Sun Yat-sen University

Bachelor Thesis

Guitarist: a sheet music viewing and playing application for classical guitar learners

Author:
Zhanrui Liang

Supervisor:
A.Prof. Chenying Gao

A thesis submitted in fulfilment of the requirements
for the degree of Bachelor of Engineering

in

School of Software

April 2014

Declaration of Authorship

学术诚信声明

本人所呈交的毕业论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料均真实可靠。除文中已经注明引用的内容外，本论文不包含任何其他人或集体已经发表或撰写过的作品或成果。对本论文的研究作出重要贡献的个人和集体，均已在文中以明确的方式标明。本毕业论文的知识产权归属于培养单位。本人完全意识到本声明的法律结果由本人承担。

Signed: _____

Date: _____

SUN YAT-SEN UNIVERSITY

Abstract

School of Software

Bachelor of Engineering

Guitarist: a sheet music viewing and playing application for classical guitar learners

by Zhanrui Liang

Guitar is a musical instrument. Given a guitar score, there exist different fingering alternatives for playing it. It is difficult for novice learners to choose a proper fingering. This thesis describes an application that provides functionalities of sheet music rendering, playing and tablature generating. For sheet music rendering, this thesis describes the procedure from loading musical data from MusicXML file to rendering it to screen. For tablature generation, this thesis proposed a fingering arrangement algorithm using dynamic programming. For each piece, this fingering arrangement algorithm finds a globally optimal solution in the proposed model. In the proposed model, finger movement including pressing, releasing, moving are considered and their effect are reflected by the cost function. Cost function in this algorithm is rule based and parameters are assigned by experiment. The application has been tested on about 30 guitar pieces and four of them are listed in this thesis.

Keywords
guitar, sheet music, fingering arrangement, dynamic programming, rendering

Acknowledgements

First, to my advisor, A.Prof. Chenying Gao, for the continuous support of the work in this thesis, for her patience, motivation, enthusiasm, and immense knowledge.

Last but not least, I would give my thanks to my family for their permanent understanding and support.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

Guitar is a popular musical instrument that can be used to play a variety of musical genres. Typically, a guitar has 6 strings and 17 or more frets, dividing about 100 positions on its fretboard. In this these, we focus on the discussion on classical guitar. To produce sound with the guitar, one use the fingers on left hand to press a string to the bar on fretboard and pluck this string by right hand. Without special techniques, each position played by this manner will produce a sound with specific pitch. Different from the piano, which has a one-to-one correspondence between each pitch and each key on the keyboard, a sound with specific pitch can be produced on different positions on the guitar fretboard. Therefore, we need to choose some proper positions for the notes that we want to play.

Typical musical scores is written on the five-line staff, from which we can know each note's pitch easily. The five-line staff is not designed for string instruments so labeling fingering instructions on it can be cumbersome. For example, once the finger number is labeled around a note, there's no space left for labeling the fretboard position, and vice versa. Scores on many published books has finger names(p, i, m, a, representing four fingers of the left hand) labeled only on the first several measures. In many cases, knowing only the finger names is not enough to finish the fingering arrangement. To describe a complete fingering arrangement, we should assign a fretboard grid and a finger for each note. These make it not easy for beginners to figure out the proper fingering.

Tablature is a form of musical score that solve this problem to some extent. It's widely used for string instruments. It adopt a different format to denote music, by telling which position(both string and fret) to use for each note, instead of telling the note's pitch.

However, tablature is not always available. Many guitar pieces have only the score version, especially pieces for classical guitar. As [1] mentioned, one reason for this is that once a guitar player become experienced enough, he/she don't need it that much.

1.2 Objectives

The goal of this work is to present an environment for reading, listening and playing the guitar sheet music. Another goal is to provide a library that helps software developers process sheet music. All the source code in this work are open-sourced.

1.3 Previous Works

Sayegh [2] proposed the "optimum path paradigm" and use dynamic programming on Viterbi network. Radisavljevic et al.[3] proposed a "path learning" method to learn the cost function for dynamic programming, from marked data.

D.Radicioni et al.[4] proposed a segment based method, which extended Sayegh's method by adding the segmentation feature.

George et al.[5] proposed a more complex model that takes the physical constraints of human hand and guitar fretboard into account. The model is then solved using a greedy algorithm.

[1] aims to provide a system of fingering arrangement and tablature generation for melodies. This system was designed for melodies, which don't support polyphony music very well.

1.4 Contribution

In this work, we present a sheet music parsing and rendering library and a global optimization fingering arrangement method that supports polyphony music. The final application can read data from MusicXML [6] and then perform the following tasks:

- Score rendering.
- Tablature generation and rendering.
- Sound playing.

The novelty of the present work, compared to the previous ones, is that it takes finger pressing, releasing and timing into consideration. The actual time needed for finger movement is considered in the cost functions. The previous works don't take the time between two successive notes into consideration. However, sometimes the timing is important. For example, when the time between two note is long, the player don't have to move the fingers too fast, so we can translate the burdens here in the final arrangement. Since our method is a global optimization method, it can take advantage from the consideration of timing.

Unlike [1], the model we proposed for fingering arrangement is designed for polyphony music so than it can be used to nearly all classical pieces.

1.5 Structure of this Thesis

This thesis is divided into three main chapters.

Chapter 2 describes different steps of the sheet processing procedure, from data parsing to the rendering. Finally, some rendering result are put at the end of this chapter.

Chapter 3 is about sound playing. It also help the reader to understand the concept of note sequence, which is also used in Chapter 4.

Chapter 4 first introduces the physical guitar model, and then propose an abstract model for fingering arrangement. It then solves the fingering arrangement problem by the dynamic programming technique. Optimizations are then proposed to improve the time performance of the main algorithm. At the end of this chapter, there are four arrangement results attached and analyzed.

Appendix A lists the instructions for installing the application.

Chapter 2

Processing Sheet Music

2.1 Introduction

MusicXML[6] is a music notation interchange format. Nowadays, most commercial music software support this format so we can use it to interchange sheet music between different software. Before MusicXML, the universal music notation interchange format is MIDI. However, a midi file don't contain enough information for rendering sheet music. For example, the direction of note stem is not stored in midi file . Another example is that a note with specific pitch can be represented differently, with the alternating symbols such as sharp, flat and natural, etc.

As the website of MusicXML states, there are more than 170 applications include MusicXML support. Also, more and more scores in MusicXML format are available in digital music databases. All input data comes from the MuseScore [1] website.

2.2 Parsing MusicXML

MusicXML, as its name implies, is an XML based format. We have a wide range of XML libraries available in different programming languages. Though we can use XML libraries to load the XML file, there are still some work to put on the semantic analyzing of the musical content.

