INTERACTIVE SYNTHESIS WITHOUT OBSCURE DIAGNOSTICS

W. Nelson J. Bridwell
Department of Electrical Engineering
and Computer Science
Northwestern University
Evanston, IL 60201

Abstract

Many digital music synthesis systems are virtually impossible to use because (1) sound playback delays take days, (2) the "system" is a vast conglomeration of incompatible software packages, and (3) short compositions require astronomical computing budgets. This paper describes MINI-SINGER, a system that will be powerful, easy-to-use, highly-interactive, and inexpensive. It will consist of separate edit, compile, sample generation, and sound playback overlays, all being written from scratch to run on a dedicated minicomputer. Particularly novel is the editor, which ensures that obscure diagnostic messages will not confront the user. The validity of source statements in the instrument definition program are checked as they are typed, rather than at compile time. Each keystroke is immediately rejected if it violates arithmetic expression syntax or implies an undefined variable/function reference. Type-ahead of implied characters within symbols is accomplished automatically.

Motivation

The current facilities for digital music synthesis at Northwestern University consist of MUSIC4BF software running on the central batch-timesharing system. Digital-to-analog conversion is performed off-line from magnetic tape on a time-available basis at the NU Psychoacoustics Research Laboratory. There are three compelling reasons to seek a more viable alternative.

The preliminary goal of any exploration into this realm is an understanding of the basic techniques (waveform generation, enveloping, modulation, filtering, reverberation, etc.). When these procedures are employed in combination the result can exhibit especially complex qualities. Interactive experimentation is vital in the development of an appreciation for these characteristics. Acceptable time delays between program modification and sound playback are on the order of minutes rather than days.

Secondly, because many systems are so haphazardly thrown
together, the beginner is usually sent off into a vast software labyrinth. Here the most elementary procedures involve traversing a maze of interconnecting languages, utilities, and command protocols. Even if the novice would flawlessly proceed according to directions with never a wrong turn, the pathway is through such a large quantity of code that undocumented trap doors (e.g. revisions, local modifications, program bugs, hardware errors, incorrect documentation) will likely be encountered. The beginner is certain to be confused when confronted with the obscure diagnostic messages that signify his deviation. Without assistance, the newcomer will likely become just one more lost soul that entered into the dungeon and has never again returned to see the light of day.

Finally, for the beginner and expert alike, music synthesis is expensive. A typical system will require 10 to 100 CPU seconds to produce one second of sound. Short compositions can consume hours of computer time. The computing budget for a synthesis class at a centralized facility can exceed the purchase cost of a dedicated minicomputer system.

Resources

The NU School of Music currently owns a DEC PDP 11/34 minicomputer with the following features:

1. integer multiply/divide,
2. 96K bytes of main memory under memory management,
3. dual floppy disk drives,
4. an alphanumeric CRT terminal (VT52),
5. a 12 bit precision digital-to-analog converter,

The overall system design is the work of the author and Professor Gary Kendall (School of Music, Northwestern University), both of whom are experienced assembly language programmers.

Objective

The composer will write a score and define instruments in a simple problem-oriented language. The instrument definition program will consist of sequences of FORTRAN-like arithmetic assignment statements. This syntax is more natural,
compact, and powerful than any of the assembler variants.

MINI-SINGER (< German minnesinger, singer of courtly love) will run on the 11/34. The central CDC 6600 may crunch floating point numbers faster, but the dedicated 11/34 configuration can provide shorter playback delays (minutes), will respond instantly and at the keystroke level when editing, and is infinitely less expensive. The system will run stand-alone and be written entirely from scratch. This is the only way that the user can be safely protected from the complex protocols that he would otherwise have to learn. Lacking any reasonable alternative, the system is being programmed entirely in MACHO-II. It will consist of four overlays (editor, compiler, sample generation, and sound playback) that are being developed in parallel. The composition will be compiled, rather than interpreted, because fast sample generation is essential for short playback delays.

The system that we are writing will use floating point arithmetic exclusively, even though the floating point unit has not been installed. It is possible to generate music with just integer arithmetic, but the underflow/overflow considerations excessively complicate the programming task for the beginner.

