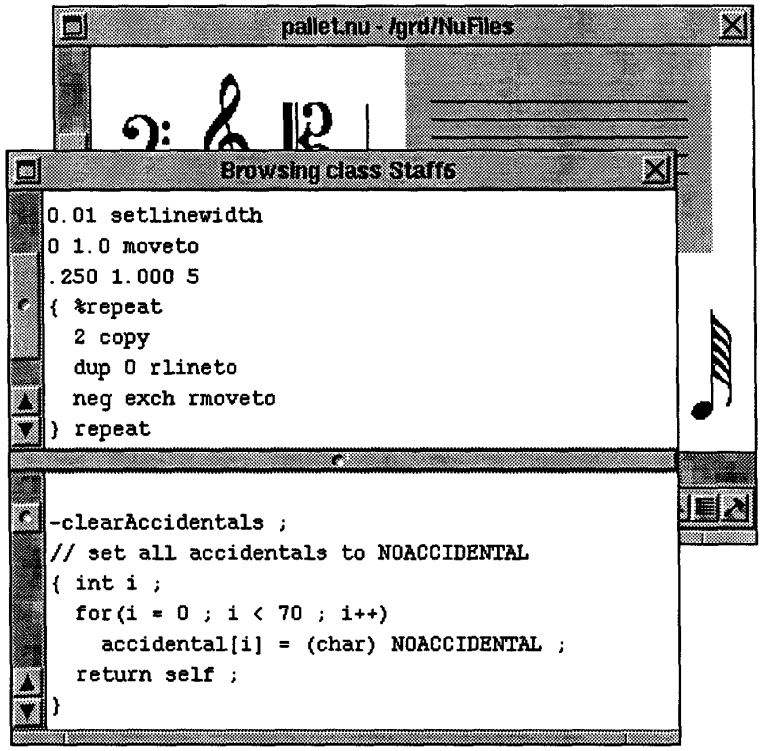


Figure 5.6: A browser window. The upper text view of the window contains PostScript language code defining the target glyph's image, while the lower view contains method definitions in Nutation's extended version of Objective-C.

and changes the rectangle's width. The middle knob, however, moves both horizontally and vertically, changing both the height and the width of the rectangle, while maintaining a constant ratio between these two dimensions. Pressing the "Accept" button causes the target glyph to be resized to fill the resized rectangle.

There are two fundamental ways in which this resizing may be computed. In the first case, not only the glyph, but any glyph sitting upon it is resized as well. Pressing the "Pile" button, however, turns it into a "Glyph" button, and subsequent resizes will change only the target glyph, without affecting any glyphs piled upon it.

Moving and resizing glyphs entirely through mouse interaction may not be the most desirable form of manipulation, especially when precise alignment is called for.

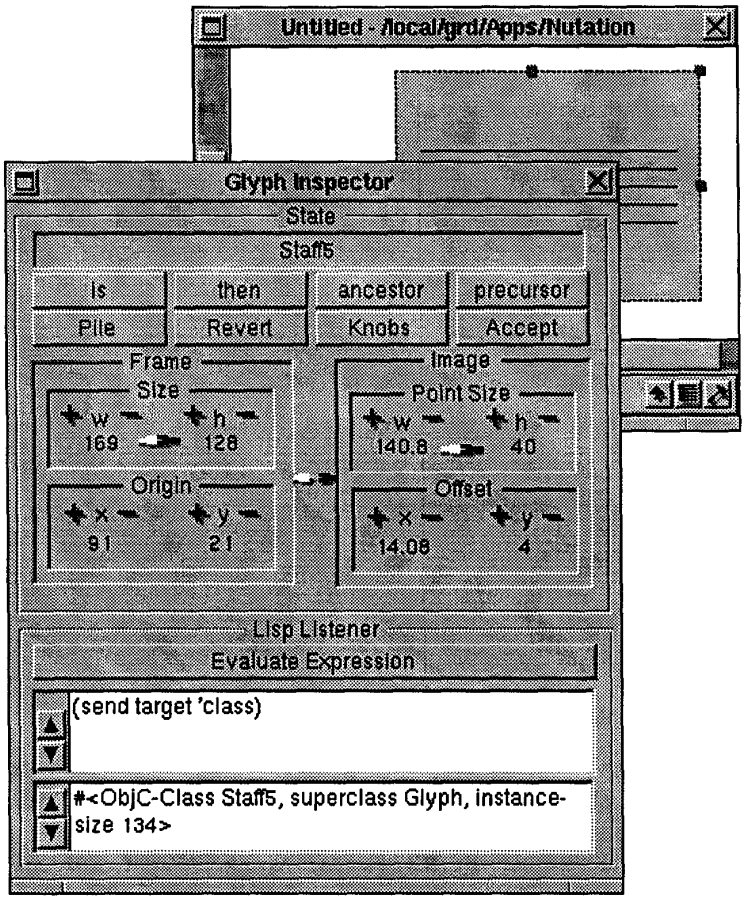


Figure 5.7: The inspector window. Most of the information in this window refers to the instance variables, or "state" of the current target glyph—in this case, an instance of class *Staff5* The dotted lines and small rectangular knobs surrounding the staff in the glyph window indicate that the staff is currently being resized.

The two panels in the middle of the inspector are provided for this purpose. They allow both the size and location of the glyph's outer boundaries, called the "Frame", as well as the image within that frame to be precisely specified. Three plug-like icons, called "couplers", are visible in this area. They determine the coupling between the various parameters of the frame and image. The frame's "size" coupler, for example, when plugged together, ensures that whenever the width is modified, the height is modified accordingly, and vice-versa. Clicking on the coupler "decouples" it, allowing the width and the height to be changed independently of each other.

The bottom window of the inspector allows access to the underlying Lisp subsystem. Utilizing Lisp's Objective-C interface, arbitrary messages may be sent to the current target glyph, to its ancestor, its precursor, and so on. Any response from the message is shown in the lower text view. Although any legal Lisp expressions may be evaluated here, its main purpose is to allow access to instance variables and methods defined in the user's own *Glyph* subclasses.

5.5 Three Glyph Sets

This section focuses on three of the notational systems Nutation is capable of supporting. It should be remembered, however, that these systems are not, strictly speaking, part of the program for, in the 3TG metaphor, particular styles of music notation are more akin to "data" than they are to "program". Indeed, this disassociation of the mechanics of the program from the particulars of any given notational style is an important aspect of the 3TG approach: it allows a single program to support a wide variety of notational styles, while offering a framework for implementing the user's own notational innovations.

In Section 3.7, the notion of *glyph sets* was introduced: sets of subclasses of class *Glyph* which define individual notational styles. This section will describe the implementation of three such glyphs sets for the Nutation program.

5.5.1 Piano-Roll Notation

Introduction

An important characteristic of Western art music in the twentieth-century has been its tendency toward novel rhythmic invention. Such composers as Elliot Carter and Pierre Boulez have extended the traditional vocabulary by calling for increased durational diversity and rhythmic precision in their works. At the same time, the opposite tendency towards less rhythmic precision and greater interpretive freedom is found in the works of such composers as Earle Brown and Luciano Berio. These new forms of rhythmic expression have led to the development of new forms of notation, including a group of techniques often referred to as "proportional", "spatial", or "time" notation.

For all their diversity, most proportional notation systems share at least two common characteristics. First, the onset time of a musical event is indicated by the horizontal location of that event's symbol on the staff. Second, the duration of the event is indicated by the horizontal length of the event's symbol. A good example of such a system which is currently very popular in computer music systems is called "piano-roll notation" (PRN). The system takes its name from its resemblance to the perforated rolls of paper used to record keystroke and sequencing information for the player pianos popular in the 1920's.

Implementation

PRN is a good starting point for understanding how glyph sets are implemented in Nutation, for only three *Glyph* subclasses are required. These three classes are all illustrated in Figure 5.8. Class *TimeLine* is the "staff" of the system, instances of class *Mark* are the musical events, while class *PluckIns* identifies the "instrument" to use in performing the passage.

Temporal information is measured in terms of units of time called "beats", marked by short black vertical lines along the bottom of the *timeLine*. Although the *timeLine* of Figure 5.8 represents a total duration of only seventeen beats, the inspector window's resizing feature allows it to be stretched horizontally to any desired duration. Pitch is represented along the vertical axis, where light gray lines serve as guides to

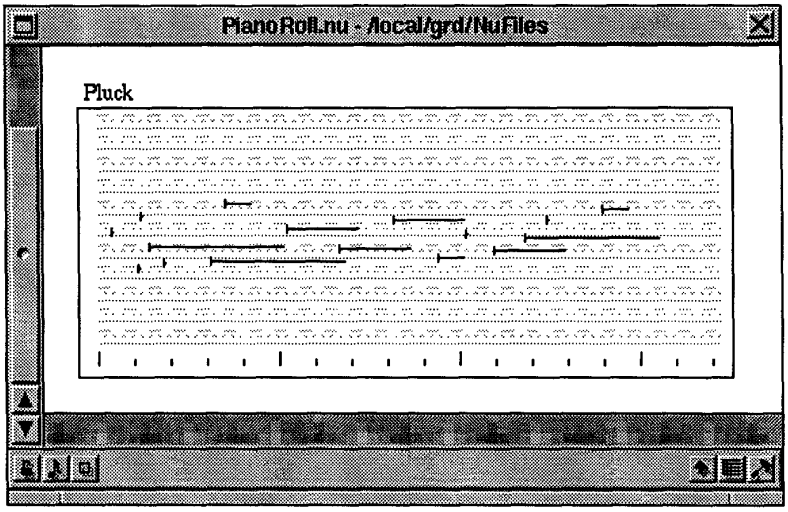


Figure 5.8: A passage in piano-roll notation.

locate pitch more precisely. Solid gray lines locate the pitch-class C, with C0 at the bottom. Dotted lines locate E, while dashed lines locate G. A number of mark glyphs have been piled on the timeline. These *marks* have been stretched to various lengths (and therefore various durations) using the inspector window's resize facility. Finally, a *pluckIns* glyph, labelled "Pluck", identifies the Music Kit synthesizer patch that Nutation will use in performing the passage.

