COMPUTER GENERATED MUSIC AS A TEACHING AID FOR FIRST YEAR COMPUTING[1]

Judy A. Franklin
Computer Science Department
Smith College
Northampton, MA 01063
jfranklin@cs.smith.edu

ABSTRACT

Computer generated music can be useful in introductory programming courses as the theme for programming projects as well as a method for demonstrating algorithms. Two ideas for student programming projects that have actually been assigned are described. But the emphasis of the paper is on the use of a technique known as algorithmic granular synthesis. This is explained first as a computer music technique and then its use as a method for demonstrating algorithms such as sorts and searches is described. Following this is a discussion of ideas for using computer music in other computer science courses.

INTRODUCTION

The evolution of computer music has paralleled that of the evolution of the machines themselves, both in hardware and in software. Currently computer music is a large area of research, dominated mainly by music faculty. But it is also an interesting area of research for computer science faculty and many of the techniques used to generate music by computer are borrowed from computer science. The field is advanced enough to have textbooks available that can be used in teaching a course on computer music. For example this author has taught a seminar course called Algorithmic Music Composition that included the use of algorithms such as cellular automata, neural networks, Markov chains, and context-free and context-sensitive grammars to generate music, to name a few. Algorithms can choose notes/pitches to create a melody, can add variations to a running musical theme or motif, or can select phrases according to an overall structural plan for a musical piece, for example. An excellent tutorial source is [Roads 1996].

Even at the first year level computer music can be used for projects in an introductory programming course for computer science majors. Two such projects that involved music are described in the following section.

Computer music may also be used as a classroom aid. An easy way is to simply play short computer generated songs to break up a class (especially useful in classes that are more than 50 minutes in length). This should provide inspiration as well, knowing that the machines the

[1]To appear in the Journal of Computing in Small Colleges

students are learning to program can be made to produce something that sounds so interesting. There are many sources for CDs of computer generated music, e.g. [Computer Music Journal 1999, ICMC 1996, SEAMUS 1999]. Students come to look forward to these little breaks. As a side note, we have a brief break in theory class; in each class one or two palindromes are written on the board from [Agee 1994]. The students enjoy the reward of making it through a difficult mathematical proof or complex example.

Computer music can also be used more directly as a means to demonstrate algorithms. It is not a replacement for other teaching techniques, especially visual ones, but it can augment these and perhaps create a different kind of appreciation of the algorithms. In turn the algorithms can be used for composition of computer songs. The specific techniques that are emphasized in this paper fall in a class called algorithmic granular synthesis. This will be described in a later section.

There are many different possibilities for using computer music in other courses and this paper ends with a discussion of some possibilities.

TWO MUSICAL PROJECTS

This section describes two musical projects that have been assigned to students in introductory courses. The first is an ear trainer and the second is a song sorter and player. The ear trainer focuses on if-else statements, switch statements, parameter passing, loops and using user input to cause a program to vary in its execution. The song sorter is a more involved project. It involves looping through an array of numbers that represent pitches and durations of notes and counting the rhythmic changes in a song. Then a list of songs is sorted according to the number of rhythmic changes, and finally the user chooses the song to play. Songs are stored as arrays and the students must manipulate an array of pointers to these song arrays. Each of these projects is described in more detail next.

Ear Trainer

The program repeatedly gives two options to a user. The program must give these options to the user, obtain the user's choice, exit a while loop if the choice is not 1 or 2, and pass the choice to a function that calls either the option1() function or the option2() function, depending on the parameter value.

Option 1 is to learn about relative differences in two tones. The user chooses the number of examples s/he wants to hear. The program loops. Each time through the loop the program generates two tones randomly (out of 7 possible tones) and plays them. The user chooses whether the first tone is higher than, equal to, or lower than the second. The computer plays the tones again and tells the user the right answer. After all of the examples are generated, the percentage of correct responses is calculated and displayed.

