ABSTRACT

Dozens of computer systems for music notation have been developed over the last 20 years, and I have been working on mine for more than 12. In this paper I attempt to give some insight into what are desirable features of such a system and also how these features relate to more general questions of "human engineering", that is, designing a computer system so that it can most efficiently interact with a (human) user. In my opinion, efficient interaction requires a system which not only has high- and low-level control features, but which "understand" its domain well enough to make switching from level to level easy. This is a state-of-the-art programming problem.

Computer music notation is an aspect of computer graphics, and I comment on levels of control in general graphics and on the problems of writing portable graphics programs.

BACKGROUND

Dozens of computer systems for music notation have been developed over the last 20 years, and I have been working on mine for more than 12. In this paper I will attempt to give some insight into what are desirable features of such a system and also how these features relate to more general questions of "human engineering", that is, designing a computer system so that it can interact with a (human) user as efficiently as possible.

The reader needs to know a few things about how I started working on this project to understand my approach. I began life as a composer. Like every composer, I found myself spending many hours on the essentially mechanical and unmusical task of neatly recopying - sometimes with transposition - scores and parts of my pieces from illegible pencil manuscripts. So it is not surprising that when I first learned something about computers in
1967 I thought, "this machine could be programmed to be my copystaf", and within a few months set out to do so. 

Incidentally, my naive first estimate of how long it would take to have a usable program was on the order of a few months - not 12 years.)

The computer situation at Indiana University was, and basically still is, unlimited access to a "main computer" (CDC 6600), very mediocre timesharing (500 baud), and almost no interactive graphics. This is not atypical of large universities, although IU's administrative policies on access to the system are extraordinarily liberal. In any case, the mediocrity of the interactive facilities forced me to take basically a "batch-processing", therefore automatic as much as possible, approach to everything.

One thing I quickly became aware of when I started programming was the enormous amount of time people at IU were wasting duplicating work already done. I saw two reasons for this: (1) programs written for one machine that could not easily be run on another; and (2) programs performing related functions that could usefully be run together except that they had - for no good reason - incompatible formats. Realizing that I would likely eventually be using different computers, I wanted to avoid both types of waste in my own work. This reasoning led me to (1) write my programs as portably as possible, and (2) work towards an integrated system of music programs - including analysis and sound synthesis components as well as notation - which could easily be used together.

I have now described my basic goals. The rest of this paper will discuss, first, how I attacked the technical problems of portability and making an "integrated system"; second, some of my ideas about human engineering in general and third, human engineering in my notation system. I'll conclude with some comments on the future of computer music notation and its relationship to general problems of computer graphics. A major thesis of this paper will be that a well-engineered system must be controllable on many levels and must allow changing levels easily. Examples will be cited to support this. 

PORTABILITY AND SYSTEM INTEGRATION

I'd like to emphasize that when I say a program is "portable", I do not mean simply that it is written in a programming language that is widely available. Extensions to most major languages are so common that this by itself guarantees nothing. I really mean

307
something very much like this definition of a 'portable program': a program that can be made to run on a new machine with less that two percent of the work it would take to write from scratch for the new machine. I also mean something this definition fails to specify: that the program can be made to run on most machines at all! The reason for this provision will become clear shortly.

Thus, a high degree of portability was one of my main goals, writing complex portable programs of any type is not easy; with graphics programs it is especially difficult to achieve. With interactive graphic programs in the strict sense of "interactive" - as of 1980 - it is impossible. Here's why.

First, based on what has already been said, for any program to be portable it must be written in a standard version, such as those approved by ANSI (the American National Standards Institute), of a very common language, or at least a subset of the language that is very widely supported. A few candidate languages are FORTRAN, PASCAL, LISP.

Secondly, if a program does graphics, it will be harder to make portable because the program must be independent of the graphics hardware as well as the CPU. Just as high-level programming languages can hide the differences between various CPUs, high-level graphics packages can hide the differences between various graphics devices; but such packages are much less standardized than programming languages - which are themselves not in such good shape.\footnote{2}

If real interactive graphics are wanted, the situation is much worse. I say "real" to eliminate devices like Tektronix' storage CRTs, which cannot selectively erase information. With such a display, the only way to, say, move a note from one staff line to the next is to erase the entire screen (with a flash of green light) and redraw it with the new note. Now, I doubt that "most computers" have interactive graphic hardware by any definition; eliminating storage CRTs from consideration settles the question. There are serious compatibility problems even with the hardware that remains: some devices are raster scan and some random scan (i.e., vector-drawing), and graphic input devices vary greatly in their characteristics.

