Scorewriter application with features aimed at Byzantine music processing

Petru Dimitriu
Faculty of Automatic Control and Computer Engineering
Technical University of Iași
Iași, Romania
dimitriupetru@gmail.com

Vasile-Ion Manta
Faculty of Automatic Control and Computer Engineering
Technical University of Iași
Iași, Romania
vmanta@cs.tuiasi.ro

***Abstract*—Byzantine music represents a vast and complex musical tradition which precedes contemporary Western music. Nowadays, it is used mostly within Eastern Christian churches. Few computer software applications deal with Byzantine music and the imperfect command of Byzantine music theory among singers often requires parallel classical notation staves. A software application written in C++ has been written, which enables the user to write simple musical scores using standard notation, as well as Byzantine music compositions using a subset of the specific Byzantine notation. The application can be used to produce transcriptions of Byzantine-notation compositions into their classical notation counterparts. Extensive data structures and algorithms for containing and manipulating the data, as well as interacting with the computer user, have been developed. Additionally, the application offers some serialization capabilities, as well as preliminary support for pitch tracking. By offering the functionality described above, the application is an example of modern technology being used to preserve cultural heritage.**

***Keywords*—software applications, educational applications, music typesetting**

I. Introduction

In the geographical areas where the former Byzantine empire has historically exerted extensive cultural influence, the prevalence of Byzantine music as church music has been one of the defining traits of the later resulting nations. Featuring a set of four octaves and eight modes, Byzantine music draws its roots to the mathematical discipline of music theory as it had been formulated during the Hellenic period and subsequently reformed by the Syrian polymath John Damascene and others [1]. Notably, Byzantine music also features a distinctive musical notation, which, unlike the classical Western notation with a staff, clefs and notes, has a more relative approach to expressing the melodic motion and employs a set of signs called *neumes*, which succeed and combine with one another to express the melody's tones, nuances and temporal features. Three main sets of signs are the *vocalic*, *temporal, consonant* signs. Other sets are used to establish the scale or the base tone, tempo etc. Byzantine music is employed as one of the main musical traditions in the Eastern Orthodox church.

Though originally developed exclusively in an ecclesiastical context, Byzantine notation in its contemporary form has proven its versatility in also being used to write secular music, with the Romanian composer Anton Pann's collections of Romanian-language Balkan folk songs of the 18th and 19th century such as *Spitalul amorului sau Cântecul dorului* serve as examples in this sense [2]. Even so, Byzantine notation is known by few people aside from Orthodox church chanters and many contemporary church hymn books present the hymns in both Byzantine and Western notations, in parallel.

As [2] notes, transcribing between Byzantine and Western notation is often problematic as there is no exact standard notation that can convey the musical embellishments found in Byzantine music. Nevertheless, an approximate transcription is necessary in many cases, as singers often cannot rely on their knowledge of the Byzantine notation, if at all.

Few computer software applications tackle Byzantine music, one of the most popular of those being the Greek-language *Chrisos Melodos* [3]. To the best of our knowledge, there is no public software application that can perform Byzantine-to-Western score transcription. We proceeded to designing such an application, by writing a computer program for the Linux operating system which supports writing simple standard and Byzantine notation compositions and partial transcription of Byzantine compositions to their approximate classical counterparts.

The remaining of this paper is organised as follows: In Section II the objectives that had been taken into consideration in the development of the application are established and the choice of resources is presented and explained. Section III is a description of the data structures used to encapsulate the data and information pertaining to the musical scores that the application manipulates. Section IV discusses the implementation of the user interface components which handle the drawing of the musical compositions and interaction with the user, providing algorithms in the form of pseudocode. Section V expands on the process of conversion between Byzantine and classical notation, explaining the algorithm that has been devised for this purpose. In Section VI the file format used by the application to store classical compositions is detailed. Section VII concentrates on the pitch-tracking component of the application and explains the mathematical formulae that is used to determine the musical note that corresponds to a certain sound frequency. Finally, Section VIII concludes the paper with considerations on the resulting software application and its usability.