Option 2 is to learn about absolute tones. The user chooses the number of examples and the program loops. It generates one tone randomly out of seven possible tones. The user chooses

a pitch (via characters ‘C’, ‘D’, ‘F’, ‘G’, ‘A’, ‘B’) for a tone that is played and the program replays the tone with the correct pitch. Once again the percentage of correct responses is calculated and displayed at the end of the loop. When these projects were assigned the environment used was an old version (4.0) of Microsoft Visual C++. A library function called beep(), found under the help menu, takes two parameters and issues a tone through the machine’s speaker. These two parameters are the tone frequency and the duration in milliseconds. Variables are initialized with the frequencies that beep() needs to generate the musical tones (i.e. C = 262, D = 296, E = 330, F = 349, G = 392, A = 440, B = 494).

The ear trainer was assigned as the first project out of three major projects the students were assigned. The students were given 10 days to 2 weeks to do these projects. In this project, they learned to use the random number generator as well as the seed function. They had to understand the use of loops within loops, and the use of a value returned by a function as a loop parameter. They were given two files. One contained the Beep() function as well as a function called Sleep() that takes a number of milliseconds as a parameter and causes the program to sleep that long. The silence that ensued separated the tones (beeps). The other contained a template program that had the main() function already defined and empty definitions for the other functions. They were not allowed to change main() and had to adhere to and learn about the modularity of the template program. They also had to use two source files and compile them together.

Song Sorter

The song sorter project is more complex. Its focus is on manipulating arrays using pointers, using a sort and a search, and making the program modular by defining and employing several functions. They were not given a program template for this one. The project entails sorting a list of songs by the number of rhythmic changes in them and then repeatedly searching the sorted list for a song with a particular number of changes. That number is to be chosen by the user and the user may continue to search for songs in this way until s/he wants to stop. For example, the following array is declared and initialized:

```
int song1[] =      /* Array initialized to notes of song       */
{
   C1, HALF, G0, HALF, A0, HALF, E0, HALF, F0, HALF, E0, QUARTER,
   D0, SIXTEENTH, C0, WHOLE, END
};
```

The elements are variables. A variable like C1 or E0 contains an integer number that is a tone or sound frequency (as in the ear trainer except there are two octaves of the C major scale available this time). A variable like HALF or QUARTER is an integer number that indicates how long that tone should be played. When the function Beep() that was used in the ear trainer is passed C1, HALF it plays the C above middle C as a half note (variable HALF is assigned a certain number of milliseconds).

A second file called thesongs.c contains 8 songs the students can use for this project. Students are also encouraged to make up their own songs and a few did. The file also contains the definitions of the variables C0, D0, etc. and SIXTEENTH, EIGHTH, QUARTER, HALF,

and WHOLE. It also declares END and assigns it the value 0. END is used to check to see if the song is over when the program plays the song.

What are these rhythmic changes? The song above, song1, has 3 rhythmic changes. The first 5 notes are HALF notes. The 6th one is a QUARTER, so there is a change from HALF to QUARTER. The 7th one is a SIXTEENTH, so there is another change from QUARTER to SIXTEENTH, and the last note is a WHOLE, so there is a third change from SIXTEENTH to WHOLE. A song that has all WHOLE notes (or all HALF, etc.), has 0 rhythmic changes.

In the program the student must first declare an array of pointers and store the addresses of the 8 songs in this array. The program next plays all of the songs for the user. This must be done by a function that takes the address of the array of song pointers as its first parameter and the number of songs there are as the second. Just before each song is played, the program prints the number of rhythm changes it contains onto the screen, as well as what each note duration is, using the name of the variable. A list such as the following should appear:

HALF
HALF
HALF
HALF
HALF
QUARTER
SIXTEENTH
WHOLE

Finally the songs are sorted using the selection sort (recall the selection sort repeatedly puts the smallest value at the top of the remaining list after comparing and finding it among all other elements in the list remainder and so sorts in increasing order). The songs are then replayed in sorted order, via the same function used initially, and then the program calls a function that prints a menu of choices, giving the range of song changes available to the user. The function returns the user's choice and this is used by a function that performs a binary search on the sorted song list for a song with the right number of rhythmic changes. If a song is found, it is played. Searches are repeated until the user enters -1. Recall that the binary search algorithm works in a similar manner to someone looking up a number in a telephone book. The midpoint is found and it is determined whether the value (name) being searched for is to the left or right of the midpoint. Then that half is searched by finding its midpoint and finding the half (by now really a quarter) of the list in which the value may lie. The portion of the list that is searched is narrowed down quickly in this way until either the value is found or the list runs out.