[1] musescore.com

2.2.1 Semantic Parsing

Semantic parsing is a process that converts data from the storage format to internal format. By internal format, we mean a format that is easy to use in the application. Since this kind of internal format varies from different application, there are few libraries available for this job. Having such a general library for semantic parsing is actually a reinvention of a general format. However, MusicXML is already a general format and is intuitive enough so having a new one is not necessary. We explored some open source computer music projects which have MusicXML support and found such internal format. The Music21 library [7] can convert between MusicXML and its own format, but it does not expose its MusicXML class as public API. The Lilypond [8] project has a script, namely "musicxml2ly.py", for converting from MusicXML to Lilypond's ".ly" format. Relevant classes used by this script are also not part of the public API and nearly undocumented, so it's not easy to reuse in our application. Combining these facts, we decide to write our own semantic parser for converting data from MusicXML to a format that can be used other parts of our application, such as rendering, sound playing and fingering arrangement.

MusicXML is strictly defined by an XML schema, which is also well documented. We can study the file format directly from the schema. In MusicXML, a musical score is stored as a tree structure[2]. It has the following main hierarchy, listed from the root:

- score
- part
- measure
- note/rest/direction

Before we introduce these element types, one thing worth to mention is that MusicXML use tenths as unit. A tenth is 1 / 10 of the distance between two staff lines.

In the following, we will introduce these element types briefly. More details about when an element will be used are in the following sections in this chapter.

2.2.2 Score

The score element is the root element. It provides the information about page layout such as page size and page margins. The scaling between tenths and real world length unit is also specified here.

[2]MusicXML has two types: time-wise or part-wise. We discuss only part-wise here

2.2.3 Part

A part element consists of measure elements. There may be several parts in a score, however, our implementation only supports one part. This is because most guitar pieces can be expressed using one part and they are actually expressed using only one part. Using one part does not mean that we can't have polyphony music since MusicXML can put chords in a single part. This will be explained in 2.2.4.

2.2.4 Measure

The measure element is the most complex part of MusicXML. The most essential elements in it are notes, rests and direction. It also contains information about time signature, key signature and tempo.

The timing mechanism inside a measure works by maintaining a time counter with time changing instructions. When a measure is entered, the time counter is set to zero. Then, when each note(or rest) is added, the time counter increase the amount of the note's duration. From this aspect, a note's occurrence can be viewed as a time changing instruction. To support polyphony music, MusicXML has the "backup" and "forward" instructions. When a backup Instruction is processed, the time counter decrease an amount stated by the instruction. A forward instruction is processed in a similar way, but make the time counter goes forward instead.

2.2.5 Note

A note element tells us the pitch, duration and display style of the note. Unlike the MIDI format, which represents a pitch by an integer proportional to the logarithm of the physical pitch value, pitch in MusicXML is represented by a (step, octave) pair. A step can be one of (C, D, E, F, G, A, B) and octave is an integer.

A note element also tells about accidentals, dots, stems, beams, etc. Further details will be explained in 2.5.

2.2.6 Other Symbols

There are some other types of symbols on a musical score, such as time signature, key signature, text, etc. These symbols are usually straightforward to handle since they do not depend on each other.

2.3 Internal Data Representation

We adopt an object-oriented style in our internal data representation design. As mentioned above, MusicXML has the score-part-measure-note hierarchy. Instead of following the MusicXML hierarchy strictly, we decide to have a sheet-page-measure-note hierarchy. Miscellaneous symbols is attached to measures and notes.

One challenge of dealing with MusicXML is that some information is optional, so we need to infer its default value from the context. For example, the distance between a measure and the measure below it may be omitted, then we need to infer this distance value from the previous measures. Sometimes we even have to hard-code default values inside our application. For example, tempo value is optional in MusicXML. However, without this value we cannot convert the music time to actual physic time, which is needed in the following work..

2.4 The Parsing Procedure

In the parsing procedure, a ParseContext is maintained. A ParseContext consist of these values:

- sheet: Current sheet.
- page: Current page.
- measure: Current measure.
- beams: A lookup table for beams parsing.
- endings: A lookup table for endings parsing, similar to beams.

To parse a MusicXML file, we follow the steps below:

1. Load the MusicXML schema file.
2. Load the MusicXML file.
3. If the file is compressed, decompress it.
4. Validate the file using the loaded schema.
5. Initialize the parse context.
6. Iterate through each measure element in the only part element.

7. For each measure, if it indicates that a new page should be created, create one.

8. For each child of the measure node, get the type of this child and use the corresponding handler to handle.

Most of these steps are straightforward: We just read data from the elements and store them to the objects.

2.5 Rendering Sheet Music

2.5.1 Overview

Sometimes a picture is worth a thousand words. Figure 2.1 is an overview of the elements on the score.

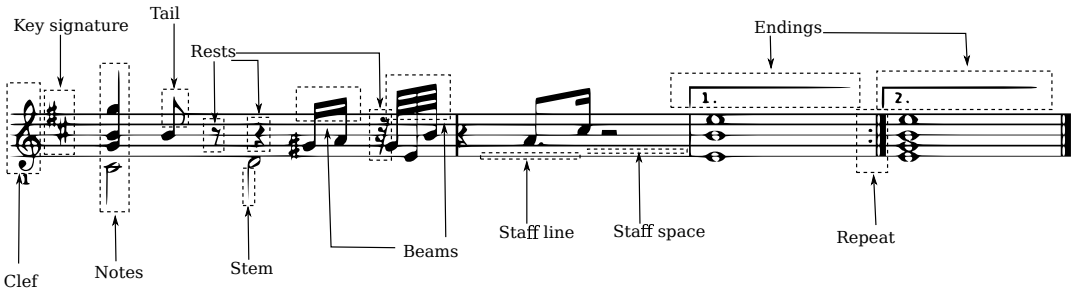

FIGURE 2.1: A sample score for illustration purpose

Elements are highlighted and labeled with names. This sample is rendered by our rendering program and then processed with Inkscape.

The fundamental part of a score are the staff lines. Each group of staff lines are five parallel long horizontal lines, forming a grid system for other musical elements. At the beginning of a staff, there's usually a clef followed by the key signature. For guitar pieces, usually the G clef is used. A staff is split into measures by the vertical bar lines. A measure helps the reader to group notes and to calculate the time. Each measure has its index number, counting from 0 or 1. It counts from 0 when the measure is partial. Although every measure has an index number, not all of them are printed and usually only the leftmost measure's index number is printed. With the aid of index number, readers can describe and locate measures efficiently.

The vertical bar lines splitting the measures may have different style. They may be thin or thick, single or double. Thin bar line is the one that used most frequently. They are used to split measures. Thick bar lines are used at the end of a score. Double bar lines that company with double vertical dots, denote start or end of a repeat.

Musical notes, along with the stem and beams connecting them, is the most complex part. Details about them will be covered in the following discussion.

2.5.2 Primitive Types

Before we continue on discussing how to render different musical elements, it's worthwhile to classify different primitives.

A typical musical score are printed using only the black color. So in our model all primitives are rendered using the same color, black.