So portable, interactive graphics is not completely. The best we can do is the non-interactive kind, also called "static" or "passive" graphics, namely what is done with hardcopy plotters. (Of course, this type of graphics can also be done with all types of CRTs, including storage.)
I will assert without argument that the programmer who wants to write a complex, highly portable graphics program has no choice of programming language: he/she must use FORTRAN, and the closer to ANSI X3.9 Standard FORTRAN (1966), the better. This is unfortunate, because ANSI FORTRAN is a weak and clumsy language. Even using ANSI FORTRAN is not enough, however: ANSI leaves undefined some important things, for example the maximum number of characters that can be stored in a variable, and of course ANSI says nothing about graphics. In fact, the only graphics primitives (i.e., built-in functions) that are universally available are primitive in another sense: "draw a straight line", "write a string of characters with specified height and location", and "write a number with specified height and location".

These three are all the primitives SMUF (my System for Music Transcription) uses, and it is written in 99 percent ANSI FORTRAN. As evidence of the success of these techniques, SMUF has driven, with only a few control card changes, mechanical and electrostatic plotters, dice wheel printers, and vector-drawing refresh raster scan refresh, and storage CRTs. With a high-resolution plotter, it can produce very high quality plots: see Fig. 1, drawn on a Calcomp 1537 and photoreduced. In addition, SMUF has recently been converted to run on a 16-bit virtual memory minicomputer from the original 62-bit microcomputer version. The conversion took about 80 hours, which is less than 1/3 percent of the 1000 or so hours involved in developing the original version.

Developing an integrated system was a relatively simple process. Having extremely limited programming time (averaging two quarter-to half-time people for the last 12 years), we decided early to use existing programs in our system such as possible, so the 24/7SM includes the well-known MUSIC V sound synthesis program, David Cernak's stochastic composing program, etc. These programs are linked with interface programs developed at Indiana University (see Fig. 2). Thus we have put together an integrated system with minimal effort, but at some cost in the system's rate of use and power. Most of the system is reasonably portable as well, although not to the same extent as SMUF.

LEVELS OF CONTROL AND HUMAN ENGINEERING

Let me begin this section by defining my terms. I use "level" as shorthand for "level of description". This is common usage in the phrase "high-level language", but is really a very familiar concept in general: the thing that is referred to on a high level
as "a computer" can be described at a slightly lower level as "a central processing unit, memory, and input and output equipment" and at a much lower level as "520 molecules." In music notation, a high-level description of a piece might sneak of "pages"; a lower-level description of "staves" or "systems"; a still lower-level description of "measures"; and an even lower-level description might mention "notes, rests, clefs, dynamic marks, slurs, barlines, and key signatures." Finally, when one talks about programming, the language is high-level, in the sense that the language allows describing operations on an abstract level, suppressing many details.16

The whole point of using any machine is to aid human beings; hence, "human engineering"—designing machines to be easily used by people—is vitally important. This is something that sounds obvious but which, I think, computer system designers very often fail to pay enough attention to.

To use a person's time most efficiently, a system should let him work most of the time at a very high level, leaving all details to the machine. This is fairly well understood: witness the ever-increasing dominance of higher-level programming languages over assembly language. What is far less well understood is the importance of letting the user change levels easily, going to lower levels when he/she needs to do something the system doesn't "know about." (An exception is, of course, when the system "knows about" everything in the problem domain. I'll come back to this.) For example, standard higher-level languages such as PASCAL, SNOBOL4, and FORTRAN do not let the programmer store into or read individual bits of variables.17 I say "standard languages" because the result of deficiencies like this is that many compiler writers add the capabilities themselves.

The advent of interactive computing is closely tied to questions of levels of control. Text manipulating or word processing systems are a good example, and one closely related to music notation. Automatic text formatting programs—e.g., DEC's RUNOFF—have been popular for years. These are programs which take some text with embedded commands for changing margins, starting paragraphs, etc., and simply produce formatted output without human intervention. Now, however, text editors with formatting capabilities are becoming increasingly popular. My point is just that in an automatic program such as a text formatter it is possible to provide high-level control and some kind of low-level control, but that really convenient low-level control is impossible. This is because the whole reason for going to a lower level is to do something the program doesn't understand, that therefore the user is really doing the work...
"manually", and that therefore he really needs to see the effect of each command immediately. In fact, one can think of making low-level changes after high-level processing as retouching. I'll use this terminology again. On the other hand, an interactive program can provide very convenient low-level control, and can also provide high-level features fairly easily. Designers of interactive programs still need to work hard on making level changing easy to do - and they rarely do.

These comments apply to all areas of computer music - Buxton et al. have emphasized multi-level control in a sound-synthesis system <8> - and to computer music notation in particular. Most music notation programs have been batch-oriented, i.e., automatic, and lack low-level control; some are interactive and lack high-level control; a few have both high- and low-level control, but even they do not necessarily allow changing levels easily. We will return to this question shortly.