The students are given the code for the binary search and selection sort, but they must modify them so they can be passed and can manipulate an array of pointers to integers, rather than an array of integers.

Comments

The students generally responded well to programmatically manipulating audio output. There were some fun moments such as noticing immediately when someone was in an infinite loop that contained audio output. Sometimes the lab was pretty noisy and that is an important consideration, although headphones can alleviate problems should they occur. I mix this kind

of project with more administrative projects such as building a database to index inventory or library books and the students like the variety. These music projects succeeded in that nearly every student was able to create a working program, and many talked about showing it to their friends and family.

GRANULAR SYNTHESIS FOR ALGORITHM DEMONSTRATION

Algorithmic granular synthesis is a method in which computer algorithms manipulate small "grains" of sound. These grains can vary in duration from the shortest duration that can be distinguished by the human ear to 1 second. The techniques discussed here use grains that have a uniform duration of 0.1 seconds. Many, many grains combined make up a new sound wave or a song. Grains may be separated by silence or may overlap to a small or large degree. Granular synthesis is a broad term within its narrow field of computer music in that a variety of techniques are used to achieve it. Some techniques manipulate sound in the time domain and others manipulate it in the frequency domain. At present we use time domain techniques based largely on algorithms that are used in programming projects in our introductory course for computer science majors. We have used the bubble sort, the quick sort and the binary search.

Generating Grains using Sinusoids

In order to generate grains using a sorting algorithm, a list of numbers is generated randomly. The list is typically from 1000 to 10000 numbers in length. This list is sorted. Recall that the bubble sort works by comparing two consecutive numbers in a list and swapping them if the first is larger than the second (sorting in increasing order). This causes the largest (heaviest) value to sink to the bottom of the list and the other values to bubble up to the top. Once a value reaches the bottom, that part of the list is no longer sorted, but another pass is made to the remaining part of the list in the same manner. When we use the bubble sort, each time there is a swap, the index of the larger of the two numbers is recorded in a second array. So the actual values being sorted are irrelevant. It is only the index values that are important and that reflect the mechanics of the sort. After the sort, a 0.1 second long sinusoid with frequency scaled by one of these index values, is generated; one grain is generated for each index value that was stored in the second array. The grains are placed side by side to create one long sound wave that demonstrates the algorithm.

Figure 1 shows a sound clip with 7 grains, each with a different frequency. The dark areas show grains with a very high frequency. In four of the grain depictions the sinusoid that generates a pitch with low frequency is evident. These sounds can be heard on the web at [Franklin, 2000].