In our model, all musical elements can be decomposed into these four primitives:

- Lines: Both vertical and horizontal.
- Beams: Beams is a special kind of line, with their two ends slanted.
- Texture: For example, clefs, quarter note heads, full note heads, tails, rests, accidentals.
- Text.

A line is actually drawn as a long thin rectangle. Formally, a "line" here should be called as a segment since it's not infinite. However, we call it a "line" for simplicity's sake. We use 5 variable to describe a line, they are:

$$(x_1, y_1, x_2, y_2, \text{thick})$$

This is illustrated by Figure 2.2. One remarkable thing is that we found it important to keep the line ends sharp instead of bevel. This is because bevel ends are hard to align with other elements, causing artifacts in the final render result.

Beams look like lines, except that their ends are slanted. Also, they are attached to the stems most of the time. If we keep using $(x_1, y_1, x_2, y_2, \text{thick})$ to represent a beam, it

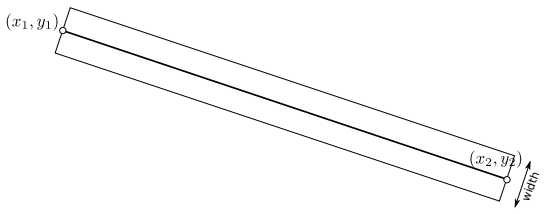

FIGURE 2.2: Illustration of line representation.

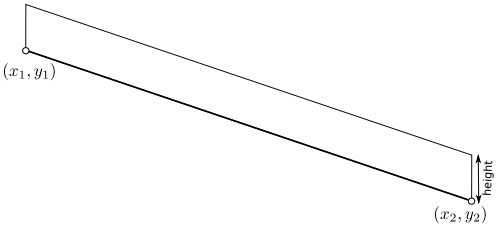

FIGURE 2.3: Illustration of beam representation.

FIGURE 2.4: Packed textures

Each template comes from the Lilypond project. They are rendered into SVG format and then converted to 500dpi PNG image.

results that we keep subtracting and adding $\mathrm{thick}/2$ when calculating its position. To solve this inconvenience, we use the following representation instead:

$$(x_1, y_1, x_2, y_2, \mathrm{height})$$

The meanings of these variable is illustrated by Figure 2.3.

To render other elements with irregular shape, we use the template method. First we decompose a complex element symbol into different parts, based on the following rule:

- If the symbol has long vertical or horizontal lines, remove it.
- For the remaining parts, create a template for each of them.

With this method, we get the templates as shown in Figure 2.4. Since we use OpenGL as the rendering backend, keeping all templates in a single texture image is quite a common way to go. However, packing different templates manually is not easy to maintain, so we make a script using a greedy algorithm to do this. With the help of the script, we can just make each template a single file with its own name, so that we can easily add or remove templates later.

Each template is represented by:

$$(x, y, w, h)$$

Each template has its pivot point (x, y). This is the coordinate on the final packed image. w stands for "width", and h for "height".

Texts are rendering in a similar way like textures, so we will not discuss it further here.

2.5.3 Note

Notes are the essential part of the score. The pitch of each note is specified by its vertical position and the duration is specified by both its head shape and its tail.

There exist 3 types of note heads. They're for full notes, half notes, and other notes, respectively. Full notes don't have stems connected, and the head is a bit larger than other notes. Half notes look like quarter notes, except that their heads are hollow. Notes with duration less than quarter has the same note heads since their duration is then not distinguished by their heads, but their tails instead. As a result, we need to prepare only 3 templates for note head rendering.

A note is usually connected by a stem, which heads up or down and has various length. The direction of a stem is told by MusicXML. Different directions of a stem is used to distinguish between different voice parts. While the head position of a stem is determined by the note head and the stem direction, the tail of it is determined by the beam attached to it, if it has any. A stem without beams can be any visually proper length. We use 35 tenths as this default value in our application.

A single stem may be connected to multiple notes. This is often seen in chords. Examples can be found in Figure 2.1.

2.5.4 Beams

Beams are for telling the duration of notes and for grouping notes. Grouped notes can express the indent of the composer. A note [3] without any beams but only a stem have the duration of $1/4$. When one more beam is attached, the duration is divided by 2. It's not hard to get the conclusion that a note with n beams have duration of $1/2^{n+2}$.

Usually, in musical score, beams has different slopes, which is determined by vertical positions of notes that they connect. However, the rules to determine the slope is seldom mentioned. Some software just let the slope be 0. This is, they render all beams horizontally, which makes the score look weird and don't comply with the musical convention.

Here comes three observations that build up the fundamental rules of our method:

1. Beams that connected to common stems and have common horizontal span should have the same slope.

[3] Here we are discussing notes with quarter note head since full notes and half notes don't have beams.

2. A proper slope for a beam should make the length of stems connecting to the beam as uniform as possible.

3. There should be only several slope value used in rendering since using too many slope value will mess up the score, which is not ideal.

A beam can be partial or complete. Partial beams can then be divided into forward and backward beams. Partial beams cannot occur by itself since a beam occur by itself is actually a tail, which is rendered differently. With this observation, we can assume that partial beams can always be drawn after complete beams.

A beam may be connected to up stems or down stems, but rarely both. We can assume that all stems that a beam connected to have the same direction. Then we have two types of beam, one with up stems and the other with down stems. Since the calculation between these two types are similar, from this point on we discuss about only the type that has up stems.

In our beam slope calculating algorithm, we have two important concepts:

- Length: the length of a beam is defined as the number of different stems that connected to it.
- Anchor: An anchor is a stem that its tail position is determined.

Note that if a beam's slope is calculated, the slope of beams below or above it should be the same value. So if we sort the beams by their length, from long to short, and then assign their slope, then we only need to calculate the topmost one in each group of beams. With the aid of anchors, sometimes we can determine the slope of a beam directly. Details is covered in the algorithm listing. 1.

After this algorithm is performed, we can determine both beam positions and stem positions relative to the measure.

2.5.5 Accidentals and dots

Accidental symbols may occur in two places. The first place is the key signature and the other is the area to the left of note heads. Dots that extending a note's duration may only occur to the right of the note. For accidentals and dots around the note head, it's quite straightforward to render. Once the note's position is determined, we just put them around.

Algorithm 1 Layout Beams and Stems

Sort beams by length, from long to short.
for beam $\in$ beams **do**
    anchors $\leftarrow$ Next available position from all up stems connected to beam.
    if length of anchors ≥ 2 **then**
        anchorL $\leftarrow$ leftmost anchor from anchors
        anchorR $\leftarrow$ rightmost anchor from anchors
        Set the two ends of the beam to anchorL and anchorR, respectively.
    else
        tails $\leftarrow$ tail positions of stems connecting to beam
        $e_m \leftarrow \inf$ ▷ Current minimum error value
        $A \leftarrow \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ 1 & 1 & \cdots & 1 \end{bmatrix}^T$
        $B \leftarrow \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix}^T$
        for $k \in \{-0.2, -0.1, -0.05, 0.05, 0.1, 0.2\}$ **do**
            bLimits $\leftarrow B - kA[:, 0]$
            $b \leftarrow max(\text{bLimits})$
            $\vec{x} \leftarrow (k, b)$
            $e \leftarrow |Ax - B|^2$
            if $e < e_m$ **then**
                $e_m \leftarrow e$
                $\vec{x}_m \leftarrow \vec{x}$
        $(k, b) \leftarrow x_m$
        $\vec{p_1} \leftarrow kx_1 + b$
        $\vec{p_2} \leftarrow kx_n + b$
        Set the two ends of the beam to $\vec{p_1}$ and $\vec{p_2}$, respectively.