I pointed out earlier the obvious fact that if a system understands its domain perfectly, low-level control, "retouching", is never necessary. It is less obvious than music notation is not just very difficult, but - until major advances in artificial intelligence have been made - impossible to program perfectly. I hope Fig. 3, from a Chopin Nocturne (opus 15, no. 2), will help convince the reader of this. What imaginable algorithm would be able to position the beams in the right hand part as shown - a notation which certainly is not technically "correct", with its beams intersecting unrelated stems, but which is probably the best that can be done with this music?

THE MUSTRAN/SNUT MUSIC NOTATION SYSTEM

As Fig. 2 shows, the SNUT notation program can be accessed through two interface programs, called JANUS and SMIRK. JANUS generates SNUT commands from MUSIC V-type note statements, SMIRK from a file built by another program called MUSTRAN. I'll concentrate here on the latter path.

MUSTRAN is a translator for an easy-to-learn alphanumerical representation of music."<9> It was developed by Jerome Venker of UNIVAC, originally for ethnomusicalogical purposes, but now includes a great deal of art music notation. A simple example is given in Fig. 4.

For the reasons discussed earlier, MUSTRAN/SNUT is not an interactive system. It provides both high- and low-level control features, but the lack of interaction adversely affects the low-

313
Figure 3

314
SLOTH CANON
FROM THE MUSICAL OFFERING, 1735
J.S. BACH
(1685-1750)

Figure 5
level features. I won't attempt to be comprehensive here; I'll just give a few examples in each category.

Several high-level features are illustrated in figures:

Fig. 5 was produced from the same data as Fig. 1 simply by asking for a set of parts instead of a score and specifying for voice 3 (the only one shown in the figure) a transposition of -6 semitones and tenor clef replacing the original bass clef. (Fig. 5 was also produced on a lower-resolution plotter than Fig. 1, the versated 1200 instead of the Calcomp 1037, taking the resolution into account; SMUT did not thicken the staves here.)

Fig. 6 demonstrates "voice grouping" and "conditional character-strings". The "MODERATO" is marked as conditional in the data for each voice. Then when SMUT was run to print the score, it was told to group the voices as 343, resulting in the interrupted barlines and the suppression of the "MODERATO" except above voices 1, 4, and 7 (the first in each group).

If a set of parts were requested, "MODERATO" would appear in each.

Figs. 7 and 8 show a very complex and high-level feature: "rhythmic simplification". Music theory and notation texts commonly make statements like this:

"There are two guiding principles for all rhythmic notation: 1. The notation should show as clearly as necessary the underlying metric organization. 2. The notation should be as concise as possible."

While perfectly correct, statements like this are very far from being explicit enough for implementation on a computer. Consider the patterns to the left of the arrows in Figs. 7a, 8a, and 9a; clearly these should be rewritten as shown on the right of the respective arrows. On the other hand, Fig. 9b is all right as it is.

I've attacked this problem by first assigning numeric strengths to every rhythmic point in the measure, as suggested for simple and compound meters in Figs. 7d and 8b. Then a few rules applied to each note (or group) tell whether it needs to be divided into tied components, and if so, where. Fig. 8 gives "before" and "after" this technique was applied to a variety of rhythm patterns from Gardner Read's book on rhythm notation. I think that most musicians would agree that the program behaved correctly everywhere except for measures 2 and 3 of Ex. 2-4a (Read's example number 2); there the program actually made things worse. These two rhythms probably cannot be handled
PARTITA SOPRA ALLELUJA OCTAVI MODI
THEME AND VARIATIONS FOR NINE HORNs
ARR. DAVE KRIEWALL
CAMIL VAN HULSE
12. (1687)

Figure 6
a. \[ \frac{3}{4} \cdot \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \Rightarrow \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \]

b. \[ \frac{2}{2} \cdot \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \Rightarrow \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \]

c. \[ \frac{2}{4} \cdot \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \text{ is ok.} \]

d. \[ \frac{4}{4} \cdot \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \Rightarrow \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \]

e. \[ \frac{6}{8} \cdot \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \Rightarrow \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \]

f. \[ \frac{7}{7} \cdot \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \Rightarrow \begin{array}{c}
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\text{\textbackslash \textbackslash} \\
\end{array} \]

Figure 7
correctly on a note-by-note basis; the pattern needs to be
looked at as a whole.¹³

Some of SMUT's low-level control features, suitable for
use in "retouching", are:
1. control of whether slurs go above or below notes, and how
   highly curved they are. Normally, SMUT is just told on
   which note the slur begins and on which it ends.
2. Control of how many measures should be put in each system.
   This is normally totally automatic.
3. An "edit" command, which allows modifying the calculated
   parameters of any symbol just before it is actually drawn.