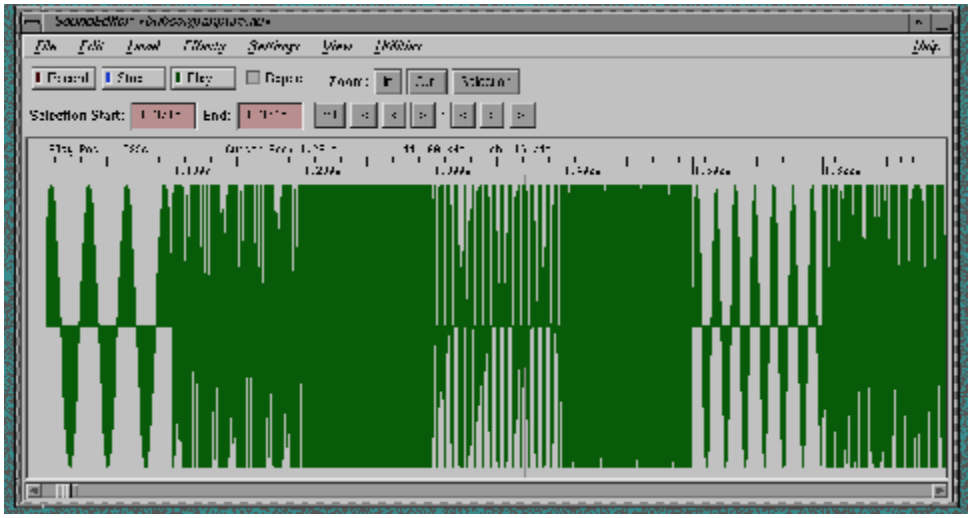

Figure 1 Clip of bubblesort indices choosing sinusoid frequency of each grain. No amplitude envelopes or harmonics are used for grain generation.

The quick sort is another algorithm that we used to generate grains of sound in order to form a composite sound wave. Recall that the quick sort is a recursive algorithm that first finds the midpoint value of the list. Then it starts at the endpoints and moves toward the center. Whenever an element on the left is greater than the center, it is swapped with an element on the right that is less than the center in value. The result is a list that is composed of two unsorted sub-lists, but each element in the first sub-list is less than each element in the second sub-list. Quick sort is then called recursively on each sub-list. As the quick sort is sorting our randomly generated list, whenever two values are swapped, the indices of both values are stored in a second array, in a manner similar to that described above for the bubble sort.

Once again, the actual values being sorted are irrelevant. It is only the index values that are important and that reflect the mechanics of the quick sort. As before, after the sort, a 0.1 second long sinusoid with frequency scaled by an index value, is generated; one grain is generated for each index value in the second array. The grains are again placed side by side to create one long sound wave that demonstrates the algorithm.

It is possible to manipulate the grains further, such as by adding harmonics to the sinusoid, or by passing the grain through an amplitude envelope (amplitude corresponds to volume). Figure 2 shows a short clip from the sound wave generated by the quick sort grains. Here two levels of harmonics have been added and the grain is passed through an envelope that simply ramps up linearly with a slope of 1 and then ramps down with the same slope.

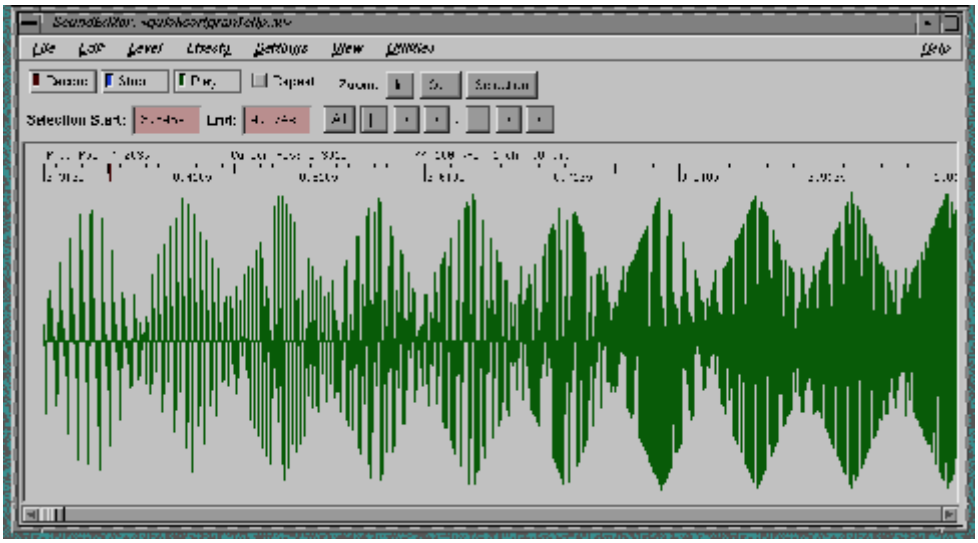

Figure 2 Index values involved in swaps in the quicksort algorithm are used to choose the frequency of sinusoids that generate the pitches of the grains. These sinusoids include 2 harmonics and each grain has an amplitude envelope.