It is instructive to trace through some of the mechanism whereby this passage is performed on the underlying synthesis hardware. Glyphs never communicate directly with this hardware; instead, they relegate the task to objects from NeXT's Music Kit. Every *mark*, for example, contains an instance of the Music Kit *Note* class. These *notes* must be assigned both a pitch and a duration value before they can be played. In response to the message *scan*, *marks* determine the appropriate pitch value for their *notes* by messaging the *timeLine* they are piled upon, passing their current location as an argument. The *timeline*, in turn, responds with the pitch value corresponding to that particular location. The *mark* then uses this information to set its *note's* pitch value. Duration is determined in a similar way, using the *mark's* width in pixels and the *timeLine's* notion of the number of beats per pixel to compute

the value.

In Nutation's internal TTree data structure, all the *marks* on a *timeLine* are organized as a single list of "then"-related objects.[6] Furthermore, the *timeLine* is left-to-right ordered, that is, it ensures that the order of the the *marks* (or any other glyph, for that matter) on this list corresponds to their left-to-right ordering on the screen. In other words, the leftmost *mark* on the screen will be the first element on the list, the next-to-leftmost *mark* will be the second element, and so on. Before they can be performed, each *mark* must set its "wait" value: the time delay (which may be 0) from itself until the next *mark*. In response to the message *scan*, this value is computed by multiplying the distance in pixels between successive *marks* by the *timeLine's* notion of the number of beats per pixel.

The actual performance of the passage takes place when the user presses the "play" button. This sends a single *scan* message, then a series of *tick* messages to the current target glyph, from whence they are relayed to the rest of the structure according to the TTree performance algorithm.[7] When a *mark* first receives a tick message, it messages the *timeLine* it is piled upon, sending its Music Kit *note* object along as an argument. From information stored in the *pluckIns* glyph, the *timeLine* knows which Music Kit *instrument* (called a *synthpatch* in the Music Kit's vocabulary) to use in realizing the *note* . It then passes the *note* to that instrument, causing the Music Kit to actually synthesize the event.

Many elaborations to this PRN scheme are possible. The goal of this implementation, however, was to present the mechanisms underlying glyph sets as clearly as possible. Although the next two sets to be presented are considerably more complex, most of the basic mechanisms involved are identical.

[6]See Chapter 3 for an explanation of terminology associated with the TTree data structure.
[7]Nutation currently uses Method 8 of Chapter 3, the "message blocking" algorithm.

5.5.2 Common Music Notation

Introduction

Despite a good deal of regional and historical variation, common music notation (CMN) has been an uncommonly well-standardized notation system for most of the past three hundred years. Byrd defines CMN (which he refers to as *conventional music notation*) as follows:

> ...any arrangement of the symbols in general use by computers in the European art music tradition from about 1700 to 1935, used with the meanings which were standard: (1) if the notation was made between 1700 and 1935, at the time it was made; (2) if the notation was made before 1700, with the meanings of 1700; or (3) if the notation was made after 1935, with the meanings of 1935.[Byr84, page 13]

There are many reasons for implementing CMN in Nutation. The system is well-understood by a large segment of the musical world, and much of Western musical culture has been conceived with it as a tool. A substantial amount of contemporary music in all genres is still entirely expressible in the system. Most notational reforms and extensions in use today take CMN as their starting point. Finally, because the system is both rich and complex, it makes an excellent test of Nutation's, and, by extension, the 3TG model's representational capability.

Staves

Class *Staff* provides services similar to those provided by class *TimeLine* in the PRN system described in the previous section.[8] Staves determine the pitch of the notes which are piled upon them on the basis of both the location of these notes and on the location and type of any clefs which are present. When a glyph representing a Music Kit instrument (such as PRN's *pluckIns* glyph) is piled on a *staff*, then the pile can be performed by the underlying synthesis hardware.

[8]This class is actually called *Staff5* by the system in order to distinguish it from other types of staves with more or fewer lines.

Like *timeLines*, *staves* are left-to-right ordered, and they always have a "wait" value of 0. As Figure 5.9 shows, this means that if a number of *staff* glyphs (each piled with clef, note, and instrument glyphs) are piled on the *rootGlyph*, they will all play at exactly the same time, regardless of how they are arranged physically on that *rootGlyph*.

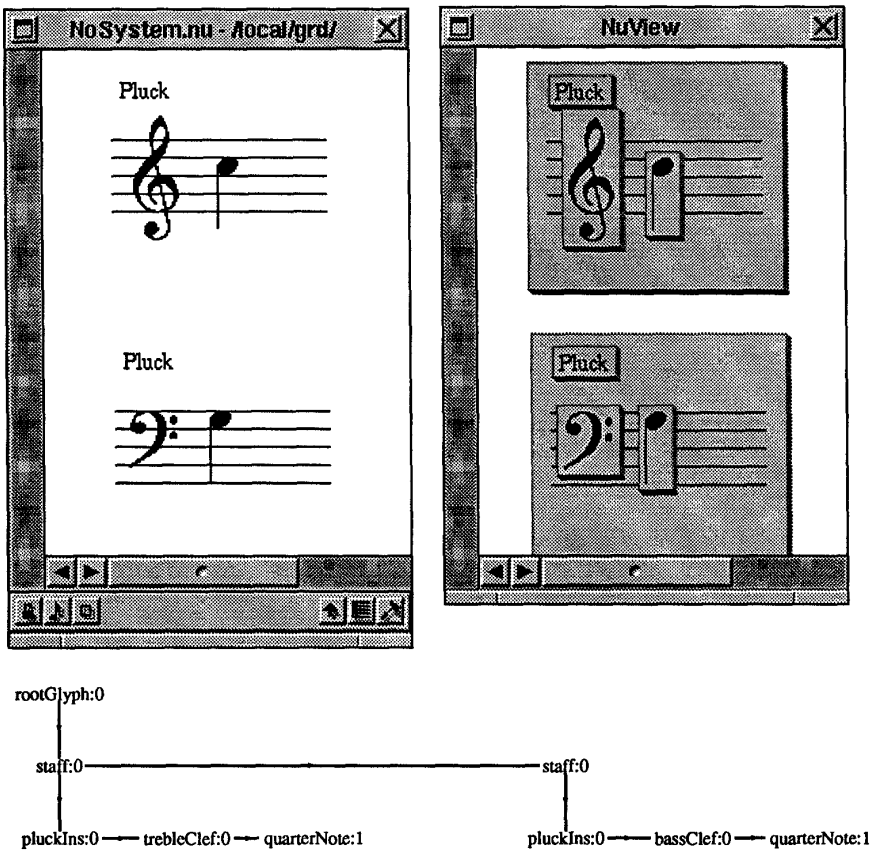


Figure 5.9: Staves. The window contains two *staves* piled on the *rootGlyph*. A raised view of the passage is shown for clarification. Since the staves have a 0 "wait" value, they will be performed concurrently, as indicated by the corresponding TTree.

Systems

Class *System* provides the functionality for sequential execution of *staves*. *Systems* determine the duration of all the glyphs which are piled upon them, then set their "wait" value accordingly. Unlike *Staves*, *rootGlyphs* are top-to-bottom ordered. As Figure 5.10 shows, whenever a number of *system* glyphs (each piled with stave, clef, note, and instrument glyphs) are piled on the *rootGlyph*, they will be played sequential from the top of that *rootGlyph* to the bottom.

Notes

Nutation provides individual classes for notes of all durations from the whole note down to the 128th note. In addition, all these classes (with the exception of class *WholeNote*) come in pairs: one version for the ascending stems, another for the descending ones. Much of the functionality of these fifteen classes is identical—in fact, they differ only in their notion of their own duration and in the image which they display.[9] Obviously, it would be a great waste of system resources to reproduce their common functionality in each class. Accordingly, an abstract superclass,[10] called class *_Note*, is provided for this purpose.[11]

Unlike the *marks* of the PRN system, where "wait" values were determined by the horizontal separation between successive *marks*, *_notes* have an intrinsic, default wait value. Class *QuarterNote*, for example, always sets this value to one beat.[12] As one might expect, piling an instance of class *Dot* on a *quarterNote* increases this value to 1 1/2 beats, while piling an additional *dot* onto the first dot increases it to 1 3/4 beats, and so on.

[9]The images of all notes, as of most of the glyphs in this CMN implementation, utilize the PostScript *Sonata* font from Adobe Systems Incorporated.

[10]An abstract superclass is a class which has no instances. It exists solely to provide common functionality to its subclasses through inheritance.

[11]Nutation adopts the convention that the name of any class which the user does not create instances of (such as an abstract superclass) begins with an underbar character.