II. Application Overview and Objectives

A. Resources

The application was written in the C++ programming language under the Ubuntu Linux operating system, using

CLion as an Integrated Development Environment and the GNU C Compiler (GCC).

The following third-party libraries have been used:

- *wxWidgets 3.0.4,* a framework for designing the user interface (UI);
- *Fluidsynth 1.1.9*, a real-time software synthesizer based on the SoundFont 2 specifications;
- *PortAudio 19*, an audio input/output library used for live acquisition of audio from the microphone.

All the third-party libraries feature free software licences and are cross-platform, which means that compilation on a different platform is possible with little or no modifications in code, resulting in a software application with the same functionality and behaviour across different operating systems.

For the purpose of pitch-tracking, the code at [4] has been used and modified to fit the needs of the application.

Fluidsynth requires a *soundfont* file in order to synthesize sounds. The free *GeneralUser GS* soundfont, available at [5], was used.

Additionally, for the display of Byzantine signs, the *EZ Fonts* True Type font package edited by the Saint Anthony's Greek Orthodox Monastery in Florence, Arizona, United States, has been used [6]. It has the advantage of providing a simple method for displaying Byzantine signs, by simply drawing plain text. Moreover, since the fonts are TrueType fonts, they are naturally scalable.

B. Capabilities

As of the date of the writing of this paper, the software application supports:

- composing and displaying standard musical scores featuring simple notes or chords, which can also have lyrics;
- composing and displaying Byzantine musical scores featuring a subset of vocalic, temporal and ornamental signs but no other signs;
- storing and retrieving composed standard-staff songs in/from files;
- transcribing Byzantine compositions into their approximate Western notation counterparts, highlighting the correspondence between groups of vocalic characters and their Western staff notes;
- playing the compositions through the third-party synthesizer library;
- live detection of sung notes using the third-party audio I/O library.

As far as Byzantine music support is concerned, we currently assume that the compositions are written in the eighth church tone (instead of supporting the full tone palette), which has its base tone on C and its musical scale (octave) corresponding to the Western major scales [1].

C. General design approach

The design of the application followed a mostly bottom-up approach, beginning with the design of the data structures that hold the information and programmatic methods related to musical elements and continuing with the user interface.

Generally, the application source code defines:

- data structures for the standard and Byzantine scores, respectively;
- custom widgets (UI controls) for displaying musical scores and interacting with the user;
- UI windows employing the widgets above;
- live audio data processing;
- interacting with the synthesizer library.

The data structures and the custom widgets that display and interact with them can be viewed as the Model and View parts, respectively, of a Model-View-Controller (MVC) design pattern, where the wxWidgets framework, through its event handling infrastructure, acts as the Controller.

III. Data Structures

The data structures that implement the internal programmatic representation of the two types of scores is designed as a group of C++ classes with specific purpose.

A. Standard score

The standard (Western) score data structures assume a representation of the score as a sequence of *chords*, each of which are comprised of an optional lyric fragment along with zero, one or more individual *notes*.

The *Note* C++ class represents a single standard musical note and is comprised of the real double-precision number of tones away from the C_0 musical note, the length of the note expressed as a real double-precision number expressed in beats (quarters) and a set of flags which can determine, for instance, whether the note features musical embellishments. It offers various methods in order to construct notes and alter notes, for instance, to create a note by specifying the name of the note (via a #define C++ preprocessor directive) and the octave, and to change the pitch by a certain number of tones.

The *Chord* class generally represents a chord, which is a group of notes that are sung/played together. It contains a *vector* from the Standard Template Library, (STL) of *Note* instances, an STL wide-character string (*wstring*) containing the corresponding lyric fragment, if any, and a set of flags which can be used, for instance, to mark that the *Chord* instance defines a rest.

The *Score* class represents a standard (Western) musical score which contains a list of chords (*Chord* instances) which comprise it, along with other settings such as the title of the composition and the tempo expressed in beats per minute. The *GroupedScore* class is an extension of the *Score* class which features a list of pairs of indices which divide the score into groups of chords. This is useful in order to easily realize and

display the correspondence between a classical score fragment and its Byzantine counterpart.