One remarkable optimization for accidental positioning is that we found a way to avoid the overlap of close accidentals. One situation is illustrated by Figure 2.5. To do this, we first sort all accidentals in a measure, from bottom to top, then left to right. Then we start to put accidentals onto the measure by the sorted order. When an accidental overlap with other ones, move it to left by half of it's width, until there's no overlap. However, to avoid an accidental be moved too far from its original position, we constraint the count of moves to 5 for each accidental. Here, the collision detect can use the brute force way, with $O(n^2)$ time complexity, where n is the number of accidentals in the measure. Since there's always only a few accidentals in each measure, this method will work well.

For the accidental symbols used in key signature, there is a set of conventional rules. The writing of key signature is relevant to the clef. Here we only discuss the G clef situation, for the sake of simplicity. In most guitar pieces we only use the G clef.

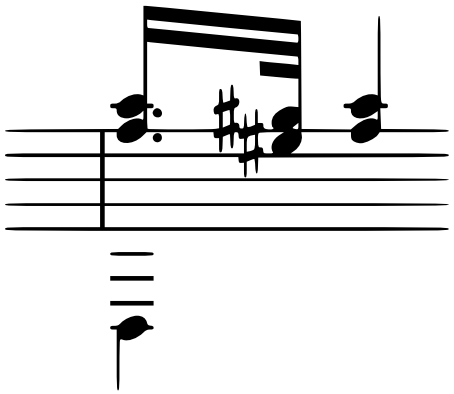

FIGURE 2.5: Avoiding accidental overlap

2.5.6 Key Signature

In MusicXML, the musical key is described by fifths and octave. When the center of G clef is aligned with the second staff line(counting from bottom to top), and if the octave is 3, we know that the second line correspond to the pitch G3. The variable fifths tells about how many fifths we have and what's the accidental type. If fifths is positive, then the type is sharp. If fifths is negative, then we have flat fifths. For example, if fifths is 1, we know that there is only one sharp, which means the key is G major. If fifths is 2, we know that there are two sharps, and the key is D major. If fifths is -2, we know that there are two flats, and the key is Bb major. The fifths variable can range from -7 to 7. Table 2.1 and 2.2 show the correspondence between fifths and key signatures.

Fifths	Key	Sharps
0	C major	
1	G major	F#
2	D major	F#, C#
3	A major	F#, C#, G#
4	E major	F#, C#, G#, D#
5	B major	F#, C#, G#, D#, A#
6	F# major	F#, C#, G#, D#, A#, E#
7	C# major	F#, C#, G#, D#, A#, E#, B#

TABLE 2.1: Table of sharp fifths

Fifths	Key	Flats
-1	F major	Bb
-2	Bb major	Bb, Eb
-3	Eb major	Bb, Eb, Ab
-4	Ab major	Bb, Eb, Ab, Db
-5	Db major	Bb, Eb, Ab, Db, Gb
-6	Gb major	Bb, Eb, Ab, Db, Gb, Cb
-7	Cb major	Bb, Eb, Ab, Db, Gb, Cb, Fb

TABLE 2.2: Table of flat fifths.

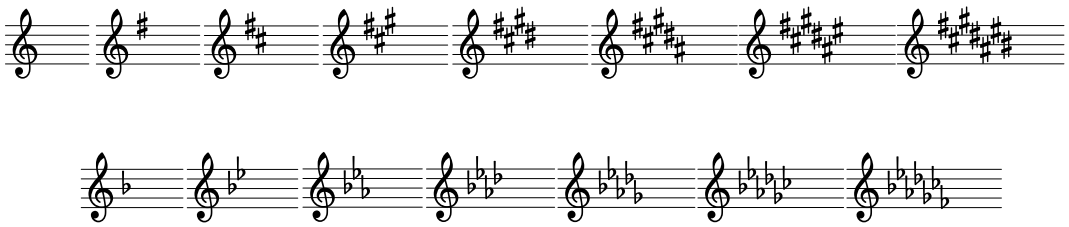

FIGURE 2.6: Different key signatures.

First line correspond to fifths with value range from 0 to 7. Second line correspond to fifths with value range from -1 to -7.

With these two table, we can know which sharp or flat should be added and their order. However these information is not enough for us to decide their position. For example, in G major we have one sharp fifths. Looking up from Table 2.1, we know that it's F. However, both the space above the first staff line and the fifth staff line are in pitch "F", so it's possible to put the sharp symbol on both places, or even more places. Actually, sound F in G clef key signature is by convention drawn on the fifth line. So we need another table, as shown in Figure 2.6. With this table, we now know the vertical position for each symbol to put.

2.5.7 Other Symbols

There are other symbols in a score, such as bar lines, endings, title, etc. We will not discuss them here since they are quite straightforward to handle.

2.5.8 Rendering Order

Here is a short summary of the rendering order of elements discussed above.

- Staff lines
- Clef
- Key signature
- Notes
- Beams and stems
- accidentals
- Bar lines
- Endings

2.6 Results

We will discuss the result through examples. Each of these examples are some consecutive measures from various scores, rendered by our rendering engine. Each example will be used to illustrate a feature of our rendering engine. All origin MusicXML scores are downloaded from the MuseScore website.

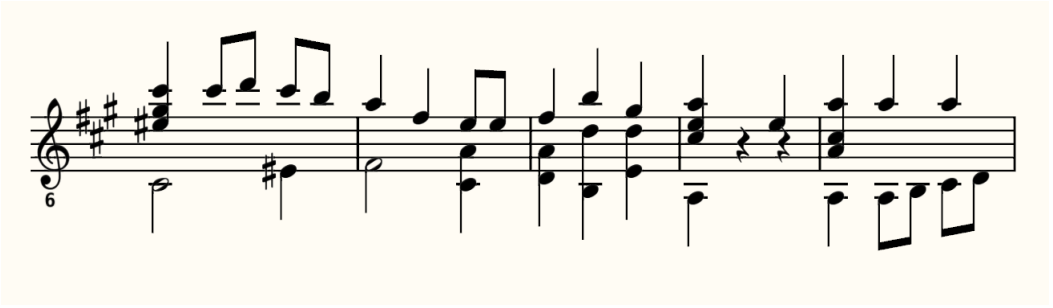

FIGURE 2.7: An overview of render result.