[12]This assignment of one beat to the quarter note is only incidently related to the conventional one-beat duration of a quarter note in, say, 3/4 time. A QuarterNote's wait value could just as well have been a half a beat or, for that matter, 10 beats. All that matters is that it be twice as long as an eighth note, half as long as a half note, and so on. The actual duration of a beat is a Music Kit parameter, and can be independently controlled by the user.

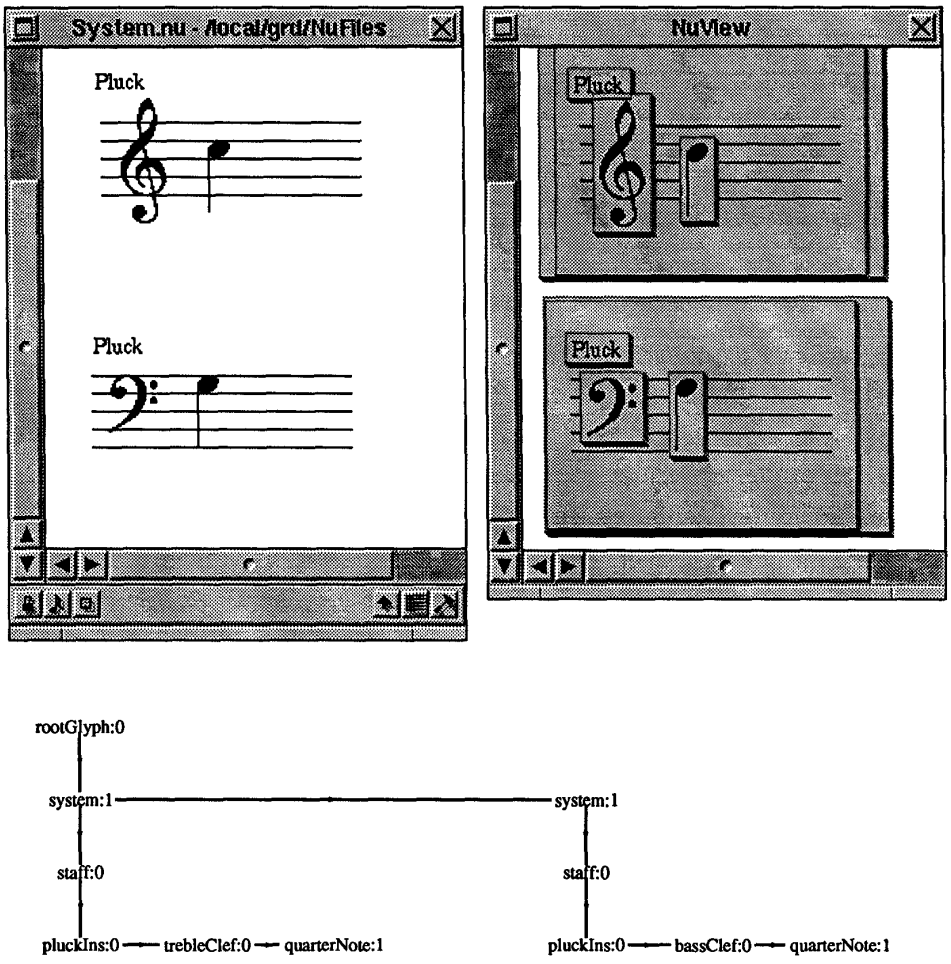


Figure 5.10: Systems. The window contains two systems piled on the rootGlyph. Because systems display only a transparent surface, they are only visible in the raised view on the right. Since the first system has a"wait" value of 1 beat, the systems will be performed sequentially, as shown by the corresponding TTree.

This implementation does very little in the way of automatic formatting. Although such formatting is certainly feasible in the 3TG approach, it is not among the goals of the implementation. Nevertheless, a certain amount of it greatly simplifies the user's task in working with the system. Instances of class *_Note*, for example, are automatically relocated to the nearest line or space whenever they are piled onto a *staff*. Instances of Class *_Clef* and Class *_Accidental* (the abstract superclasses from which all clefs and accidentals inherit) also behave in this way. *Barlines* automatically align themselves to the bottom and top lines of whatever *staff* they are piled upon.

Ledger Lines

One approach to implementing ledger lines in Nutation would be to provide "ledger" glyph subclasses displaying one, two, three, four, and so on, ledger lines. Whenever a ledger line is needed, the user could pile the appropriate ledger line glyph on the staff, then pile its note on top of the ledger. Such an approach would, of course, render any passage with extensive use of ledger lines extremely tedious to enter, let alone modify. Consequently, instances of class *_LedgerLine* are provided automatically by the implementation, appearing wherever and whenever the conventional syntax of CMN requires them.[13]

Figure 5.11 shows two *_notes* together with their corresponding *_ledgerLines*. As the raised view to the right reveals, all *_ledgerlines* always display an image of six lines. When fewer than six lines are required, the extra lines are effectively made invisible by laying them over the *staff*, where they will simply duplicate some portion of the image of this *staff*. In this way, situations requiring up to six ledger lines can be handled with a single glyph class.

Figure 5.11 also reveals a surprising implementation detail. Rather than piling *_notes* onto their associated *_ledgerLines*, a *_note* and its *_ledgerLine* coexist at the same hierarchical level. Either implementation is possible, however, for *_ledgerLines* are purely visual aids, adding no functionality whatever to class *Glyph*. Keeping

[13]Like the abstract superclasses *_Note* and *_Clef*, class *_LedgerLine* begins with an underbar. It is not an abstract class, however—the underbar indicates that the user never creates instances of the class (they are created and deleted automatically by subclasses of class *_Note*.

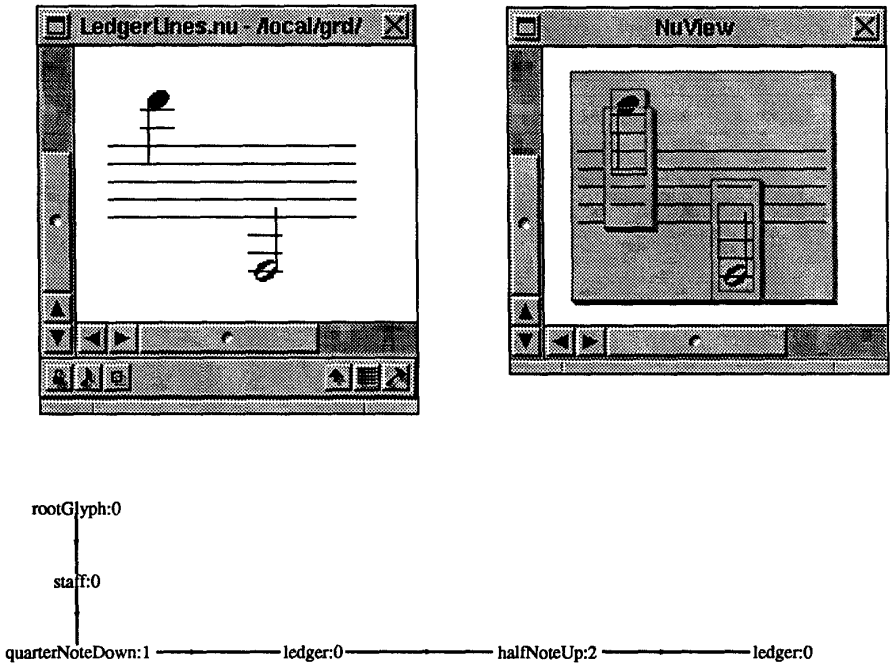


Figure 5.11: LedgerLines. Note that *_ledgerlines* always display 6 lines. As the corresponding TTree shows, *_ledgerLines*exist at the same hierarchical level as the notes they are associated with.

_notes and *_ledgerLines* on the same level, however, avoids any clipping of the *_note's* image which might otherwise take place.

Accidentals and Key Signatures

Accidentals are implemented as subclasses of the abstract superclass *_Accidental*. These subclasses differ from one another only in their response to the message *value*. A *sharp*, for example, will respond with the integer 1 ("raise" a semitone), while a *flat* will respond with -1 ("lower" a semitone).

Recall that *_Notes* have no intrinsic notion of their own pitch. Instead, in response to the "scan" message, they determine this pitch by messaging the *staff* they are piled upon, passing their screen location as an argument. Subclasses of class *_Accidental*

also respond to "scan" by messaging their *staff*. They cause the *staff* to alter its notion of the pitch value of any given location, thus affecting the interpretation of any subsequent *_note* at that location. As might be expected, *barLines* cause the *staff* to forget these alterations.