Once a list of values is sorted, it can be searched by the binary search. We performed the binary search and when the midpoint of each sub-list being searched was calculated, we saved this midpoint index value in a second array. These index values were then used as frequencies for sine waves as before and grains generated. This time the grains are separated by space so they are heard as “blips” with silence in between. Figure 3 shows the grains generated by the binary search. The frequencies change quite a bit in the beginning, but at the end they are very similar, as the search narrows.

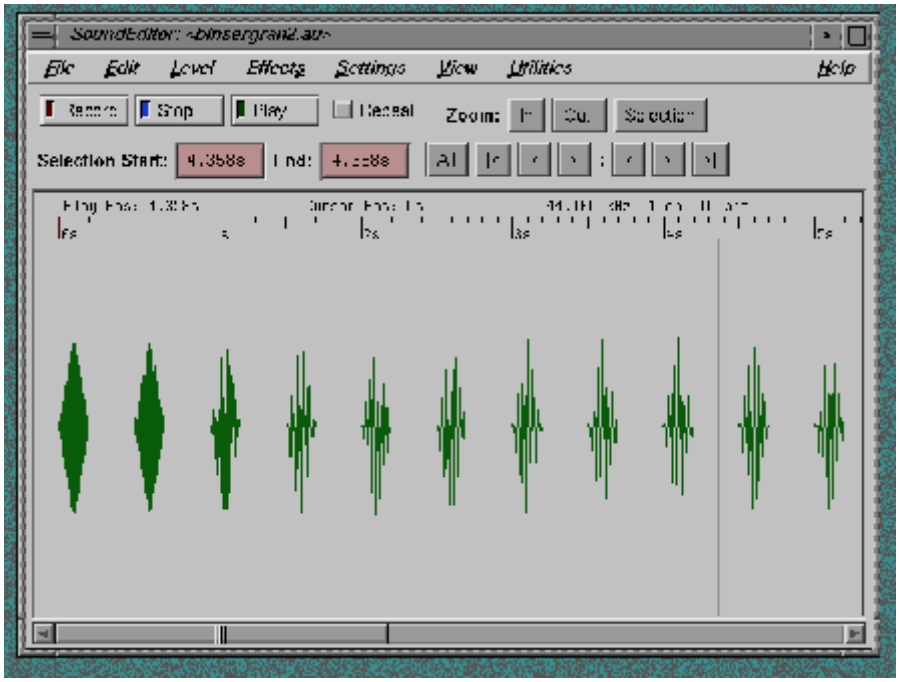

Figure 3. Binary Search. Indices that are successive midpoints in the binary search algorithm are scaled to choose the frequency of sinusoids that generate the pitches of the grains. These sinusoids include 2 harmonics and each grain has an amplitude envelope.

Generating Grains using Musical Recordings

We also used the three algorithms above to choose grains of sound from digitized recordings of people playing instruments. We were able to use the resulting sound waves in our own musical compositions. One of the people recorded was a senior who pays the guitar (Shana Negin); using a recording of another student should be inspiring to first year students. We made a recording of her improvising on guitar so that the composition would be original. We use the same array of index values obtained from the binary search. This time however, these indices are used as locations in the digital sound wave that represents the recording and a sample that is 0.1 seconds in duration is taken from that location. In other words a grain is generated. These grains are placed in sequence as before with space in between and Figure 4 shows the result. Notice that the grains are quite distinguishable in the beginning but are difficult if not impossible to distinguish by the end. The midpoints of the binary search are so close as the search is narrowed that they are used to choose samples that are highly overlapping and start at nearly the same location in the sound wave.