These features make it possible to do just about anything, but
they don't necessarily make it practical. There is simply no
substitute for interaction for doing this kind of "retouching".
And so, I've tried to improve SMUT's high-level capabilities to
minimize the need for retouching. An indication of my success is
that none of the examples accompanying this paper required any
retouching. In any case, my programming time has not been
wasted, because the ultimate music notation program will need
extremely good high-level control and low-level control and
most difficult - ability to switch levels.

CONCLUSIONS

I'd like to conclude with a few remarks on the future of music
notation programs and on the relationship between music notation
problems and general graphics problems. There is already at
least one notation program that provides very good low-level
control, namely Leland Smith's MSZ, now in use at Stanford
University. (Not surprisingly, it relies heavily on interactive
graphics.) And high-level control facilities can be pushed well
beyond where they are now without any real conceptual problems.

The really hard problems, as I've suggested already, are related
to switching among levels and to what Ivan Sutherland called the
"Structure of Drawing Problems" many years ago.¹⁴ In a recent
paper¹⁵, Lakin proposes solving Sutherland's problem with eight
be called a true structure oriented graphic editor. See fig. 9a,
one of Sutherland's examples, about which I quote from Lakin:

"Sutherland's example is a graphic editing situation in
which there are five similar boxes, each with a dot in it,...some pointing device is positioned as shown and a
BEFORE command is given. "The correct computer response
depends upon the underlying structure of the drawing." For
instance, the user could mean delete the dot in that box, delete the whole box, or perhaps delete the group of two boxes of which that box is a member."

In this example, there is no visible indication whatever that the lefthand two boxes form a group (see Fig. 9b). Lakin's solution is to make the user tell the program what the underlying tree structure is, and provide an option for displaying back the structure (the thin lines in Fig. 9b). This approach is promising for a general solution to Sutherland's problem, but for music it will be far too clumsy: a system cannot support this "high-level retouching" without understanding what it's doing. For example, the user says "move this measure to the next staff" (after painting to a measure with, e.g., a light pen). The system should fix justification, break slurs that now cross staves, and unbreak slurs that no longer cross staves. Also, tree structure is probably too restrictive for music notation (or for any other representation of music, for that matter).

So far, this is difficult, but solvable. Now let's say that a user formats his/her music, then, in order to save vertical space between staves, makes a low-level change: "move the left end of this beam 5 units right". Music engravers actually do this sort
of thing.) Now the user discovers a whole measure missing and inserts it - a much higher level change which requires completely repositioning the beam. To do the right thing with it, the program will need to understand why the user moved the beam to begin with. By 1980 standards, this is another artificial intelligence problem. I've already used Fig. 5 to suggest that a complete and completely automatic music notation program is an AI problem. One can at least console him/herself with the thought that writing a program that can be told why you did something funny (such as having a beam cutting through a stem) should be easier than writing a program that can decide to do something funny on its own.

ACKNOWLEDGEMENTS

The work described here would have been impossible if not for the generosity of the Wubel Computing Center of Indiana University in providing computer time and logistic support of all kinds. I would also like to thank Jia Mattner, Scott Kle, Peter Montalvo, and Rosalee Merlina for many, many helpful discussions of various aspects of this work. Finally, I thank Rosalee for working on MUSTRAN and SMIRE.

NOTES


2. A great deal of work has been done on graphic software standards in recent years. See Status Report of the Graphics Standards Planning Committee, published as Computer Graphics 13,3 (1979); however, it will probably be another five years before standard-conforming packages are really widely available.

3. The raster scan/vector drawing dichotomy exists in hardcopy graphic devices also. The only reasonably stable way I know to run on both types is to use only vector drawing primitives (like these), which can easily be simulated with a raster device. This trick works less well with interactive devices for reasons of response time.

4. I am very happy to be able to say that there are definite signs of improvement in standardized programming languages as
well. ANSI's FORTRAN 77 is a big improvement over 1966 FORTRAN, and if we're lucky we may even be able to forget all FORTRANS in a few years.


7. True, PASCAL has RECORD and PACKED datatypes that support most of the operations one would want to do at the bit level—but not all. (I am told the language C really does let you do everything you might need to do.)


11. These rules will be given in a subsequent paper of mine.


13. Read also gives two versions of each example similar to my "before" and "after" versions which he calls "traditional" form and "modern" form. He implies that he prefers the "modern" versions at least for contemporary music. I, and most musicians, I have asked, disagree.


323