TJHSST Computer Systems Lab Senior Research Project
Designing a Music Scripting Language
2008-2009
Casey Mihaloew

Abstract:

The problem addressed in this project is finding a way to describe music and implement turning it into a sound file. The language will cover basic notes with rhythms and articulations. It will take into account different instruments as well as multiple of the same instrument. It will eventually allow continuous changes of certain factors like pitch, volume, and tempo, known as glissando, crescendo or diminuendo, and accelerando or ritardando. Currently, it can recognize notes of different pitches and modifiers on the pitches.

Background:

There has been a lot of work done in the area of computer music. A good survey of early work on computer music is Programming Languages for Computer Music Synthesis, Performance, and Composition, by Gareth Loy and Curtis Abbott. The focus of this paper is first to outline the various difficulties in designing a programming language for music, then discuss sound synthesis, and finally review many of the languages already created up to the time of publication (June 1985). There had already been a lot of advances in this area by 1985 as it was one of the earlier hard problems attempted by computer scientists.

For those readers not completely familiar with music, some basic terms will need explanation. The tempo is how fast the music is going and is typically measured in beats per minute (bpm). An accelerando is when the music gradually speeds up and a ritardando or rallentando is when the music slows down. The pitch is what frequency the note vibrates at, a higher frequency equating to a higher pitch. An interval is a ratio of frequencies, specifically an octave is 2:1. A glissando is when the pitch changes gradually. Volume is how loud the music is, with forte meaning loud and piano meaning soft. A crescendo is a gradual increase in volume while a decrescendo or diminuendo is a decrease in volume. There are also modifiers to notes that change one or more of the parameters, such as accents, accidentals, and other articualtions.

Sample Program

```
{import tempos}
{import volumes}
{def $i (tuba,trumpet,horn)}
{(initInstr)$i$i}
{$i(noteType basic)}
{$i(timeSignature 4:4)}
{$i(tempo $allegro)}
{$i(volume $forte)}
{$i(test)}
{(tuba,trumpet,horn)(
([C2:1/2],[D2:7/2]),
([C4:1],[B3:3]),
([G3:4] tenuto),

([C2:3],[D2:1]),
([E4:2],[G4:2]),
([F3:1],[E3:1],[D3:1],[C3:1])
)}
{$i(play)}
```

Procedures and Methods:

The fist problem that was dealt with was how to keep the language concise. With four parameters for each note in addition to modifiers, describing each note and all of its aspects could yield a long and unwieldy text file. The solution that I came up with is mapping. the easiest way to describe what mapping does is to give an example. The line "([C2:1],[C2:1],[C2:1],[C2:1])(staccato, tenuto)" under mapping will turn into "([C2:1] staccato,[C2:1] tenuto,[C2:1] staccato,[C2:1] tenuto)". This loops over all of the items in the first list and appends it to the corresponding item in the second list. If either list runs out of items, it starts over from the beginning of that list. It keeps going until the longer of the two lists is completed. This, once the program is completed, should drastically reduce the number of lines that one needs to type in order to describe the whole piece of music. In order to have two lists that don't map, currently I use curly brackets to encompass a specific need of mapping.

The design of the classes in this object-oriented project is one of the integral parts. The whole program is run by the driver, as is typical. The driver reads in a file and sends it to the parser which returns a list of commands. The driver then reads through the commands to see what it needs to do. As it goes through a proper file, it will create Instruments in a Hash Table. The instruments in turn keep track of an array list of Notes, as well as certain other parameters such as tempo, volume, time signature (class TimeSignature), and how to parse the notes, stored in the NoteInterpreter class. The note keeps track of what instrument it is played on, its volume, length, tempo, and pitch. The pitch is stored in the Pitch class, which keeps track of the name of the note and frequency. Various key parameters (both parts of the key signature, duration of notes) could be stored as a double, but more exactness can be had by storing them in a Rational class that was written for this project, involving adding and multiplying with stored numerators and denominators.

The current parsing does not work perfectly. It makes an array of commands to be parsed. For each command, it turns the nested parentheses into an arbitrary depth array. It then expands the mapping. The error in this way is that there is sometimes no difference between “)(“ and “,” and so it doesn’t map things that should be done. Fixing this is still an open problem for me.

The first section of output has the parsed program. The second section describes all of the notes recorded by each instrument as well as when the notes start and stop.

Results and Conclusions:

The result of this project will be an easy way of turning a text file into a sound file. It can be used by other researchers as a basis for any music generation or analysis project

The result of the test in the previous section is that all of the four parameters and modifiers are being stored successfully. I conclude that the next step will be to actually get sound to come out of the machine.

```
initInstr,tuba,tuba,
…
horn,play,
tuba ins. tempo: 120.0. volume: 0.6. time signature: 4 4
trumpet ins. tempo: 120.0. volume: 0.6. time signature: 4 4
…
trumpet ins. Start: 2.0 . Stop: 3.0 for note 2 counts of E4(329.6275569128699)
trumpet ins. Start: 3.0 . Stop: 4.0 for note 2 counts of G4(391.99543598174927)
horn ins. Start: 0.0 . Stop: 2.0 for note 4 counts of G3(195.99771799087463) tenuto
horn ins. Start: 2.0 . Stop: 2.5 for note 1 counts of F3(174.61411571650194)
horn ins. Start: 2.5 . Stop: 3.0 for note 1 counts of E3(164.81377845643496)
horn ins. Start: 3.0 . Stop: 3.5 for note 1 counts of D3(146.8323839587038)
horn ins. Start: 3.5 . Stop: 4.0 for note 1 counts of C3(130.8127826502993)
```