Implementation Schedule

The approach that we have taken is to first perfect a simple system and then to expand its capabilities. The stages of development should be as follows:

1) The composer will define a single instrument. Generation and playback will be restricted to only as many samples as can fit in main memory (perhaps 1 second of sound).

2) The composer will define multiple instruments and a symbolic score. Several minutes worth of samples will be generated to and played back from digital magnetic tape.

3) The composer will also be able to access libraries of scores, instruments, and sound files. Specialized hardware will be integrated into the system.

The editor has been designed, written, and tested by the author. It is now fully operational. The compiler is currently being written by Ken Frett (Department of Electrical Engineering and Computer Science, Northwestern University). Thus, this paper will focus primarily on the external and internal behavior of the editor.
Instrument Definition Language

An instrument program consists of three sections:

1. The note initialization assigns starting values to certain variables.

2. The frame update computes slowly-changing parameters (envelopes, modulation indices, filter center frequencies, etc.) at a fast enough rate (for instance 50 times per second) that there is no perceptible discontinuity. This feature can substantially reduce the playback delays.

3. The sample generation contains the final computations (modulation, waveform generation, filtering, etc.) that assign a value to the variable SAMPLE. These operations are performed at a sample rate high enough to avoid fold-over.

Each section contains a sequence of FORTRAN-like arithmetic assignment statements. There are no looping or conditional expressions. The differences from regular FORTRAN are:

1. Variable names are alphabetic and of any length.
2. All variables are floating point.
3. There are no arrays.
4. There is no exponentiation operator.
5. Function calls are delimited by square brackets [ ].
6. The assignment expression begins in the first column.

In future versions alphanumeric variable names, the exponentiation operator, and conditional assignment statements will be added.

The following example could be a legal instrument source statement:

\[ \text{ALPHA} = \text{G} \ast (\text{DC} \ast \text{BANDPASS} \ast \text{AMP} \ast \text{BUZZ} \ast \text{FREQ}, 6, \text{FMNI}, \text{FMNI} / 3, 0 ) \]

Consider the following simple instrument:
(frame rate)
FR=50
(sample rate)
SR=20000
(note initialization)
DURATION=1.0
LOWF=CYCLE[115,0]
HIGHF=CYCLE[130,0]
CARRIER=CYCLE[1000]
(frame update)
CONTROL=EXPONENTIAL[HIGHF,DURATION,LOWF]
ENVELOPE=LINEAR[0,0.2,1,DURATION=0.2,0]
INDEX=SPLINE[0,0.1,10,DURATION=0.2,4,0.1,0]
(sample generation)
SAMPLE=ENVELOPE*SINE[CARRIER+INDEX*SINE(CONTROL)]

This is an example of a frequency-modulated instrument where the control frequency CONTROL and modulation index INDEX change during the course of the note. The initialization defines the note duration (one second) and converts (CYCLE) several frequencies into waveform table increments. The frame update contains interpolation functions EXPONENTIAL, LINEAR, and SPLINE that have parameters of the form: (starting value, time interval, value, time interval, ... time interval, ending value). The sample generation computes the output value SAMPLE from a sequence of sine wave table lookups.

If this program is executed with a frame rate of 50, a sample rate of 20,000, and main memory can contain 20,000 samples, then the following sequence of events will occur. The initialization section will be executed first and only once, setting the starting values for DURATION, HIGHF, LOWF, and CARRIER. Then, for fifty times the frame update section will be executed once (updating the parameters ENVELOPE, CENTER, and INDEX), each time followed by four-hundred iterations of the sample generation code. The SAMPLE output from each iteration of the generation code is saved for later playback.
note initialization (only once)
frame update (1st time)
sample generation (1st time)
sample generation (2nd time)
...
sample generation (399th time)
sample generation (400th time)
frame update (2nd time)
sample generation (401st time)
sample generation (402nd time)
...
sample generation (19,200th time)
frame update (50th time)
sample generation (19,201st time)
sample generation (19,202nd time)
...
sample generation (20,000th time)

Editor Dialog
------- -------

Serving as the central user-system interface, the editor is utilized to enter/modify the instrument definition program. Except momentarily when a note is being generated, it is always in continuous operation. When the composer loads his floppy disk into the drive and bootstraps the system the editor automatically comes up, displaying the main index.