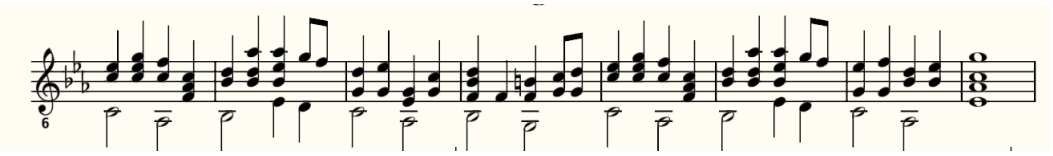

FIGURE 2.8: Example with dense measures.

The example illustrates the result of rendering chords. In a chord, multiple note heads are attached to the same stem. The length of a stem need to be extended properly to support more notes.

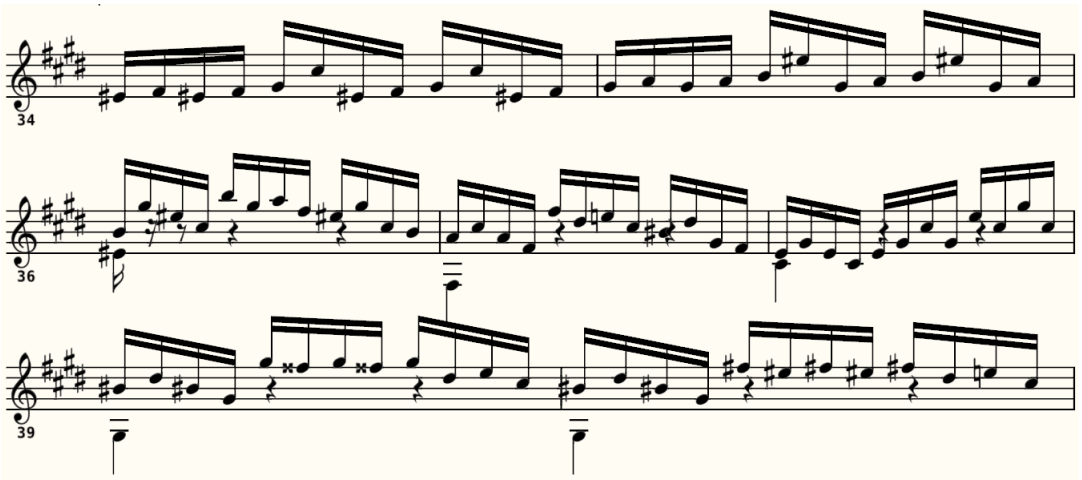

FIGURE 2.9: Render result for multiple beams.

In this example we can see that the slope of a beam is affected by the notes.

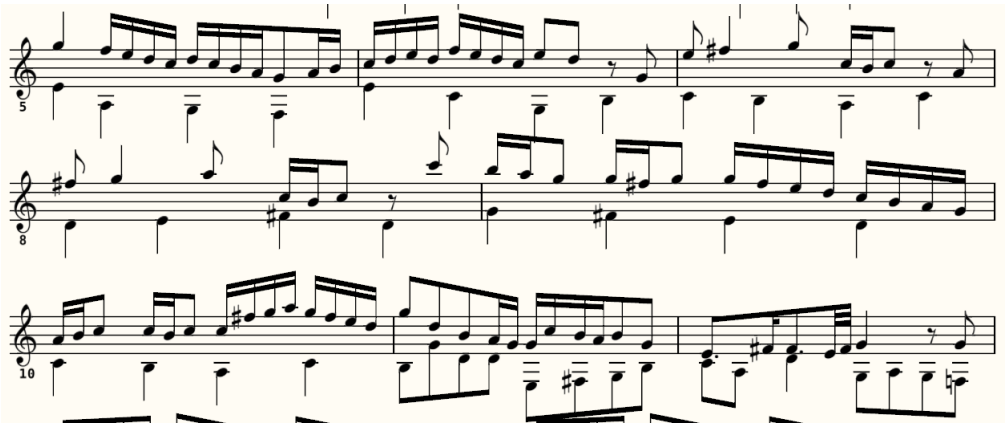

FIGURE 2.10: Measures that contains beams with different length in the same group.

In the first measure(with measure number 5), there is even more than one short beams under the long beam in the same group. We can see here our algorithm handled these situation properly.

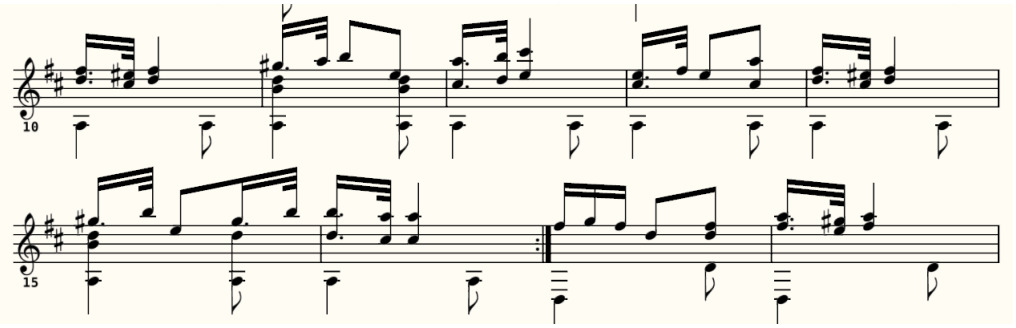

FIGURE 2.11: Example with complex beams.

In this example we can find that partial beams are rendered properly in groups with multiple levels of beam. In the first measure of second row, the second group of beams, which is more complex than others, is also rendered properly.

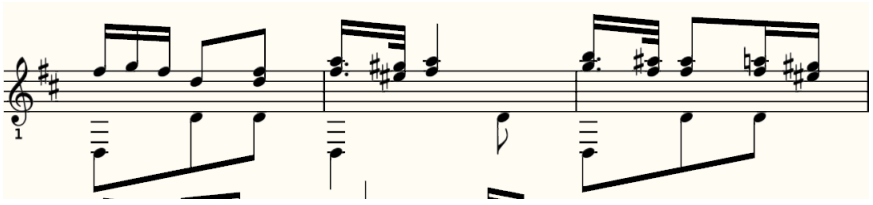

FIGURE 2.12: Example to illustrate the accidental positioning optimization.

In the second and third measure we can see that the sharp symbol are placed properly without overlapping. Even if they are very close to each other.

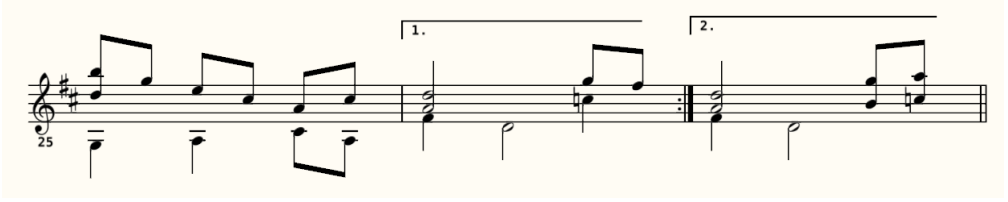

FIGURE 2.13: Example with endings.