Accidentals which appear in key signatures have a larger scope than ordinary accidentals, for they alter not only a single pitch, but an entire pitch class. Furthermore, they have an entirely different scope: unaffected by barlines, only another key signature can redefine an existing one. In order to properly provide this functionality, the implementation must be able to determine whether or not a given accidental is part of a key signature. Class *KeySig* is provided for this purpose. As Figure 5.12 shows, a *keySig* is completely invisible, for its image is nothing more than a transparent surface. *KeySigs* exist solely to provide a place to pile *_accidentals*, distinguishing

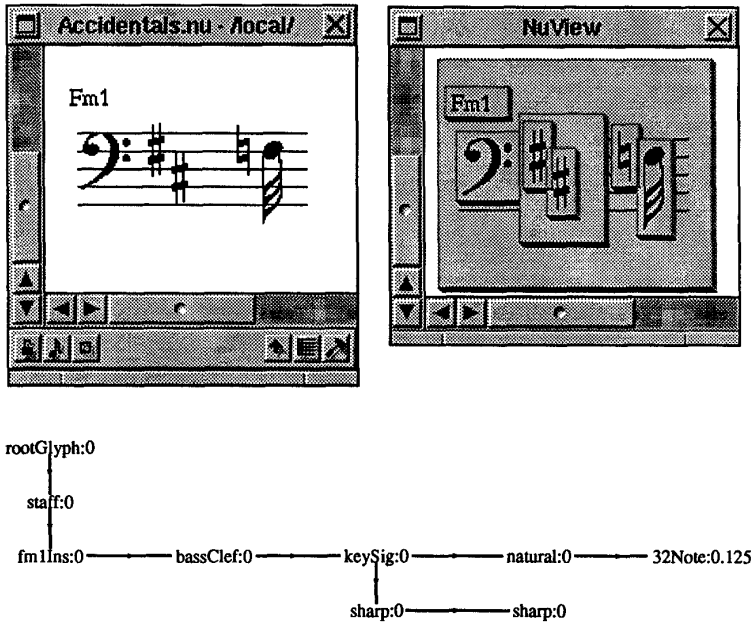


Figure 5.12: Accidentals. Since accidentals display only a transparent surface, they are completely invisible, except in the raised display.

Beams

Class *Beam* is the sole glyph class in this implementation to use the "unique image" feature introduced in Section 5.4.1. Rather than sharing the same image among all its instances, each individual *beam* glyph draws its image on the basis of whatever glyphs are piled upon it.

Whenever the current target glyph is a *beam* and the "freeze" button is pressed, the *beam* computes and draws the stems and beams as appropriate, then messages all the glyphs piled upon it to redraw themselves. In addition, *beams* affects the way in which some of these glyphs will redraw. Subclasses of class *_Note* (other than class *WholeNote*) actually have a second image, called their "altImage", consisting solely of a note head with no tail. Whenever a *_Note* is "frozen" to a *Beam*, it will display this altImage.

Figure 5.13 outlines the process of creating a beam structure from the user's point of view. First, a *beam* glyph is obtained, and *_notes* (or, for that matter, any glyph subclass) are piled on top of it. Freezing the *beam* causes the entire pile to be redrawn as a beam structure. While frozen, the individual glyphs which make up the pile cannot be moved. When the pile is "melted", however, the beam structure disappears and the stems of the *_notes* reappear. The individual glyphs which make up the pile may then be manipulated as needed.

Multi-part Music

The notation of homophonic and polyphonic music is problematic for notation programs which provide automatic score playing, and is especially difficult when determining the identity of individual parts is at issue. This section introduces solutions, based on the 3TG approach, to three different notational conventions in multi-part music: several simultaneous single-part staves, multiple notes on the same note stem, and several independent polyphonic lines on the same staff.

The first case, that of several simultaneous single-part staves, requires no special mechanics to achieve. As we have seen, *staff* glyphs have a "wait" value of zero. This means that whenever several *staves* are piled onto a *system*, they will all play

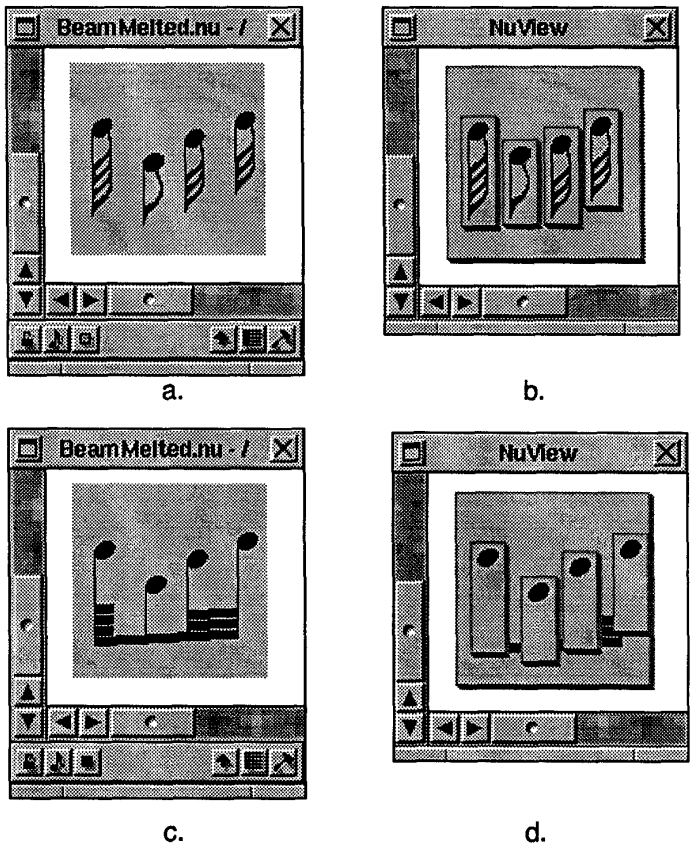


Figure 5.13: Beams. In panel a, several *_notes* have been piled on a *beam*. The *beam* is the target glyph in this case, as indicated by its gray shadow. In panel c, the pile has been frozen (note the change from an empty rectangle to a solid rectangle in the "freeze" button), causing it to redraw itself as a beam structure. The raised view, panel d, reveals that *_notes* on a frozen *beam* display only their note heads.

simultaneously. Figure 5.14 presents an example of this style of notation.

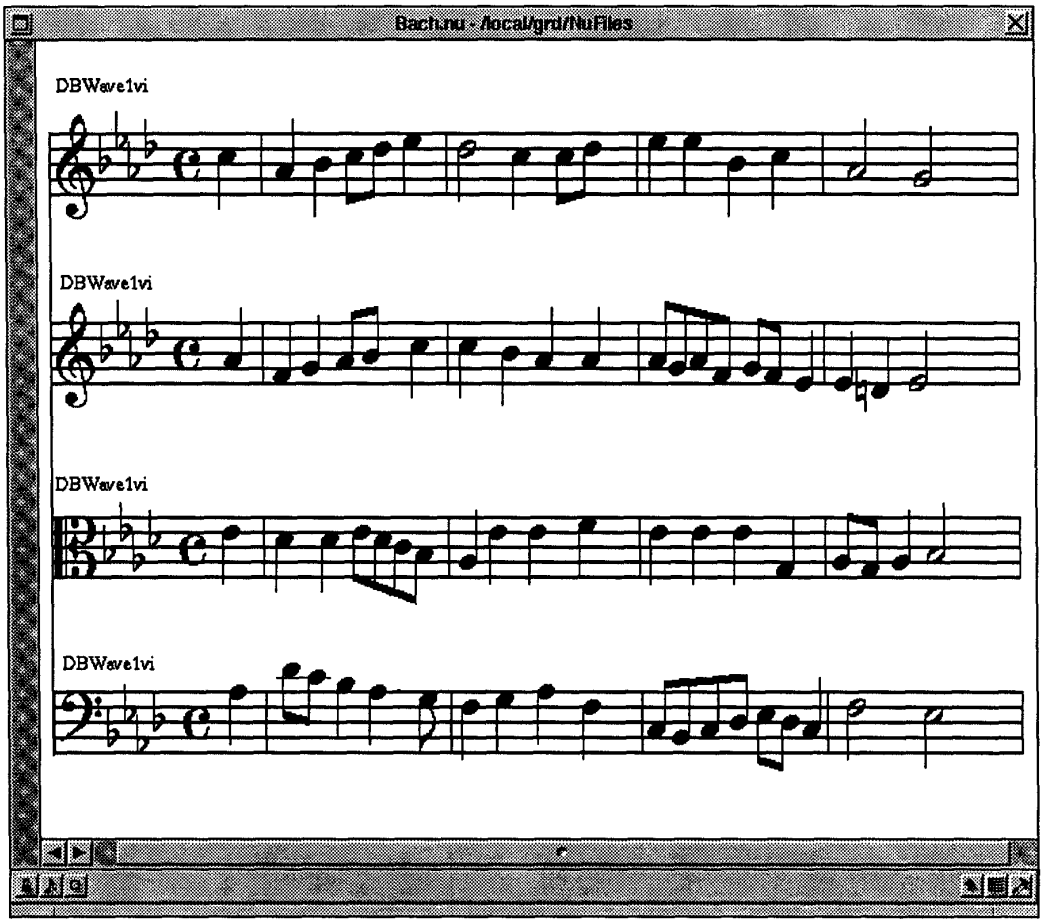


Figure 5.14: Simultaneous single-part *staves*.The excerpt is from the Choral *Ich bin's, ich sollte büßen'*, from J.S. Bach's *St. Matthew Passion*.

The second case, multiple note heads sharing the same note stem, is almost as simple to implement as the first case. As Figure 5.15 demonstrates, *_notes* can simply be piled on top of one another with their stems coincident. The TTree in this figure shows that all these *_notes* are connected to each other by vertical, "is" branches. Since "is" branches have no time delay at all, all these *_notes* will play simultaneously.