Type "1" to change the frame rate,
      FR=200
"2" to change the sample rate,
      SR=20000
"3" to edit the note initialization,
"4" to edit the frame update,
"5" to edit the sample generation, or
"6" to perform the note.

The editor is organized around the structure of the instrument program. At each level a section of the program is displayed and the composer is presented with a menu of characters that he can type to select various operations. In the above instance if the key "1" was struck the editor would respond with

Enter a new numerical value for the frame rate,
followed by the RETURN key:
      FR= _

Here the underline character represents the CRT terminal cursor. The user could then type the characters "50"

Enter a new numerical value for the frame rate,
followed by the RETURN key:
      FR=50_
and the return key, after which the index would be
re-displayed with the new frame rate.

Type "1" to change the frame rate,
FR=50
"2" to change the sample rate,
SR=20000
"3" to edit the note initialization,
"4" to edit the frame update,
"5" to edit the sample generation, or
"6" to perform the note.

Entry of the program is accomplished by first typing "3" to
insert the note initialization section. A heading and the
first line number will come into view.

Enter the note initialization section.
Terminate each source line with the RETURN key.
Input a blank line when you are done.

01_

It is at this juncture that the unique nature of the editor
becomes immediately apparent. As each character of a source
line is being input, it undergoes a syntax and reference
check. Each keystroke is instantly rejected (the terminal
beeps and pretends that the user never hit that key) if it
violates arithmetic expression syntax or implies an
undefined variable/function reference. For instance, if
while typing the first line ("DURATION=1.0") a plus sign is
entered instead of the equal sign, the computer will
complain. Likewise, it will indicate displeasure if you
mistype the decimal point as a comma. The complete scope of
the input verification logic is discussed later. For now
let us discuss why this is desirable.

We all remember how after working hard on our very first
computer program, we were rewarded with such clear and
friendly responses as:

**************************************************************************
FATAL ERROR - ILLEGAL SYNTAX IN LINE 6
FATAL ERROR - DANGLING OPERATOR IN LINE 10
FATAL ERROR - EXCESSIVE LEFT PARENTHESIS IN LINE 14
FATAL ERROR - STACK OVERFLOW IN LINE 17
**************************************************************************

While most compiler diagnostics are cryptic, others are
simply abysmal. The usual solution is to provide the
services of a seasoned programming aide. Without expert
advice the beginner may be forced into a trial-and-error
search because these messages do not sufficiently pin-point
the necessary corrective actions. In contrast, the
MINI-SINGER editor will not allow the entry of incorrect
statements in the first place.
The design goal of the editor is simplicity. Consequently it is line-oriented and employs no special characters (escape, control, etc.) other than the delete key. Once a line is entered, there is no way that it can be modified without entirely retyping it. The input checking logic minimizes the chances that a line will have to be reentered. Without it, the user might find himself introducing even more typographical errors when making corrections.

Perhaps you have now become convinced that the strongest point of a system can be what it will not permit you to do. It is interesting that the type-ahead feature is a spin-off of the variable/function reference checking logic. The editor must know all the function names and all the variable names that your have thus far defined. Whenever you start to type a name reference, if the characters that you have entered imply all or part of the rest of the name, these inferred characters will automatically appear. Thus, as the second line of the note initialization section is being entered, once the user has typed "Y"

Enter the note initialization section.
Terminate each source line with the RETURN key.
Input a blank line when you are done.

01 DURATION=1.0
02 LOWF=CY_

the system can uniquely identify this reference to the function CYCLE, and will instantly insert "CLE[".

Enter the note initialization section.
Terminate each source line with the RETURN key.
Input a blank line when you are done.

01 DURATION=1.0
02 LOWF=CYCLE[

The remainder of the line "115.0"] would then be typed in by the user, followed by the return key.

Enter the note initialization section.
Terminate each source line with the RETURN key.
Input a blank line when you are done.

01 DURATION=1.0
02 LOWF=CYCLE[115.0]
03 _

The delete key is used to erase mistakes in the current input line. Each delete keystroke will backspace (over implied characters) to before the last character that you typed in. While typing in the third line, if
Enter the note initialization section.
Terminate each source line with the RETURN key.
Input a blank line when you are done.

