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

Casey Mihaloew

March 31, 2009

Abstract

The goal of this project is to create a scripting language that can describe music and be compiled into various forms of displaying music. There are four basic quantities that have to be dealt with in seperate ways: pitch, volume, note length, and tone. There is also the problem of defining these quantities for every note without having miles of repetative text. This problem will be addressed by a unique method that I call mapping that allows groups of commands to have common pieces. An additional goal will be to expand it to include transposing and other computational music theory functionality.

1 Introduction

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.

2 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.

3 Development

3.1 Formulating Syntax

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.
There are some limits to this syntax. It is highly specialized to the purposes stated, and if some new functionality were added, it would be hard to add it to the language.

3.2 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)}
```

3.3 Hierarchy

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). The note keeps track of what instrument it is played on, its volume, length, tempo, and pitch. It also stores this pitch symbolically and has some methods for computational music theory like transposition by an Interval. All of the volume, pitch, and tempi are stored in a Parameter class, which has a duration and a Function. The function can be of various types to take care of constant functions or glissandos/accelerandos/decrescendos. 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.

3.4 Parsing

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.

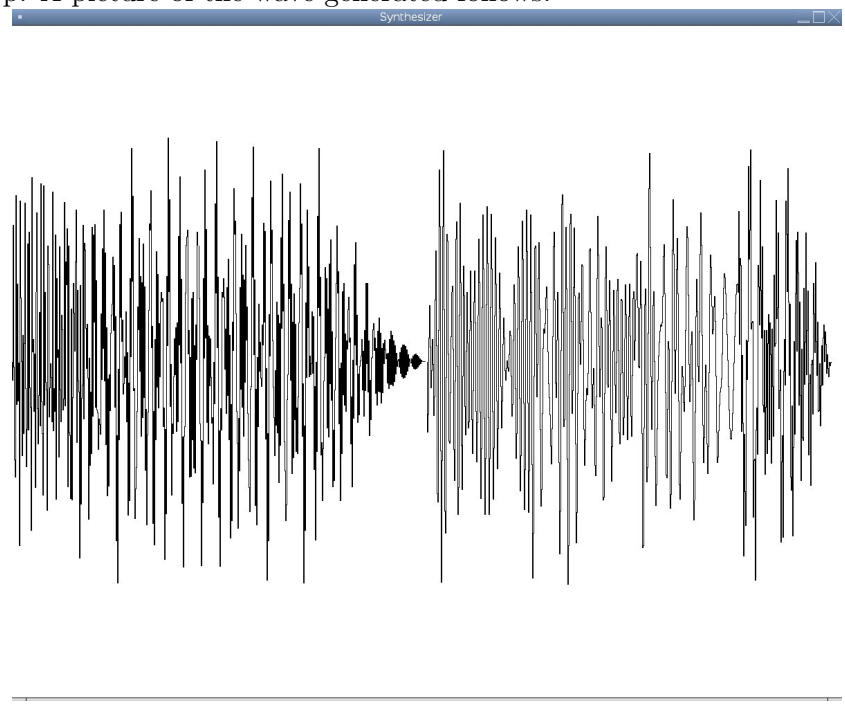
3.5 Testing

The output of my program is as follows.

```
initInstr,tuba,tuba,
initInstr,trumpet,trumpet,
initInstr,horn,horn,
tuba,noteType,basic,
trumpet,noteType,basic,
horn,noteType,basic,
tuba,timeSignature,4:4,
trumpet,timeSignature,4:4,
horn,timeSignature,4:4,
tuba,tempo,120,
trumpet,tempo,120,
horn,tempo,120,
tuba,volume,0.6,
trumpet,volume,0.6,
horn,volume,0.6,
tuba,test,
trumpet,test,
horn,test,
tuba,[C2:1/2],
tuba,[D2:7/2],
trumpet,[C4:1],
trumpet,[B3:3],
horn,[G3:4],tenuto,
tuba,[C2:3],
tuba,[D2:1],
trumpet,[E4:2],
trumpet,[G4:2],
horn,[F3:1],
horn,[E3:1],
horn,[D3:1],
horn,[C3:1],
tuba,play,
trumpet,play,
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
horn ins. tempo: 120.0. volume: 0.6. time signature: 4 4
tuba ins. Start: 0.0 . Stop: 0.25 for note 1/2 counts of C2(65.40639132514966)
tuba ins. Start: 0.25 . Stop: 2.0 for note 7/2 counts of D2(73.4161919793519)
tuba ins. Start: 2.0 . Stop: 3.5 for note 3 counts of C2(65.40639132514966)
```

```
tuba ins. Start: 3.5 . Stop: 4.0 for note 1 counts of D2(73.4161919793519)
trumpet ins. Start: 0.0 . Stop: 0.5 for note 1 counts of C4(261.6255653005986)
trumpet ins. Start: 0.5 . Stop: 2.0 for note 3 counts of B3(246.94165062806206)
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)
```

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. A picture of the wave generated follows.



3.6 Sound

I went about generating sound by a method known as additive synthesis. Using this, generating sound involves adding several different frequencies of sine waves to create a single instrument sound. All of these sounds are then added together to create the final piece of music. In a typical instrumental sound, like a wind or string instrument, there are a series of harmonics generated at frequencies twice, thrice, four times, and so on of the fundamental frequency. The various ratios of the magnitudes of these harmonics are what makes a clarinet sound different from

a trumpet or cello. Currently, I only use the fundamental frequency. The next step is to add tone to the notes.
Currently, the method that I use for the sound generation is to have a Sound class that stores an array of doubles that represent the amplitude of the note at that specific position in time. I instantiate a Sound for each Note and then add them together using a method in the Sound class. This works, but the overhead for creating all of those arrays causes the program to run slower than I would hope, taking around ten percent of the final length of the sound file.
The newer theoretical method is to have one sound for each instrument. I increment time by one over the sample rate, then get the tempo to find out how many beats this translates to. From that I can increment the functions attached to the pitch, volume, and tempo (often time doing nothing). The way to figure out what note can be done by keeping a precomputed array in the instrument class. This neglects to create an absurd number of arrays, but still has the relative slowness of additive synthesis. For the purposes of the project, it should be fine.
There are some nuances that I have implemented in the sound generation. I started with making the sound length different for different articulations. For tenuto, the note takes up one hundred percent of the time allotted for the note. For a normal note, it takes up ninety percent of the time allotted. For a staccatto note, the sound only takes up fifty percent of the time allotted. Even with this, there was a strange click as the note ended, so I tapered the end of the note off to zero. This involved linearly decaying the last quarter of the note from the full volume to zero. This eliminated the click and made it sound more human. This could still be improved by having a more continuous decay and using more different methods for different articulations.

4 End Matter

The goal of this project is to create a scripting language that can describe music and be compiled into various forms of displaying music. There are four basic quantities that have to be dealt with in seperate ways: pitch, volume, note length, and tone. There is also the problem of defining these quantities for every note without having miles of repetative text. This problem will be addressed by a unique method that I call mapping that allows groups of commands to have common pieces. I will also demonstrate the use of this language by generating sound. Additionally, expandable parts of the project will be able to do some music theoretical calculations.

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.

5 Literature Cited

Lloy, Gareth, and Curtis Abbott. Programming languages for computer music synthesis, performance, and composition . ACM , 1985. 31 Oct. 2008 ¡http://portal.acm.org/citation.cfm?id=4468.4485coll=Portaldl=ACMCFID=8612290CFTOKEN=77807504¿.