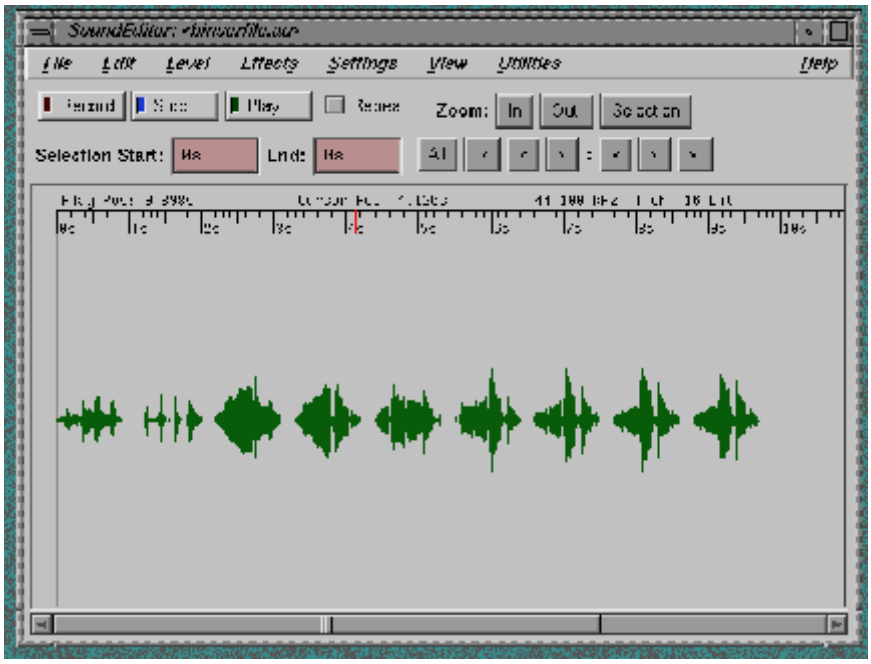

Figure 4. Binary search Using guitar improvisation as sound wave. Indices from the midpoints of the search algorithm are used to choose the location of grains of sound from the original sound file.

Finally, a recording was also made of a flute improvisation (by the author) and the list of indices resulting from a bubble sort were used in the same way to take out granular samples of the improvisation, using each index as a location in the original sound wave. While it is difficult to see the result in a visual representation of a clip of the composed sound wave, it can be seen in Figure 5 that there is a varied but repetitive structure that is a result of the bubble sort and that the new sound wave is complex enough to be used in a musical composition created as computer music. Indeed, it has been.

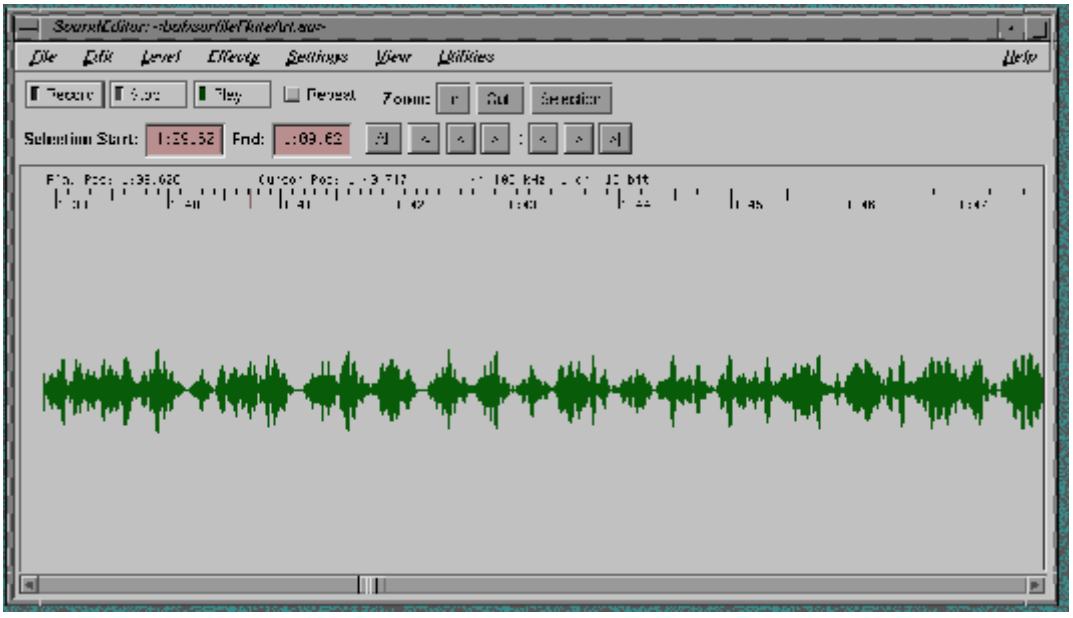

Figure 5 Bubble sort with flute improvisation as sound wave. Indices from the swaps made in the sort algorithm are used to choose the location of grains of sound from the original sound wave.