The *ScoreIO* namespace contains methods which perform the serialization and de-serialization of the contents of the *Score* class instances to/from external files.

B. Byzantine score

In the design for the data structures for Byzantine scores it is assumed for now, for simplicity, that the song is always composed in the eighth tone, starting on the initial note C_4.

The *ByzantineSign* class represents a simple or composite vocalic character, along with any auxiliary sign drawn above or below it. These fields are represented as wide characters (*wchar_t*) which correspond to the Unicode characters assigned to the respective signs in the *EZ Psaltica* font.

The *ByzantineScore* class represents a Byzantine musical score, containing an STL *vector* of signs (instances of *ByzantineSign*). It also implements a method to convert a Byzantine score into a Western score (an instance of *GroupedScore*).

IV. Custom UI Controls

In order to enable the user to interact with the data structures presented above and to properly display them on the screen, a set of special custom user interface controls had to be designed. These controls act as the View component of the MVC pattern discussed above.

Following the guidelines of the wxWidgets UI framework, they are implemented as classes derived from the framework-provided wxPanel and wxFrame classes. The operation of the custom controls is detailed in this section.

A. Standard score

The class that handles the displaying of Western scores has been called *wxScoreControl,* as seen in Figure 1. It contains a pointer to the *Score* instance that it manipulates and displays, and another pointer to a hidden *Score* instance that acts as a clipboard. The custom control is capable of laying out the song it represents either on a single row or on multiple staff rows.

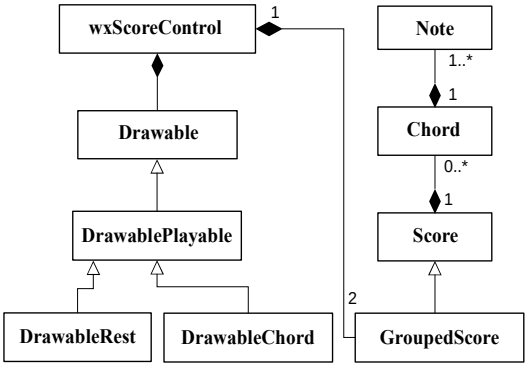

Figure 1: UML diagram of the classes related to Western score data processing

It implements basic event handlers in order to react when the user interacts with the control such as by hovering the mouse pointer above the control or clicking on it. Since the full contents will often be impossible to display, the control offers scrollbars that the user can use to adjust the viewing window; this requires that all coordinates of the objects within the control are always translated according to the scrolling coordinates.

1) The *build* phase

The painting process involves two main subprocesses. The first one, the *build* phase, involves linearly iterating through the list of *Chord* instances in the *Score* class and constructing an auxiliary data structure for each, which will contain specific information on how to draw the respective element. The general approach used in the *build* procedure is displayed in the form of pseudocode in Figure 2. It is only employed whenever a change that affects the layout of chords in the score occurs. This specific information is contained in instances of the *Drawable* class or one of its specialized derived classes, as it can be seen in Figure 1.

The common fields of the *Drawable* instances are the absolute coordinates of the rectangle that contains the drawable elements, which are highlighted whenever the user hovers the mouse pointer over and the user can click in order to select the respective elements. These coordinates are set during the *build* phase according to the type of the *Drawable* instance and according to whether it has extra signs, such as accidentals or embellishments, in the case of chords.