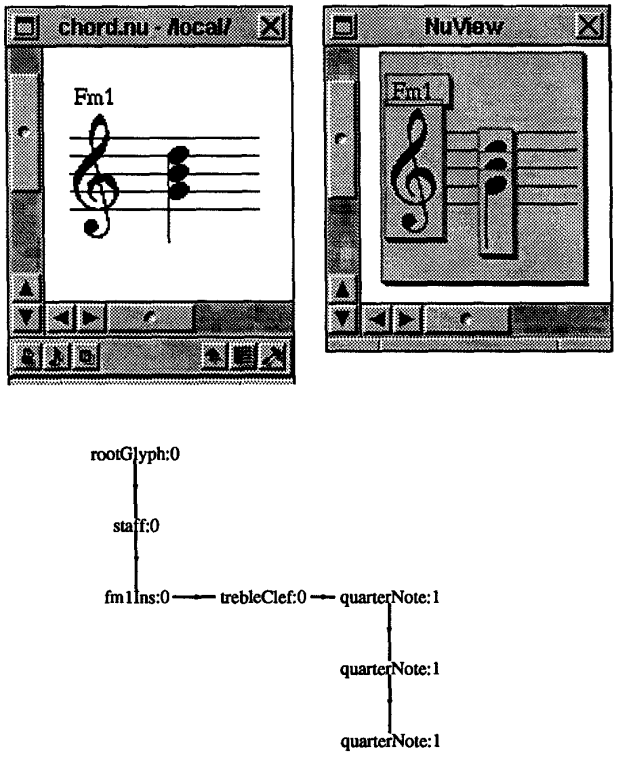


Figure 5.15: Chords. As the raised view indicates, multiple notes on the same stem can be implemented by simply piling *_notes* on top of each other. The corresponding TTree shows that these *_notes* are all joined by vertical, "is" branches, and thus they will all play simultaneously.

Unfortunately, this method is only convenient for notes which have no flags, such as half notes and quarter notes. Notes with flags need to be resized, using the inspector, so that their flags are invisible. An alternative, not implemented in Nutation, would be to have such notes display their "altImage", as they do whenever they are frozen onto a *beam* glyph.

The situation is considerably more complex when multiple independent parts are to be placed on a single staff. Byrd identified the major problem areas as follows:

> Two voices sharing a staff are basically accommodated by some simple modifications of the standard rules. For example, stems for the upper

> voice are always pointed up and for the lower voice down, and slurs, ornaments, and groupet accessory numerals all move to the outside. Many problems can arise, however. For example, the voices can cross, so that the upper voice is temporarily lower in pitch; in this case interference between symbols in the two voices, requiring some repositioning, is quite likely. (Bach's *Six-Part Ricercare* from *The Musical Offering* is a spectacular example: it has three voices per staff almost continuously). In fact, *variable* numbers of voices on a staff and voices moving from one staff to another are not at all unusual; these occurrences sometimes leads [sic] to ambiguities when fewer than the maximum number of voices are present.[Byr84, page 40]

When Byrd wrote these lines, he was primarily interested in the music *setting* problem, and the attendant issues of automatic formatting. Such formatting is not among the goals of this implementation, however, and is left almost entirely up to the user. Nevertheless, many of the same problems confront any system which attempts to play such a multi-part texture, for in both cases the system must be able to sort out which notes belong to which part.

This implementation provides a class *PartQ* for exactly this purpose. These *partQs* display no image at all—their role is simply that of holding a single musical part. Whenever a partQ is piled onto another partQ, it is linked into the TTree as that partQ's "is" glyph, forming a chain of vertical, "is" branches in the TTree. Furthermore, all partQ's in this chain are automatically aligned on the screen so that their origins are at the same point.

Figure 5.16 shows the process of creating a multi-part, single staff passage from the user's point of view. First, separate parts are created, as shown in windows a and b. Normally, these parts would be constructed over top of a *staff*, though the staff has been omitted from this figure for clarity. The parts are then piled successively on top of a staff, as shown in frame c. The corresponding TTree shows that whenever a *partQ* is piled onto another *partQ*, it is always attached as its "is" glyph, displacing any other "is" object.

This implementation scheme allows a virtually unlimited number of parts to be

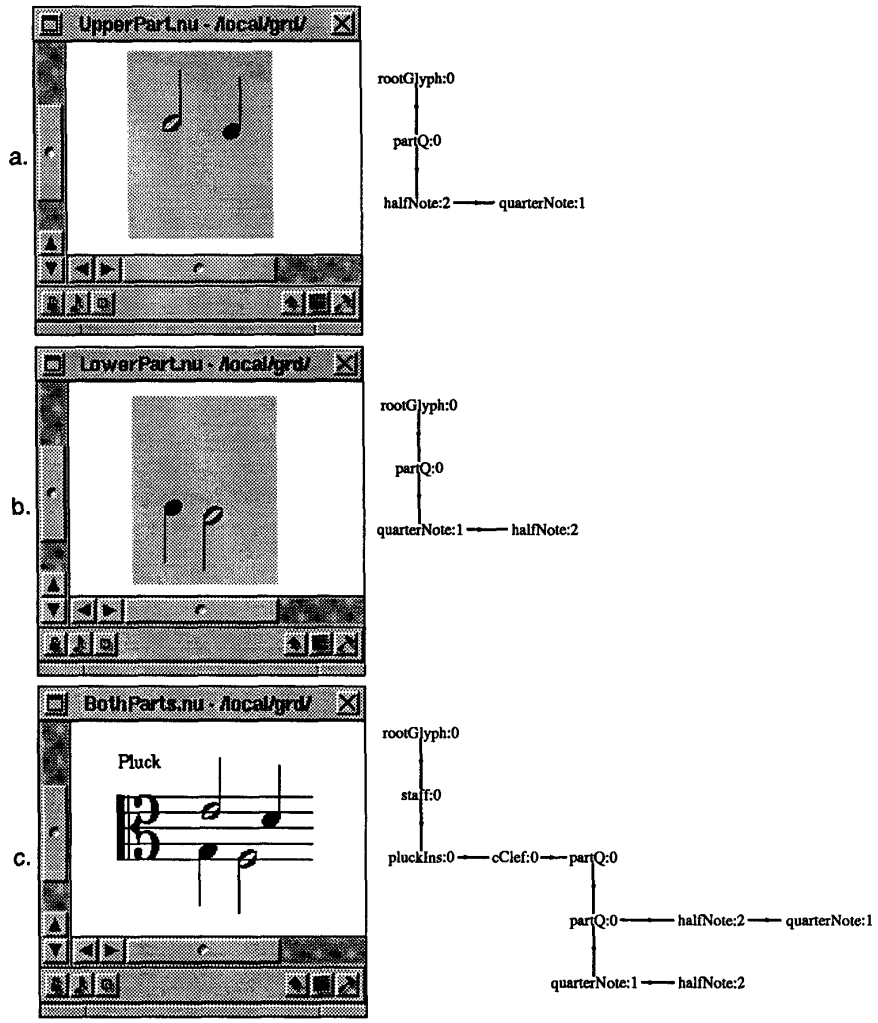


Figure 5.16: Multiple parts on the same staff 1. Since *part* objects display nothing, they are completely invisible. In frames a and b, they have been made the current target glyph in order to show their location.

displayed on the same staff, while making it obvious to the system which notes belong to which part. Unfortunately, this is not necessarily visually obvious to the user. Furthermore, multiple *partQs* on a *staff* form a "stack" data structure, and only the topmost one can be accessed by the user at any one time. Editing any other part involves the unpiling and subsequent repiling of parts lying on top of it. Although several schemes for turning this stack into a "deque" or "queue" were attempted (the "Q" in the class name *PartQ* is a remnant of one of these attempts), nothing entirely satisfactory was found.

This section concludes with a more complex example of multi-part music. Figure 5.17 shows the opening measures of J.S. Bach's *Chaconne* from the *Partita in D minor* for solo violin. The passage features as few as two and as many as four separate polyphonic voices at any one time, often changing the number of voices in the middle of a measure. The implementation uses four *partQ* glyphs throughout. When fewer than four parts are required, invisible rests could have been inserted. In this example, however, notes in the extra parts simply double existing notes in the required parts. By placing these doubling notes directly over top of each other, the extra parts effectively disappear from the display.

5.5.3 Okinawan Notation

Introduction

Okinawan notation (ON), a tablature system for the three-stringed, lute-like *samisen*, was invented in the eighteenth century and extended in the 1920's to include the complex, highly ornamental Okinawan singing style. La Rue, in his 1951 study of the system, remarked that

> "While it is now nearly six years since I came across the Okinawan notation, I shall never forget my naive surprise that music could be effectively written down in a manner so utterly different from our own." [Rue51, page 34].