Chapter 3

Playing Sound

Sound playing implemented using midi output in our application

Technically, a piece of music can be treated as a sequence of notes. In our model, a note is described by (pitch, start time, end time). The pitch format here is different from the pitch format in the rendering engine, which use (step, octave) to describe a pitch.

3.1 Pitch Conversion

We tell the midi output server the notes' pitch by a number ranged from 0 to 255. The standard sound A4, with pitch 440Hz, is represented by value 69 in the midi protocol. With this knowledge, we can convert the (step, octave) pair into the midi pitch level by the formula: `pitchLevel = index('C D EF G A B', step) + octave * 12 - 48 + 60`. Here the "index(list, element)" function is used to locate the element in the list. For example, index('C D EF G A B', 'C') = 0, index('C D EF G A B', 'E') = 4.

3.2 Time Conversion

Usually, under the musical context, the unit of time is beat. For example, when the time signature is 3/4, it means there are 3 beats in each following measure and the beat type is quarter beat. To convert to real world time, tempo is needed. Tempo tells us how many beats are there in one minute. Suppose we have a note with duration equals to t_d, and tempo equals to k. Then the real world duration can be calculated by:

$$t'_d = \frac{60 t_d}{k}$$

The result unit is in seconds. Using this formula, we can accumulate the durations to get the real world time relative to the beginning of each measure for each note.

There is another element that can affect the duration of a note's duration. They are dots. With a single dot, the duration of a note is multiplied by 1.5. However, with 2 dots, the duration is not multiplied by 1.5^2. Instead, the result duration will be $t_d * (1 + \frac{1}{2} + \frac{1}{2^2})$, where t_d is the origin duration. Similarly, with n dots, the converted duration will be:

$$t'_d = t_d \left(1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^n}\right) = t_d \left(2 - \frac{1}{2^n}\right)$$

3.3 Note Sequencing

To get the final sequence of notes, we need to resolve the repeats in a musical score. At first, all measures are chained together by in the order of their number. When we start to read a score, imagine the there is a cursor put at the beginning of the first measure. As we continue to read, this cursor starts to move forward. When a end of repeat is encountered, the cursor need to jump back to the corresponding start of repeat. When this repeat is encountered again, it should be ignored. An example is illustrated by Figure 3.1. Sometimes there may be endings come along with the repeat, this is illustrated by the example in 3.2. Other kinds of repeat exists such as D.C.al Find(da capo al fine), etc. We will not discuss them here.

FIGURE 3.1: An example for typical repeat.
The read order should be 1,2,3,2,3,4.

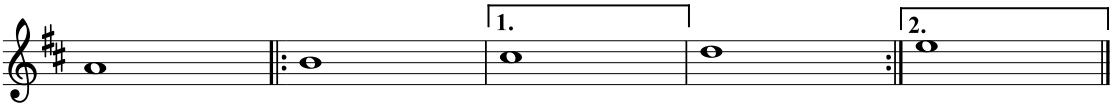

FIGURE 3.2: An example for repeat with endings.
The read order should be 1,2,3,4,2,5.

3.4 Putting All Together

The midi protocol support two types of event: note_on and note_off. We tell the midi server to start playing a sound with note_on(pitch, volume, channel) and use note_off(pitch, volume, channel) to stop the corresponding sound later. To work with the midi server,

we need to convert the node sequence to note events. Each note will generate exactly two events, one note_on event and one note_off event.

In our implementation, each event is described by (event_type, time, note). To convert from notes to events, we first generate $2n$ events from the n notes. Then, the events are sorted by their time. Note that there maybe some different events have the same time. In this case, we use the event type as the comparison key. More specifically, the note_off event is considered "less" than the note_on event. Otherwise, we are not able to play consecutive notes with same pitch.

The sound playing algorithm is shown in Algorithm 2. To make it sound more fluently, we use an extra thread to run the code with implementing algorithm.

Algorithm 2 Algorithm for Sound Playing

```
  procedure PLAY_SOUND(events)
Require: events sorted by (time, event_type).
    t ← 0                                              ▷ Current time
    for event ∈ events do
        Δt ← event.time − t
        sleep(Δt)
        level ← convert_pitch_level(note.pitch)
        if event.type is note_on then
            note_on(level, 127, 1)
        else
            note_off(level, 127, 1)
```

Chapter 4

Fingering Arrangement

4.1 Introduction

A guitar consist of several parts. In this problem, we focus on the fretboard only. Figure 4.1 shows names of the part that we interest about.

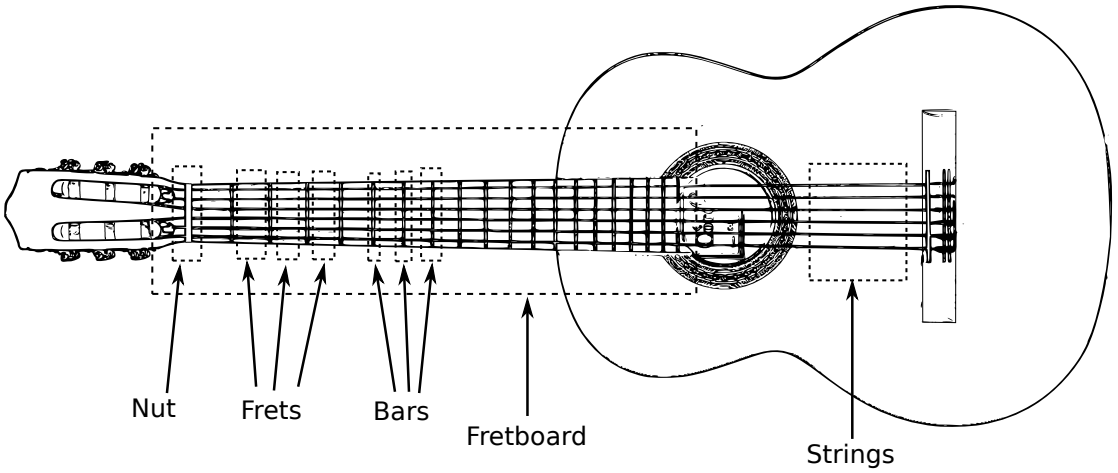

FIGURE 4.1: Classical guitar

The fretboard can be viewed as a grid system split by 6 strings and 17(or more) bars. For consistence, we assume there is 17 bars on the fretboard. A grid can be described by the (fret, string) pair, where fret ranges from 0 to 17 and string ranges from 0 to 5. The frets count from left to right in Figure 4.1, and strings count from top to bottom.