Each *DrawableChord* instance additionally contains a pointer to the corresponding *Chord* instance, the two integer extremities of the line of the drawn chord and a *vector* of the integer vertical positions of the dots belonging to the notes that comprise the chord. A number of programmatic flags determine whether the chord should have its stem drawn upwards or downwards.

```
procedure build():
    clear drawablesList;
    rowIndex ← 0;
    xRelativeInsideStaff ← xInitialOffset [in pixels];
    if scoreControl is not wrapped then:
        maxControlWidth ← ∞
    else:
        maxControlWidth ← scoreControl.width
    create Drawable instances for first two G- and F-clefs;
    append them to drawableList;
    for each Chord c in score:
        if c is a rest then:
            drawable ← new DrawableRest;
            set drawable.innerRectangle.x according to
                xRelativeInsideStaff;
            set drawable.innerRectangle.y according to rowIndex
        else:
            drawable ← new DrawableChord
            for each Note n in c:
                determine whether n has accidental etc;
                if n is between F3 and A3 or above B4 then:
                    stem should be pointing up for chord c;
                determine vertical position of notehead within staff
                and store it in drawable;
                determine and load number of flags;
                if both current and previous chords have flags and it is
                    feasible to join them then:
                        set a flag for drawable of current and previous
                        chords to signify that their flags should be drawn as
                        beams instead;
                <other operations>
            calculate and store position and dimensions for
                drawable.innerRectangle in order to contain all notes in c
            append drawable to drawableList
            xRelativeInsideStaff ← drawable.innerRectangle.width +
                                   spaceBetweenChords + xRelativeInsideStaff

            if xRelativeInsideStaff > maxWidth then:
                xRelativeInsideStaff ← xInitialOffset
                rowIndex ← rowIndex + 1
```

Figure 2: Pseudocode of the general approach of the *build* procedure

The *build* phase begins by creating an empty list of *Drawable* instances and adding two *Drawable* instances for the first two G- and F-clefs, respectively. The computation of the positions of the chords on this first row follows. The drawable elements for the G-clef and F-clef are the first two elements that are inserted into the *vector* of *Drawable* instances.

As the iteration through all the *Chord* instances contained in the *Score* instance progresses and positions for the chords and notes are successively generated, the horizontal positions of the newly-generated drawable elements are tracked in an internal variable *xRelativeInsideStaff*. If the elements no longer fit horizontally in the current viewport, then a new staff row is created (an internal variable is incremented and its corresponding clefs are added to the vector of *Drawable* instances) and *xRelativeInsideStaff* is reset.

In order to achieve a correct display of musical notes, the durations and pitches of the chords are also kept track of. If any note in a chord is higher or equal to B_4 or between F_3 and A_3, then the stem of the chord will be drawn downwards. If two chords on the same staff whose duration is lesser than a quarter have the same duration and their stems are pointing in the same direction, the stems will be united with a beam instead of having flag symbols drawn.

Finally, the *build* method computes the difference between the maximum extent of the drawn elements and the actual dimensions of the viewport and displays, hides or updates the attached scrollbars, if any, accordingly.

The *build* method is only called when a chord is added, removed or modified or the dimensions of the viewport are changed and the a new layout must be generated.

2) The *paint* phase

The *paint* phase, briefly explained in Figure 4, follows the *build* phase but is called whenever the user interacts with the control in any way. It involves iterating through the list of *Drawable* instances pre-generated during a previous *build* phase and applying a drawing approach specific to each *Drawable* type. The information in the *Drawable* instances allows this subprocess to also enjoy linear complexity. In order to draw shapes such as dots, flags and accidentals, a set of pre-loaded raster bitmap images are used.

Apart from the actual drawing of the musical elements, the *paint* phase also handles the filling of rectangles that contain the staff elements with specific colours for the situation when the chords are hovered, selected or highlighted, as in Figure 3.

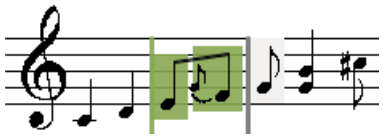

Figure 3. Screenshot of the standard score control. Two selected chords (with green background), one hovered chord (with gray background) and a chord containing two notes can be observed.