The Okinawan system contrasts markedly with both PRN and CMN. The need to demonstrate the 3TG model's ability to support a wide variety of notational styles

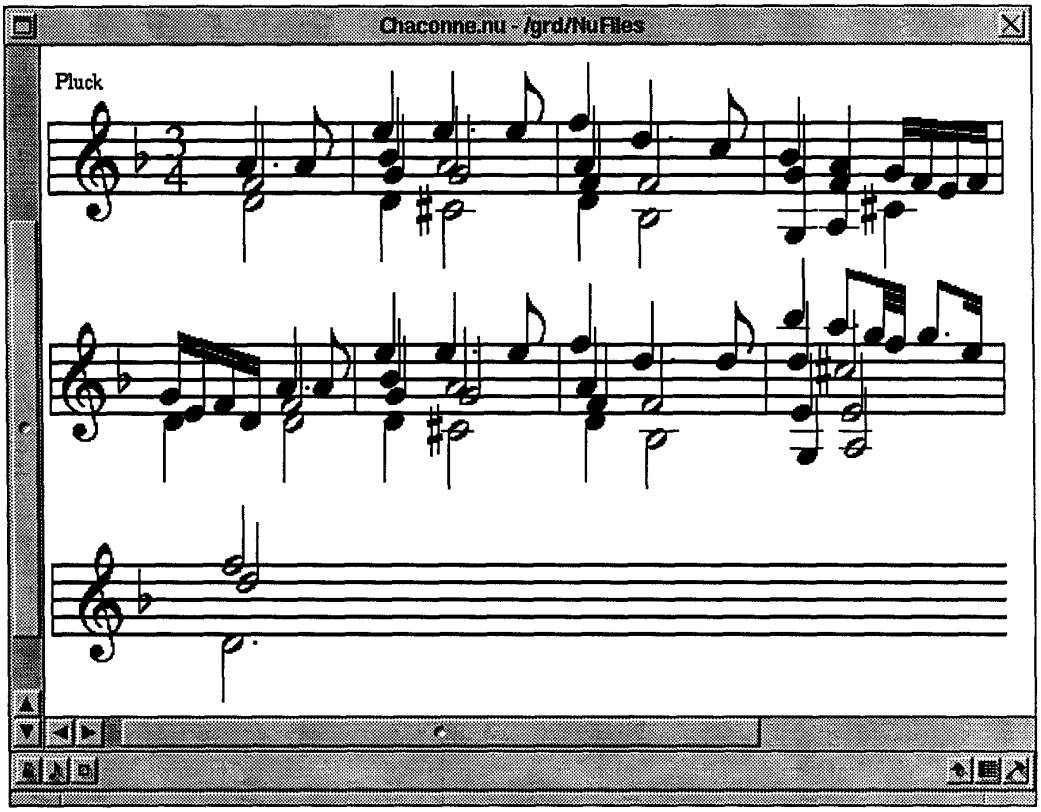


Figure 5.17: Multiple parts on the same staff 2. The excerpt shows the opening measures of J.S. Bach's *Chaconne* from the *Partita in D minor* for solo violin.

together with the visual beauty of the Okinawan system made it an ideal candidate for implementation in Nutation.

The Notation System

Since it is relatively unknown in Western cultures, a brief explanation of ON will be presented here. The reader is referred to La Rue's article for further details [Rue51].

ON makes use of a rich variety of symbols. Pitch is represented in a manner similar to German lute tablature, with individual characters provided for each string-fret combination on the samisen. The actual interpretation of these symbols is dependent on the instrument's tuning which, in addition to its conventional form, uses three

types of scordatura. The vocal line uses the same set of symbols, with the pitches determined from the corresponding samisen pitch. In order to represent the profusion of ornamentation characteristic of the vocal style, a wealth of ornamental symbols are used.

In CMN, musical event duration is determined primarily by the shape of individual symbols, whereas sequence and pitch are determined primarily by those symbols' location on a staff. In ON, the situation is reversed: pitch is indicated by symbol shape, with sequence and duration indicated by staff location. The staves of the system (see Figure 5.18), which are read right-to-left, top-to-bottom, are composed of two separate columns: one for the samisen, one for the voice. The rectangles which divide the samisen column can be thought of as representing a quarter-note duration each. Eighth-note subdivisions are notated by placing symbols on the line, while the symbol resembling the letter "O" is a rest. The rhythm of the samisen part in the figure, then, is quarter/eighth/eighth/quarter/quarter rest.

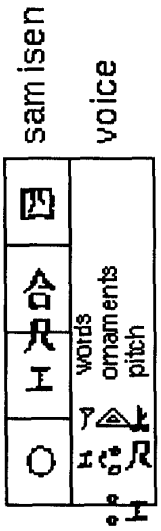


Figure 5.18: The "staff" of Okinawan notation.

Running parallel to the samisen column is the voice part. This part is further subdivided into three columns: the right-most presents the pitch symbols, the middle

column the vocal ornaments, while the text to be sung is presented in the leftmost column. Rhythm in the voice part is keyed to the rectangles in the samisen column.

Implementation

The symbols displayed by most of the glyphs in this implementation were obtained by digitally scanning illustrations found in La Rue's article, then converting the results into PostScript bitmaps. The number of glyph subclasses needed was surprising: no fewer than 65 classes were created, compared to 44 for CMN and 3 for PRN.

Since ON reads from right-to-left, a special glyph, called an *ONPage* glyph, is used to override the *rootGlyph's* inherent top-to-bottom ordering. As Figure 5.19 shows, this glyph displays the title "Okinawan Notation" in this implementation. The staves of ON are represented by the class *ONGrid*. Since the pitch of any symbol is independent of its staff position, class *ONGrid* is somewhat simpler than CMN's class *staff*. Two further glyphs classes, called *ONTrack* and *ONVoiceTrack*, are used to to distinguish between the samisen column and the voice column.

As we have seen, the voice column of ON is further divided into separate columns for pitches, ornaments, and words. Since words and ornaments never occur without a pitch, however, no separate glyphs were needed to represent these columns. Instead, what might be rather loosely called a "note" can be constructed by piling the ornaments and words onto the pitch glyph. Figure 5.20 shows such a note pile. Typically, there will be several ornaments and perhaps several words attached to each pitch glyph.

Pitch glyphs (subclasses of the abstract superclass *_ONsf*, for Okinawan Notation string-fret) are very similar to the *_notes* of the CMN implementation. They each have a music kit *note* object which they use to interface with the synthesis hardware during performance. Ornaments piled on a pitch glyph cause amplitude and frequency envelopes to be applied to the *note* object, imitating the frequency scoops and dynamic variations of Okinawan vocal style. Word glyphs, in this implementation, are confined to single vowels. They tell the corresponding *note* how to choose from among a predetermined set of presampled waveforms during synthesis.

Like the voice part, the samisen part may contain ornamentation, though this

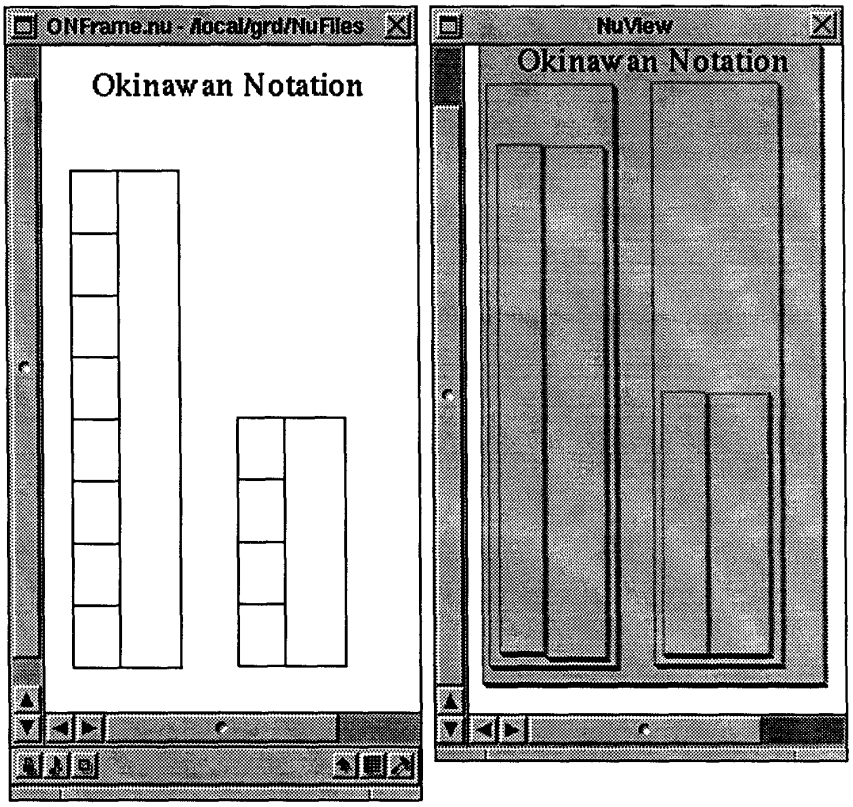


Figure 5.19: The Okinawan notation framework.

is never as elaborate as its vocal counterpart. Small teardrop-shaped glyphs piled onto pitch glyphs trigger these ornaments. Rather than programming the mechanics necessary to have these ornaments perform themselves, it proved more convenient to take advantage of the ability of clipping to suppress detail. As Figure 5.21 shows, ornaments can be written out in full (in ON, of course!), piled onto the tiny tear-drop shaped ornament glyph (which will completely clip them from view), then piled onto the pitch glyph. The ornament glyph itself has no unique functionality whatsoever.