USING COMPUTER MUSIC IN OTHER CLASSES

Grammars have been used to represent music in various forms as early as the ninth century [Roads 1985]. There are many references to works using grammars in this way. [Roads 1985] has a nice overview in a chapter called "Grammars as Representations for Music." Another reference that has been used for a project in teaching theory of computation (at the undergraduate level) is [Johnson-Laird 1991] in which regular and context-free grammars are used to characterize different processes underlying musical improvisation. Students have presented this paper to the class and one student went on to present it in her music theory class. [Kohonen 1991] describes an algorithm that uses a context-sensitive grammar to devise rules that explain how a set of sample songs are generated. These grammatical rules are then used with a seed melody to generate a new melodic composition. If two rules have the same left hand side, one is chosen randomly. A student implemented this technique in an upper level seminar course on algorithmic music composition.

Many other courses could be augmented with music. Circuits and Systems and Computer Architecture are two candidates. For example, we can place a small amplifier and speaker on a circuit board with the microprocessor chip that is already used in this course and the project will be to build the circuit to do granular synthesis. In computer architecture our students read papers and make presentations on state-of-the-art of computer architecture design. There are ways to bring music even into this foray. Music workstations are available for serious computer music composers who develop electronic instruments via computer. These are high end processors that are integrated with musical keyboards and development software. It could be fun to look at the architectural design decisions that were made to tailor that machine for its musical use, for example.

DISCUSSION

Computer music techniques can be used to elucidate the algorithms and concepts taught in introductory programming courses. The use of music applications is also a reminder to computer science students that they can apply what they learn to artistic enterprises rather than thinking of what they learn as solely technological or business tools. The introductory course may be more accessible to non-majors by bringing in a liberal arts influence.

The granular synthesis methods briefly described above demonstrate the algorithms and can even simply be used as an audio aid to make the course more fun in order to foster learning. The next step in using algorithms for music in the introductory course is to develop specific teaching modules that use music to demonstrate concepts and as example projects and applications. In particular, some time must be devoted to making extended examples that show each stage of the algorithms shown above.

RTcmix (Real-Time cmix) is software written in C/C++ that is freely available on the web (http://www.music.columbia.edu/cmix/). The complete source code is available and one can programmatically implement instruments (as they are called) for granular synthesis or other kinds of synthesis. Furthermore, functions needed to create and read and write sound files are included. The figures in this paper are snapshots of windows of the Sound Editor from a Silicon Graphics Machine.

REFERENCES

Agee, Jon 1994. So Many Dynamos! HarperCollinsCanadaLtd. Canada.

Computer Music Journal. 1999. Sound Anthology. MIT Press. Cambridge MA.

Franklin, Judy 2000. Listen to Computer Science Algorithms. "http://www.cs.smith.edu/~jfrankli/music/algorithms.html".

ICMC 1996. International Computer Music Conference CD. International Computer Music Association (http://www.computermusic.org).

Johnson-Laird, P. N. 1991, Jazz Improvisation: A Theory at the Computational Level. in Representing Musical Stucture, Howell, West, and Cross eds., Academic Press, London.

Kohonen, T., Laine, P., Tiits, K. & Torkkola, K. (1991) A nonheuristic automatic composing method. In P.M. Todd & D.G. Loy (Eds.), Music and Connectionism (pp. 229-242). MIT Press. Cambridge, MA.

Roads, Curtis 1996. The Computer Music Tutorial. MIT Press. Cambridge MA.

SEAMUS 1999. Music from SEAMUS vol. 9. The Society for Electro-Acoustic Music in the United States. Los Angeles CA.