When playing the guitar, one use the left hand to press a grid, then use right hand to pluck the corresponding string, then a sound can be produced. Some sounds can be produced by plucking the string without pressing the grid since all strings are attached to the nut. This is so-called "play by empty string". We consider fret 0 is used in the empty string situation.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	$E_4(64)$	$F_4(65)$	$F_4^{\#}(66)$	$G_4(67)$	$G_4^{\#}(68)$	$A_4(69)$	$A_4^{\#}(70)$	$B_4(71)$	$C_5(72)$	$C_5^{\#}(73)$	$D_5(74)$	$D_5^{\#}(75)$	$E_5(76)$	$F_5(77)$	$F_5^{\#}(78)$	$G_5(79)$	$G_5^{\#}(80)$	$A_5(81)$
1	$B_3(59)$	$C_4(60)$	$C_4^{\#}(61)$	$D_4(62)$	$D_4^{\#}(63)$	$E_4(64)$	$F_4(65)$	$F_4^{\#}(66)$	$G_4(67)$	$G_4^{\#}(68)$	$A_4(69)$	$A_4^{\#}(70)$	$B_4(71)$	$C_5(72)$	$C_5^{\#}(73)$	$D_5(74)$	$D_5^{\#}(75)$	$E_5(76)$
2	$G_3(55)$	$G_3^{\#}(56)$	$A_3(57)$	$A_3^{\#}(58)$	$B_3(59)$	$C_4(60)$	$C_4^{\#}(61)$	$D_4(62)$	$D_4^{\#}(63)$	$E_4(64)$	$F_4(65)$	$F_4^{\#}(66)$	$G_4(67)$	$G_4^{\#}(68)$	$A_4(69)$	$A_4^{\#}(70)$	$B_4(71)$	$C_5(72)$
3	$D_3(50)$	$D_3^{\#}(51)$	$E_3(52)$	$F_3(53)$	$F_3^{\#}(54)$	$G_3(55)$	$G_3^{\#}(56)$	$A_3(57)$	$A_3^{\#}(58)$	$B_3(59)$	$C_4(60)$	$C_4^{\#}(61)$	$D_4(62)$	$D_4^{\#}(63)$	$E_4(64)$	$F_4(65)$	$F_4^{\#}(66)$	$G_4(67)$
4	$A_2(45)$	$A_2^{\#}(46)$	$B_2(47)$	$C_3(48)$	$C_3^{\#}(49)$	$D_3(50)$	$D_3^{\#}(51)$	$E_3(52)$	$F_3(53)$	$F_3^{\#}(54)$	$G_3(55)$	$G_3^{\#}(56)$	$A_3(57)$	$A_3^{\#}(58)$	$B_3(59)$	$C_4(60)$	$C_4^{\#}(61)$	$D_4(62)$
5	$E_2(40)$	$F_2(41)$	$F_2^{\#}(42)$	$G_2(43)$	$G_2^{\#}(44)$	$A_2(45)$	$A_2^{\#}(46)$	$B_2(47)$	$C_3(48)$	$C_3^{\#}(49)$	$D_3(50)$	$D_3^{\#}(51)$	$E_3(52)$	$F_3(53)$	$F_3^{\#}(54)$	$G_3(55)$	$G_3^{\#}(56)$	$A_3(57)$

TABLE 4.1: Fretboard pitch table on standard tuning

Number in the parentheses are the midi pitch level.

Four fingers in ones left hand are used for pressing the fretboard. Thumb is not used in this job, it is used to grip the neck of guitar. In our discussion, fingers of left hand are numbered into 0, 1, 2, 3, where 0 is for the index finger, and so on. A single finger can press only one grid at a time, except the index finger. We can use the index finger to press more than one grid at a time. This technique is called "baring". It cannot be used in every combination of grids. Instead, it can only be used to press the grids on the same fret and a consecutive range of strings. Figure 4.2 shows an example of the baring technique.

FIGURE 4.2: Baring by the index finger.

In this example, fret 5 of strings (0, 1, 2) is pressed by the index finger.

Normally, without special techniques, each grid on the fretboard can only produce a specific pitch of sound. For example, The grid (5, 0) can produce sound A4(440Hz) and grid (1, 0) can produce C4. Table 4.1 shows the pitch that can be produced by each grid on the fretboard. From this table we see that the pitches on the same string are ascending uniformly. Once the pitch of grid (0, i) is known to be p_i, the pitch of grid (f, i) can be calculated by $p_i + f$.

The problem of fingering arrangement is to assign a grid and a finger for each desired notes that we want to produce.

In Table 4.1, we can see that different grids may produce same pitch. For example, both grids (5, 5) and (0, 4) can produce A2, and both grids (1, 1), (5, 2), (10, 3), (15, 4) can produce C4. Actually, almost every pitch can be produced by more than one grid. Also, different grids can be pressed by different fingers. All these make it impractical to enumerate all possible combinations to find the optimal solution. By optimal we mean the one that is most easy to play. This will be defined formally in the next section.

4.2 Model

In this problem, we describe a note by (pitch, start time, end time). The solution to the problem should resolve each note by assigning a (finger, fret, string). Since a note can also be played by empty string, the finger value in (finger, fret, string) tuple may equal to -1, which means no finger.

$$\text{fingerings} \leftarrow \text{Arrange}(\text{noteSequence})$$

4.2.1 Key Times

A key time is a time that one or more notes start. Key times are important because all the finger movements start at key time. The first step to solve this problem is to extract all start times from the note sequence as the key time collection. The key time collection is a sequence of different key times.

4.2.2 Frame

A frame at time t is defined as the collection of notes whose time interval contains t. More formally, suppose the note is represented by (p, t_1, t_2), and the note sequence is denoted by N, then the frame is

$$\text{frame(t)} = \{(p, t_1, t_2) | (p, t_1, t_2) \in N, t_1 \leq t < t_2\}$$

In our method, only frames at key times are interested. All frames are generated by a scanning algorithm, which is described in Algorithm 3. First we convert the notes into note events, as we do in chapter 3. Then during the scanning process, we use a list to keep track of the active note events. At time t, from the active events we know the notes in the current frame.

4.2.3 Play State

A play-state at time t describes the state of the left hand and the ringing strings. With these information, we will be able to calculate the cost at time t or cost between two successive key time.

A play-state can be represented by

$$(b, F(i), S(i), R(j))$$

Algorithm 3 Generate Frames

function GENERATE FRAMES(T, E) ▷ T is the key time sequence. E is the note event sequence.
Require: E is sorted by start time.
F ← empty set ▷ The frame sequence.
E_a ← empty list ▷ Active events.
$p \leftarrow 0$
for $t \in T$ **do**
$E_a \leftarrow \{e | e \in E_a, e.\text{end} > t\}$
while $p < \text{length}(E)$ and $E(i).\text{start} == t$ **do**
Append $E(i)$ to E_a
$p \leftarrow p + 1$
make a frame using E_a and then append it to F
return F

Where b tells this state is barring or not; F and S are functions and their domain is the finger collection $0, 1, 2, 3$. In this state, finger i is pressing the grid $(F(i), S(i))$. R is also a function and its domain is the string collection $0, 1, 2, 3, 4, 5$. $R(j) = \text{True}$ means that in this state, string j is used.