This section concludes with an example, Figure 5.22, of a complete ON composition. The work, which is taken from La Rue's paper, is not a traditional Okinawan

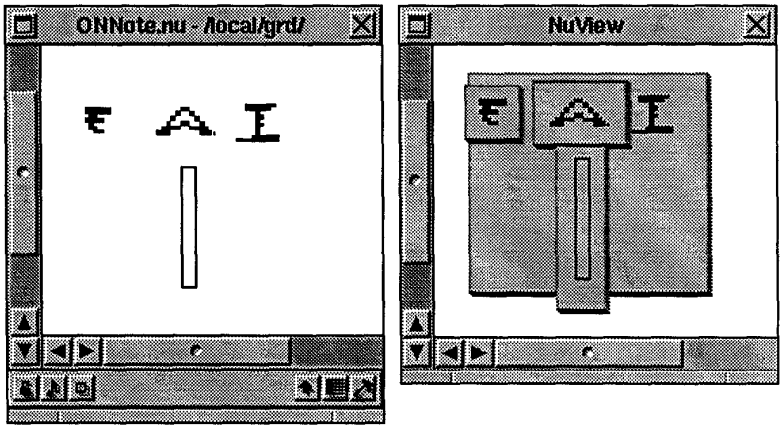


Figure 5.20: The structure of an Okinawan "note". This pile consists of a pitch glyph, two ornamental glyphs, and a word glyph.

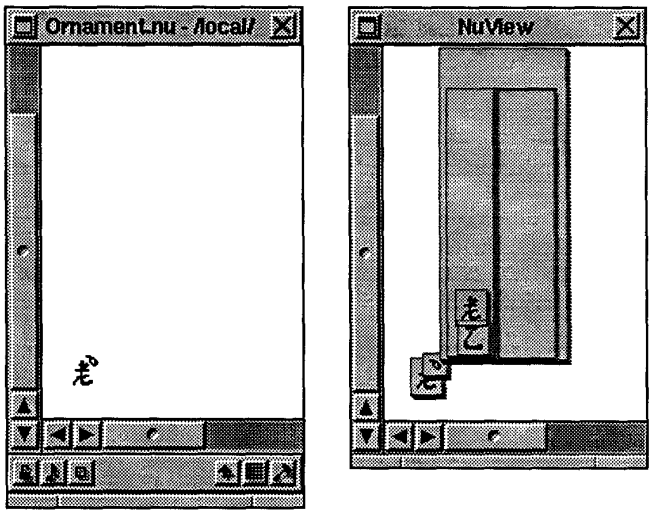


Figure 5.21: A samisen ornament. As revealed by the raised display, the tiny teardrop shape conceals the ornament in the form of a fully-written out passage of ON.

piece. Instead, it was written specifically to demonstrate as many facets of the notation system as possible in a single work.

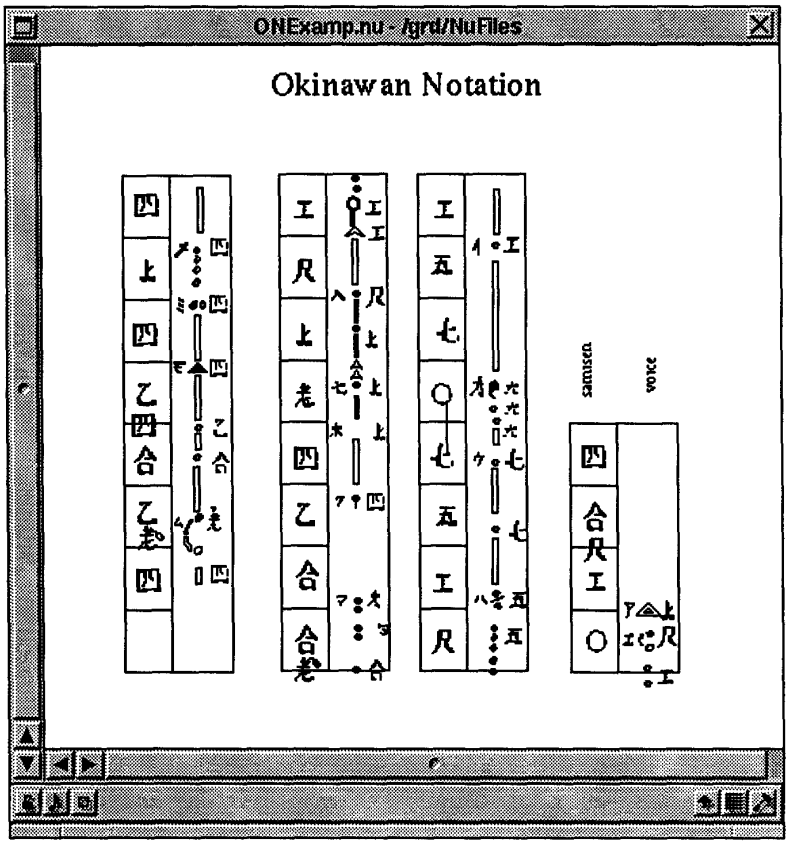


Figure 5.22: A composition in Okinawan notation.

5.6 Summary

This chapter has described a computer implementation of the 3TG model. The system, dubbed *Nutation*, was built primarily with the goal of supporting claims made about the model throughout the dissertation.

Nutation takes the form of an object-oriented, visual programming language, and

provides both an integrated programming environment for the development of glyph subclasses and a direct manipulation system for manipulating instances of these subclasses. Three very different music notation styles, implemented as "glyph sets", were presented: a simple piano-roll notation system, a subset of common music notation, and an Okinawan tablature notation system for voice and samisen.

While certainly not constituting "proof" of the dissertation's claims, the existence of three fully functional systems, each capable of displaying and playing substantially complex scores in real time, certainly supports their validity.

Chapter 6

Conclusion

6.1 Contributions

This dissertation has presented an original model of computerized notation programs for composition. This *three-dimensional, transparent glyph* (3TG) model represents musical scores as three-dimensional constructs made of piles of boxes of unit thickness called *Glyphs*. Equivalent to tree structures, these piles specify hierarchical relationships among their component glyphs. If these glyphs are implemented as objects, then the images and conventions of particular notational styles can take the form of data encapsulated as object specifications in an object-oriented, visual programming language.

An original data structure, called the *TTree*, was presented as a possible implementation structure for the model. A TTree is the binary correspondent of an ordered tree with time delays associated with its right-going arcs. In a TTree, time can be thought of as flowing "along" these arcs, rather than somehow "between" them, as in the corresponding ordered tree. When the glyphs of the 3TG model are implemented as nodes of a TTree, their hierarchical interrelationships not only specify structural connections, but define temporal ordering as well.

The model strongly suggests a unique, direct-manipulation-style user interface. Because the model is based on concrete, familiar objects of everyday experience, it makes an excellent metaphor for the design of such an interface. It was shown that

most of the elements of the interface could be based directly on such a metaphor, resulting in an interface with a good deal of "conceptual integrity".

A working computer prototype of the model, dubbed *Nutation*, was described. Although not constituting proof of any rigorous kind, the prototype does at least lend credence to many of the claims made about the model throughout the dissertation. One particular claim—that the model is capable of supporting a wide variety of music notation styles—was given especially strong support, for the implementation included a piano-roll notation system, a subset of common music notation, and an eighteenth-century tablature system for voice and samisen from the island of Okinawa.

6.2 Directions for Further Research

There are many aspects of the 3TG model which were addressed only partially or not at all in this dissertation. The algorithmic generation and manipulation of glyph piles, for example, warrants further exploration, particularly in light of the declarative nature of the model's data representation, for it is generally believed that declarative language descriptions are easier to manipulate than procedural ones.

The model does not address the issue of multiple simultaneous hierarchies. Simultaneous generation along multiple hierarchical dimensions was shown to be a powerful tool for analysis as early as Winograd's systemic grammar scheme for tonal music [Win68]. More recently, Dannenberg has proposed an extensible data structure for music representation which allows the encoding of multiple hierarchies [Dan86]. While Nutation and the 3TG model certainly don't preclude such structures, they don't provide any particular support for them either.

A number of pragmatic issues arose during the coding of the model's computer implementation. These issues primarily concerned problems of *scale*, for Nutation is currently not very successful at dealing with very large scores. How should something of the complexity of, say, a complete opera, be organized? As a single glyph pile? As a set of independent, sequentially-ordered piles? The graphics algorithms used by the implementation were neither particularly innovative or efficient, and though they served well for the short examples presented in the dissertation, they almost certainly

Bibliography

[Ado85] Adobe Systems Incorporated. *PostScript Language Reference Manual*, 1985.

[Ado89] Adobe Systems Incorportated. *PostScript Language Extensions for the Display PostScript System*, 1989.

[AK86] David P. Anderson and Ron Kuivila. Accurately timed generation of discrete musical events. *Computer Music Journal*, 10(6):48–56, 1986.

[App88] Apple Computer, Inc. *Human Interface Guidelines: The Apple Desktop Interface.* Addison-Wesley, Reading, Mass., 1988.

[BC89] Judith R. Brown and Steve Cunningham. *Programming the User Interface: Principles and Examples.* John Wiley & Sons, New York, 1989.

[BPRB81] William Buxton, S. Patel, William Reeves, and Ronald Baecker. Scope in interactive score editors. *Computer Music Journal*, 5(3):50–56, 1981.

[BRBM77] William Buxton, William Reeves, Ronald Baecker, and Leslie Mezei. The use of hierarchy and instance in a data structure for computer music. *Computer Music Journal*, 2(4):10–21, 1977.

[Bro75] Frederick P. Brooks, Jr. *The Mythical Man-Month.* Addison-Wesley, Reading, Mass., 1975.

[Byr84] Donald Alvin Byrd. *Music Notation by Computer.* PhD thesis, Indiana University, 1984.

would prove inadequate for large scores.

It should be emphasized that the model may very well provide solutions to many if not all of these problems. Music notation by computer is a very large and complex undertaking, and only a subset of the issues involved could be realistically addressed within the scope of this dissertation. Only further research will tell if even more powerful conceptual tools than the 3TG model are needed to solve them.