B. *Byzantine score*

In the case of displaying the Byzantine score, the *wxByzantineScoreControl* class (seen in Figure 5) has been written for the purpose. Its functioning is similar to that of

```
procedure paint():
    initialize drawing canvas dc;
    set background color of dc to white; clear background;
    for each i in [0,numberOfRows]:
        draw 5 hotizontal lines, representing the staff
    for each Drawable d in drawableList:
        if d.innerRectangle is outside of control viewport then
            skip iteration
        if d is hovered then:
            fill within the bounds of d.innerRectangle with the hover
                specific colour
        if d is selected then:
            fill within the bounds of d.innerRectangle with the selection
                specific colour
        if d is DrawableChord then:
            for each note in the Chord referred by d:
                draw its notehead with associated accidental, if any
            draw stem using data in d
            if chord referred has flags then:
                if flags should be drawn then:
                    draw flags
                else:
                    if previous chord can join flags with current one then:
                        draw beams between current and previous;
                    else:
                        store index of current chord until its pair is found
        <other operations>
```

Figure 4: Pseudocode of the general approach of the *paint* procedure

Figure 5: Screenshot of the Byzantine score control, with three groups of signs selected. The red signs, from left to right are a temporal and a consonant sign, respectively.

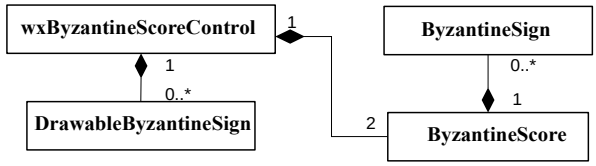

Figure 6: UML diagram of classes pertaining to Byzantine score data processing

wxScoreControl and it makes use of a class named *DrawableByzantineSign* to contain the necessary information generated during the homologue *build* subprocess and used for display in the homologue *paint* subprocess. The classes related to the Byzantine score data processing are explained diagrammatically in Figure 6.

Using the *EZ* font package eases the process of representing neumes internally, as for each sign there is a one-to-one correspondence to a Unicode character in the font. The *paint* subprocess simply draws the text contained in the specific data structures. As such, in the case, *DrawableByzantineSign* instances only contain the rectangle coordinates and a pointer to the *ByzantineSign* instance that they represent. Moreover, the characters in the *EZ* package are designed in such a way that modifier signs will be correctly placed relative to their base signs, in a similar manner to the correct placement of the isolated letter accents above (and not besides) the corresponding preceding letters in regular fonts.

V. Byzantine to Classical Notation Conversion

A method to convert scores contained in *ByzantineScore* instance to Western *Score* instances has been implemented by the authors in the *ByzantineScore* class. It produces an instance of the *GroupedScore* class.

The conversion procedure, explained briefly in Figure 7 involves two steps. The first step is a linear parsing of the

vocalic characters in the *ByzantineScore* instance. For each vocalic character, the corresponding note, or notes, are immediately added to the *GroupedScore* instance, along with a pair of indexes defining the group containing it. The effect of some temporal signs is introduced now.

The effect of each group of neumes builds up on the musical tone generated by the preceding neumes. The effect of some of the neumes, as implemented in the application, follows the guidelines traced by [1] and [2], and is exemplified in Table I.

TABLE I. EXAMPLES OF BYZANTINE NOTATION CHARACTERS

Shape of Byzantine character	Meaning
ˉ	maintains current step, unaccented
ˊ	maintains current step, accented
ˋ	2 steps up, accented
ˎ	1 step up, then two other successive steps up
ˏ	1 step down, then another step down

```
function ConvertToClassical(ByzantineScore byzScore):
    resultScore ← new empty GroupedScore;
    currentNote ← Note(C4);
    for each ByzantineSign sign in byzScore:
        append one or more notes to resultScore, according to kind
            of sign using currentNote as reference;
        currentNote ← last appended note;
        set the limits of a new group in resultScore, which will
            contain the notes added in this iteration;

    for each ByzantineSign sign in byzScore:
        i ← index of sign (and of corresponding group in resultScore)
        for each temporal or consonant sign attached to sign:
            perform required changes (alter duration, add accent etc)
            if new notes have been inserted in the process then:
                alter indexes of the group #i accordingly
                for each remaining group in resultScore:
                    update indexes of group accordingly

    return resultScore;
```

Figure 7: Pseudocode of the Byzantine-to-classical conversion function

Subsequently, the list of Byzantine signs is parsed again from first to last, in order to apply the effect of the temporal or consonant signs which affect more than one vocalic character, such as *digorgon,* whose usual action spans three vocalic characters, generating a triolet [3]. If a group of notes in the resulting *GroupedScore* instance needs to be adjusted in terms of the number of notes it contains, then all the groups (pairs of indexes) following the adjusted group need to be adjusted as well, rendering this process with quadratic time complexity.