01 DURATION=1.0
02 LOWF=CYCLE[115.0]
03 HIGHF=_

the user should mistakenly hit a "D" instead of a "C", the editor will recognize this as a reference to the variable DURATION, and automatically fill in the "URATION".

Enter the note initialization section.
Terminate each source line with the RETURN key.
Input a blank line when you are done.

01 DURATION=1.0
02 LOWF=CYCLE[115.0]
03 HIGHF=DURATION_

To correct this error the user would type the delete key only once, erasing (the implied characters "URATION") to before the last character ("D") that he typed. Then he could enter

Enter the note initialization section.
Terminate each source line with the RETURN key.
Input a blank line when you are done.

01 DURATION=1.0
02 LOWF=CYCLE[115.0]
03 HIGHF=_

the characters "CY", (the editor filling in "CLE[",) the characters "130.01", and the return key.

Enter the note initialization section.
Terminate each source line with the RETURN key.
Input a blank line when you are done.

01 DURATION=1.0
02 LOWF=CYCLE[115.0]
03 HIGHF=CYCLE[130.0]
04 _

INPUT VERIFICATION
----- -----------

Internal to the editor is a subroutine that is called to accept one new line of the instrument program from the terminal. Upon exit, this routine returns in an input line buffer either nothing at all or an assignment statement. Each character of the line is checked immediately as it is typed in. Valid characters are placed into the next
position of the line buffer and displayed on the terminal. Invalid characters are not moved to the line buffer, are not echoed on the screen, and result only in a "beep" from the terminal. The delete key will cause removal of the last character from the line buffer and the display. If, upon encountering the return key, the line buffer is empty or contains a complete and legal arithmetic assignment statement then this routine will exit.

Of each line this verification procedure first requires a result assignment and then an arithmetic expression. The result assignment is any non-null alphabetic string that is terminated with an equal sign. Syntax verification is accomplished with simple precedence rules. The following precedence table indicates which characters can be the legal successor to the last input buffer character.

last line buffer character:       \( N \)

\( N \) - nothing, empty buffer

current input character:         \( x \)

alphabetic                      \( A \)

equal sign                     \( = \)

return key                     \( R \)

delete key                     \( D \)

This next precedence table is employed to verify arithmetic expression syntax. When dealing with numerical constants it is necessary to distinguish between integer digits (left of decimal point) and fraction digits (right of decimal point). The verification logic can then accept a decimal point after an integer digit ("A=2."), while still rejecting a decimal point after a fraction digit ("A=2.5."). Since the ASCII code defines only 128 characters, this distinction is accomplished with the high order bit of each line buffer byte. When a digit is stored in the line buffer the high order bit is set only if the preceding character is a decimal point or a fraction digit. This routine also contains special logic to track the parenthesis and function nesting levels. In future versions the editor will ensure that each function call contains the correct number of parameters.
last line buffer character: \( A, I, F, O, (, ) [ ] \)
\( I \) - integer digit
\( F \) - fraction digit

<table>
<thead>
<tr>
<th>current input character:</th>
<th>A</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabetic</td>
<td>D</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>digit</td>
<td>.</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>decimal point</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>* / operators</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>minus operator</td>
<td>(</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>left parenthesis</td>
<td>)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>right parenthesis</td>
<td>[</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>left bracket</td>
<td>]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>right bracket</td>
<td>,</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>comma</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>return</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

While the arithmetic expression is being typed this editor subroutine is also ensuring that it contains no undefined variable or function references. If we are beginning, continuing, or terminating the entry of a variable/function name (current input or last line buffer character is alphabetic) then a reference check must be performed. The alphabetic string that we have entered is the search key. Function name searches are conducted in a pre-defined internal lookup table. A scan of preceding program statements for result assignments is used to determine which variables are currently defined. If there is no defined name that begins with the search key then the input character is rejected. When the first match is found the rest of the variable or function name is copied to the line buffer. Subsequent matches delete any conflicting implied characters. When the search is complete the current input character and all remaining implied characters are displayed on the terminal.

Conclusion

This paper has described a novel approach to interactive digital music synthesis. This preliminary system is nearly operational. User experience and expanding requirements (scores, digital tape sample generation) will dictate the future evolution of MINI-SINGER. It also appears that the new editor input verification and type-ahead techniques could be applied in general-purpose program development systems.