[Byr86] Donald Byrd. User interfaces in music-notation systems. In *Proceedings of the International Computer Music Conference 1986*, pages 145–151. Computer Music Association, 1986.

[Cho85] John M. Chowning. The synthesis of complex audio spectra by means of frequency modulation. In Curtis Roads and John Strawn, editors, *Foundations of Computer Music*, chapter 1, pages 6–29. MIT Press, Cambridge,MA, 1985.

[Col74] Hugo Cole. *Sounds and Signs.* Oxford University Press, London, 1974.

[Cox86] Brad J. Cox. *Object Oriented Programming:An Evolutionary Approach.* Addison-Wesley, Reading, Mass., 1986.

[Dan86] Roger B. Dannenberg. A structure for representing, displaying, and editing music. In *Proceedings of the 14th International Computer Music Conference*, pages 241–248. Computer Music Association, 1986.

[Dan89a] Roger Dannenberg. Real-time scheduling and computer accompaniment. In Max V. Mathews and John R. Pierce, editors, *Current Directions in Computer Music Research*, chapter 17, pages 225–261. MIT Press, Cambridge,MA, 1989.

[Dan89b] Roger B. Dannenberg. Music representation issues: A position paper. In *Proceedings of the 15th International Computer Music Conference*, pages 73–75. Computer Music Association, 1989.

[Day89] Mary Carlo Day. Designing the human interface: An overview. *AT&T Technical Journal*, 68(5):2–8, September/November 1989.

[Die85] Glendon R. Diener. Formal languages in music theory. Master's thesis, McGill University, Montreal, Quebec, 1985.

[Die88] Glendon R. Diener. TTrees: An active data structure for computer music. In *Proceedings of the 14th International Computer Music Conference*, pages 184–188. Computer Music Association, 1988.

[Die89a] Glendon R. Diener. Nutation: Structural organization versus graphical generality in a common music notation program. In *Proceedings of the 15th International Computer Music Conference*, pages 86–89. Computer Music Association, 1989.

[Die89b] Glendon R. Diener. TTrees: A tool for the compositional environment. *Computer Music Journal*, 13(2):77–85, 1989.

[Die90] Glendon R. Diener. Conceptual integrity in a music notation program. In *Proceedings of the 16th International Computer Music Conference*, pages 348–350. Computer Music Association, 1990.

[DJ85] Charles Dodge and Thomas A. Jerse. *Computer Music: Synthesis, Composition, and Performance.* Schirmer, New York, 1985.

[FG82] Allen Forte and Steven E. Gilbert. *Introduction to Schenkerian Analysis.* W.W.Norton and Company, New York, 1982.

[Fra] Franz Inc., Suite 275, 1995 University Ave, Berkeley, CA 94704 USA. *Allegro CL User Guide.*

[GL85] J. D. Gould and C. Lewis. Designing for usability: Key principles and what designers think. *Communications of the ACM*, 28(3):300–311, 1985.

[Gom77] David A. Gomberg. A computer-oriented system for music printing. *Computers and the Humanities*, 11(2):63–80, 1977.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[Gre90] Thomas R.G. Green. Limited theories as a framework for human-computer interaction. In D. Ackermann and M.J. Tauber, editors, *Mental Models and Human-Computer Interaction I*, pages 1–39. Elsevier Science Publishers B.V., North-Holland, 1990.

[Gru90] Jonathan Grudin. The computer reaches out: The historical continuity of interface design. In Jane Carrasco Chew and John Whiteside, editors, *Empowering People: CHI 90 Conference Proceedings*, pages 261–268. ACM Press, 1990.

[HSF89] Walter B. Hewlett and Eleanor Selfridge-Field. *Computing in Musicology: A Directory of Research.* Center for Computer Assisted Research in the Humanities, 525 Middlefield Road, Suite 120, Menlo Park, CA, 1989.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 1979.

[Jac89] R.J.K. Jacob. Direct manipulation in the intelligent interface. In P.A. Hancock and M.H. Chignell, editors, *Intelligent Interfaces: Theory, Research, and Design*, chapter 6, pages 165–212. Elsevier Science Publishers B.V., North-Holland, 1989.

[JB65] L.A. Hiller Jr. and R.A. Baker. Automated music printing. *Journal of Music Theory*, 9(1):129–152, 1965.

[JB89] David Jaffe and Lee Boynton. An overview of the sound and music kits for the NeXT computer. *Computer Music Journal*, 13(2):48–55, 1989.

[Kau67] Walter Kaufmann. *Musical Notations of the Orient: Notational Systems of Continental East, South, and Central Asia.* Indiana University Press, Bloomington, 1967.

[Kay84] Alan Kay. Computer software. *Scientific American*, 251(3):53–59, 1984.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, Mass., 1973.

[Loy89] Gareth Loy. Composing with computers—a survey of some compositional formalisms and music programming languages. In Max V. Mathews and John R. Pierce, editors, *Current Directions in Computer Music Research*, chapter 21, pages 291–396. MIT Press, Cambridge,MA, 1989.

[Mar86] Mark of the Unicorn, Inc. *Professional Composer User's Manual*, 1986.

[Max81] John Turner Maxwell III. Mockingbird: An interactive composer's aid. Master's thesis, Massachusetts Institute of Technology, 1981.

[Min86] Marvin Minsky. *The Society of Mind.* Simon and Schuster, New York, 1986.

[MO83] John Turner Maxwell III and Severo M. Ornstein. Mockingbird: A composer's amanuensis. Technical report, Xerox Corporation, 1983.

[NeX89a] NeXT Inc. *The NeXT System Reference Manual*, chapter 2: The NeXT User Interface. NeXT Inc., 1989.

[NeX89b] NeXT Inc. *The NeXT System Reference Manual*, chapter 16:The Mach Operating System. NeXT Inc., 1989.

[NeX89c] NeXT Inc. *The NeXT System Reference Manual*, chapter 6: Program Structure. NeXT Inc., 1989.

[NeX89d] NeXT Inc. *The NeXT System Reference Manual*, chapter 24: PostScript Operators. NeXT Inc., 1989.

[NS73] William M. Newman and Robert F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, New York, second edition, 1973.

[Pas88] Passport Designs Inc., Suite 103, 625 Miramontes St., Half Moon Bay, CA 94019 USA. *Score User's Guide*, 1988.

[Pol88] Peter G. Polson. The consequences of consistent and inconsistent user interfaces. In Raymonde Guindon, editor, *Cognitive Science and its Applications for Human-Computer Interaction*, chapter 2, pages 59–108. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1988.

[Ras82] Richard Rastall. *The Notation of Western Music: An Introduction.* Greenwood Press, New York, 1982.

[RC84] Xavier Rodet and Pierre Cointe. Formes: Composition and scheduling of processes. *Computer Music Journal*, 8(3):32–47, September/November 1984.

[Rea87a] Gardner Read. *Music Notation: A Manual of Modern Practice.* Taplinger Publishing Co.,Inc., New York, second edition, 1987.

[Rea87b] Gardner Read. *Source Book of Proposed Music Notation Reforms.* St. Martin's Press, New York, 1987.

[Row82] Lewis Rowell. *Thinking About Music.* The University of Massachusetts Press, New York, 1982.

[Rue51] Jan La Rue. The Okinawan notation system. *Journal of the American Musicological Society*, 4(1):27–35, 1951.

[Sch83] Bill Schottstaedt. Pla: A composer's idea of a language. *Computer Music Journal*, 7(1):11–20, 1983.

[Shn83] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer Magazine*, 16(8):57–69, 1983.

[Shn87] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, Reading, Mass., 1987.

[Shu88] Nan C. Shu. *Visual Programming.* Van Nostrand Reinhold Company, New York, 1988.

[SLU89] Lynn Andrea Stein, Henry Lieberman, and David Ungar. A shared view of sharing: The treaty of Orlando. In Won Kim and Frederick H. Lochovsky, editors, *Object-oriented concepts, databases and applications*, chapter 3. ACM Press frontier series, New York, N.Y, 1989.

[Smi73] Leland Smith. Editing and printing music by computer. *Journal of Music Theory*, 17(2):292–309, 1973.

[Smi75] David Canfield Smith. *Pygmalion: A Creative Programming Environment.* PhD thesis, Stanford University, 1975.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In *OOPSLA '87 Conference Proceedings*, pages 138–146. ACM Press, 1987.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Rockville, Maryland, 1988.

[Wae90] Yvonne Waern. On the dynamics of mental models. In D. Ackermann and M.J. Tauber, editors, *Mental Models and Human-Computer Interaction I*, pages 73–93. Elsevier Science Publishers B.V., North-Holland, 1990.

[Wan88] Paul S. Wang. *An Introduction to Berkeley UNIX.* Wadsworth Publishing company, Belmont, California, 1988.

[Weg87] Peter Wegner. Research directions in object-oriented programming. In Bruce Shriver and Peter Wegner, editors, *The Object-oriented classification Paradigm*, pages 479–560. MIT Press, Cambridge,MA, 1987.

[Win68] Terry Winograd. Linguistics and the computer analysis of tonal harmony. *Journal of Music Theory*, 12:2–49, 1968.

[Yav85] Christopher Yavelow. Music software for the Apple Macintosh. *Computer Music Journal*, 9(3):52–67, 1985.