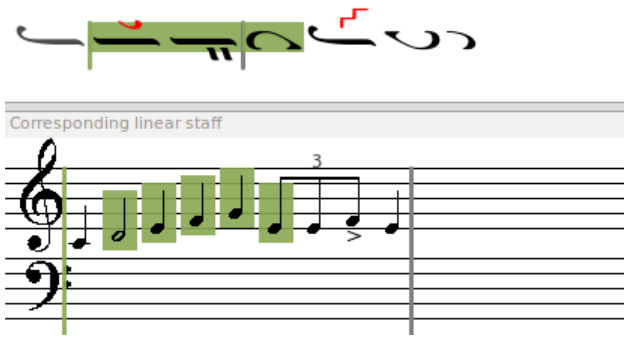

Figure 8: Screenshot of a Byzantine score and the classic score generated from it. Highlighting a group of neumes on the Byzantine staff causes the corresponding notes on the classic staff to be highlighted.

The correspondence established through the groups in the *GroupedScore* instance enables the user to immediately visualize the correspondence between the two notational systems, as seen in Figure 8.

VI. CLASSICAL NOTATION SCORE FILE I/O

The *ScoreIO* namespace mentioned above implements linear methods to write and read classical-notation score data to/from external files. The format used is both human and machine readable. It consists of a UTF-8 encoded text file with a number of single-line statements, as explained in Table II.

TABLE II. SCORE FILE FORMAT

Line contents	Example	Effect
i <integer>	i 0	Sets a musical instrument for the score; the musical instrument will be used to play the song using a synthesizer
t <integer>	t 80	Sets the tempo for the composition, expressed in beats per minute
n <string>	n My song	Sets a title for the composition
C <real> <real> [<string>]	C 29.5 0.25 Hello	Defines a chord, with the pitch defined by the first real number as the amount of tones away from C_0 and the duration defined by the second real number as a fraction of a whole note, with an optional lyric fragment at the end

VII. LIVE PITCH TRACKING

The application also features pitch tracking capabilities, by employing an implementation of the Fast Fourier Transform (FFT) algorithm applied to live data streamed using the PortAudio library, as explained at [4], with a sample rate of 8192 samples per second and a FFT window of 4096 samples. After the greatest amplitude is determined linearly, the musical note corresponding most closely to the determined value is determined.

In music, an *interval* represents the ratio between two sonic frequencies. When the frequency of a note is equal to the double of the frequency of another note, the musical interval between the two notes is called an *octave*.

Musical scales divide octaves into a number of intervals, whose number is usually eight, hence the name. The division points are the notes that are usually used in composition. In the English notation, the names of the notes ordered by their pitch in ascending order are: C, D, E, F, G, A, B.

The common musical interval called *tone* represents one sixth of an octave. A *semitone* is equal to one twelfth of an octave. In *equally-tempered* scales, the interval between each consecutive pair of steps is equal to $2^{1/12} \approx 1.05946$.

The application stores in memory the frequencies in Hz for all notes from C_{-1} to C_8 with a step of one semitone in a table for fast lookup.

Given that A_4 has its frequency conventionally set at exactly 440Hz by the ISO 16 standard, we can further devise a mathematical formula that can be used to compute the frequency of any musical note in an equally-tempered scale.

Let $f_{A4} = 440\,Hz$ be the frequency of A_4.

We can then compute the frequency of C_4 by taking into account that the interval between C_4 and A_4 is 9 descending semitones. Thus, the frequency of C_4 is:

$$f_{C4} = f_{A4} \cdot 2^{-9/12}.$$

In order to find the frequency of C_{-1}, which is 5 octaves lower than C_4, we divide the frequency of C_4 by 2^5:

$$f_{C-1} = f_{A4} \cdot 2^{-9/12} \cdot 2^{-5}.$$

We can now compute the frequency of any musical note n by employing the following formula:

$$f_n = f_{C-1} \cdot 2^s \approx 8.17579 \cdot 2^s,$$

where s is the number of semitones between C_{-1} and the note whose frequency we seek to compute.

VIII. Results and Conclusion

The resulting application successfully performs the operations described in Section II, featuring a window for working with Western scores, one for working with Byzantine scores and one for the live detection of pitch. The first two windows allow playing the score being worked on, while highlighting notes as they are being played.

Within the window used for working with Byzantine compositions, the capabilities of the Byzantine-to-Western conversion methods are leveraged, by providing a live transcription of the Byzantine composition currently being worked on. Playing the Byzantine composition will highlight both the neumes on the Byzantine staff and the notes on the corresponding classic staff as they are played.

Live pitch-tracking is used in a module of the application which requires the user to sing a number of notes displayed on a staff. As the user successfully reaches the notes, the next note to be sung is highlighted on the score.

These applications of the modules described above prove the practical and educational potential of the application. The application could be used as a self-teaching tool for acquiring theoretical and practical knowledge of both classic and Byzantine music. By making Byzantine music more accessible and easier to employ, the application could, thus, contribute to the better conservation of cultural heritage by the use of modern technology.

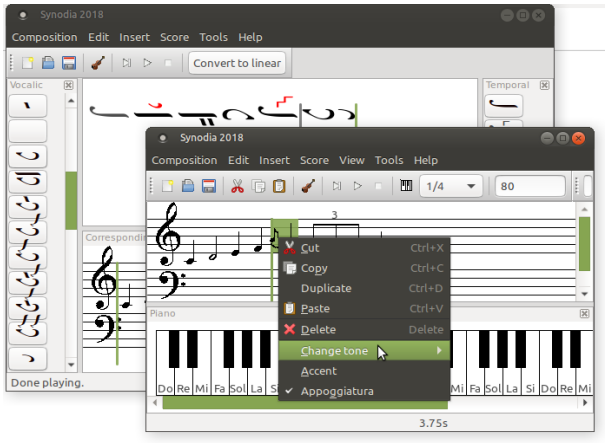

Figure 9: Screenshot of the built application

References

[1] Pantiru, G. "Notația și ehurile muzicii bizantine". Bucharest, Editura Muzicală a Uniunii Compozitorilor, 1971, pp.68–73.

[2] Pann, A., Ciobanu G. (coord.) "Cântece de lume", Bucharest, Romania, Editura de Stat pentru Literatură și Artă, 1955, pp. 26–77

[3] "ΒΥΖΑΝΤΙΝΗ ΜΟΥΣΙΚΗ, ΙΣΟΚΡΑΤΗΣ, OCR ΜΟΥΣΙΚΟ, OCR ΠΟΛΥΤΟΝΙΚΟ, ΠΑΡΑΔΟΣΙΑΚΗ ΜΟΥΣΙΚΗ, BYZANTINA ΜΟΥΣΙΚΑ ΒΙΒΛΙΑ, MUSIC COMPOSER, MELODOS, θέσεις εργασίας Πτολεμαιδα, OCR polytonic, music OCR, Tradition music" [Online], http://www.melodos.com/. Accessed 16th May 2018.

[4] Roche, B., "Frequency detection using the FFT (aka pitch tracking) With Source Code" [Online], http://blog.bjornroche.com/2012/07/frequency-detection-using-fft-aka-pitch.html, Accessed 16th May 2018

[5] "SountFonts and SFZ files" [Online] https://musescore.org/en/handbook/soundfonts-and-sfz-files#gm_soundfonts Accessed 16th May 2018

[6] "Byzantine Music Notation - Βυζαντινή Μουσική" [Online], http://www.stanthonysmonastery.org/music/ByzMusicFonts.html, Accessed 16th May 2018.

[7] Urmă, D., "Acustică și muzică". Bucharest, Editura Științifică și Enciclopedică, 1982, pp. 12-52