In our method only play-states at key times are interested. A valid play-state for time t is a play-state that can produce all notes in $\text{frame}(t)$. At each key time, we may have multiple valid play-states. The solution of the fingering arrangement problem can be described by a sequence of play-states. In such a solution, a play-state is chosen for each key time.

Suppose in a solution, play-state S_i is assigned to the i-th key time t_i. Since a play-state tells us all information about how to put the left hand, knowing S_i and S_{i+1} is sufficient to let the player move the left hand. The cost at all play-state S_i, and between all S_i, S_{i+1} pairs, are summed up to get the total cost for this solution, using equation 4.1.

$$\begin{aligned} &\text{SolutionCost}(\{t_0, t_1, \cdots, t_n\}, \{S_0, S_1, \cdots, S_n\}) \\ = \;& \textstyle\sum_{i=0}^{n} C_{\text{static}}(S_i) + \sum_{i=0}^{n-1} C_{\text{transition}}(t_i, t_{i+1}, S_i, S_{i+1}) \end{aligned} \tag{4.1}$$

The cost functions $C_{\text{static}}(S_i)$ and $C_{\text{transition}}(t_i, t_{i+1}, S_i, S_{i+1})$ will be discussed in section 4.4.

4.2.4 State Transition Equation

We use a dynamic programming method to find the play-states solution.

Let $C(t_i, S)$ be the minimum cost to stay at play-state S at key time t_i. Then we can calculate $C(t_i, S)$ using dynamic programming. All valid play-state for each frame is generated before we calculate $C(t_i, S)$. We denote the valid play-state for frame at time t_i as V_i.

$$C(t_i, S) = C_{\text{static}}(S) + \min_{S' \in V_{i-1}} \left[C(t_{i-1}, S') + C_{\text{transition}}(t_{i-1}, t_i, S', S) \right] \tag{4.2}$$

Our goal is to find the play-state S such that $C(t_n, S)$ has the minimum value. Once we got such an S, we can trace back the play-state sequence by the decision on every step to get the corresponding play-state sequence.

4.3 Generating Valid play-state

A play-state is considered valid if and only if it's playable. Human hand has physical constraint, so some of the grid combinations are impossible to play. Suppose we now have a play-state $S = (b, F(i), S(i), R(j))$, the constraints are listed below.

1. For any $i, j, i \leq j$, we have $F(i) \leq F(j)$.
2. For any $i, j, i \neq j$, we have $S(i) \neq S(j)$.
3. For any $i, j, i \leq j \wedge F(i) = F(j)$, we have $S(i) > S(j)$.
4. When $b = True$, then for any $i > 0$, we have $F(i) > F(0)$.
5. Fingers should not be too far away from index finger.
6. When $b = True$, fingers should not be too close from index finger.

Here we set the constraint distance value to maximum or minimum distance that our hand can stretch. The values we used in our implementation are listed in Table 4.2.

Fingers	1	2	3
max	0.06	0.10	0.13
min	0.008	0.025	0.025

TABLE 4.2: Finger constraint distance values(in meters).

To calculate the distance between any two grids, we need to know every bar positions. A guitar produces sound by the vibration of strings. When a string is pressed onto the fretboard, its length that contribute to the sound generation is effectively shortened.

The frequency of the sound that a string produced is proportional to the multiplicative inverse of the effective length of the string. From the design of guitar, we know that the position of the 12th bar is in the exactly middle of a string. If the length of a string is L, then the distance d_i of the i-th bar can be calculated by solving equations 4.3.

$$\begin{cases} l_i = L - d_i \\ l_{12} = L - \dfrac{L}{2} \\ l_i p_i = l_{12} p_{12} \\ p_i = p_{12}(\sqrt{2})^{\frac{i}{12}} \end{cases} \tag{4.3}$$

Solving these equations we got

$$d_i = (1 - 2^{-i/12})L$$

With this formula we can calculate the distance of any pair of grids.

To generate all valid states for a frame efficiently, we use the strategy that removes the invalid states as early as possible. First, the fret for index finger is chosen. With the index finger chosen, the remaining search space become much smaller. Then we choose to use the index finger or not. Even if the index finger is not used, the fret position of it can describe the hand position. If the index finger is used, a string will be assigned to it. After these steps, the search space is small enough so we can use a brute force recursive searching method to resolve the remaining notes.At each level of recursion, we resolve a note at current frame, and then test the validness according to the constraints. If all the notes are resolve at some level, we start to assign fingers to the used grids. Finally, a new valid play-state is generated.

4.4 Cost Function

We have two types of cost functions. The first type is the static cost function. The other type is transition function. Static cost is only relevant to frame and play-state at a single key time, while transition cost is relevant to two key times.

4.4.1 Static Cost

Static cost function consists of two parts: the "missing penalty" and the "high fret penalty". Missing notes in a play-state is heavily penalized. A note is considered missing in a play-state when it's not resolved by the state. In Figure 4.1, we can see that frets

higher than the 12th bar is on the front surface of the guitar. This make it more difficult to use the frets there, because there's no place for the thumb to grip the neck(it's out of the neck!). Therefore, we have penalty on high frets. In our implementation, fret higher than 9 are considered high. Summarizing these two types of cost, and suppose play-state to be (b, F, S, R) we get:

$$C_{\text{static}}(S) = n_{\text{miss}} K_{\text{miss}} + \sum_{F(i) > F_{\text{high}}} (F(i) - F_{\text{high}}) K_{\text{high}}$$

Where K_{miss} and K_{high} are constants we got by experiment.

4.4.2 Transition Cost

Transition cost describes the cost when the hand translates from one state to another state. It can be calculated by summing the movement cost of the arm and the movement cost of each finger. Suppose we are to calculate $C_{\text{transition}}(t_1, t_2, S_1, S_2)$. The predicate function $P(S, i)$ tells if finger i is pressed in play-state S. Then, for finger i from fingers other than index, we observe these 4 situations:

$\neg P(S_1, i) \wedge \neg P(S_2, i)$
: A free transition, finger i is not used in both play-state. No cost to add.

$\neg P(S_1, i) \wedge P(S_2, i)$
: Finger i need to pressed on the grid $(F_2(i), S_2(i))$ before time t_2. From S_1 we can know that when should string $S_2(i)$ stop, then we can calculate the time Δt the finger can use to press. Add $K_{\text{press}}/\Delta t$ to total cost.

$P(S_1, i) \wedge P(S_2, i)$
: The finger is pressing some grid in both states. Note that the grids it press may be the same or different. If they are the same, add $K_{\text{keep}}(t_2 - t_1)$ to total cost. If they are different, the finger is moved from grid $(F_1(i), S_1(i))$ to $(F_2(i), S_2(i))$, so we add

$$K_{\text{movefret}} \left[(F_1(i) - F_1(0)) - (F_2(i) - F_2(0)) \right]^2 + K_{\text{movestring}} \left(S_1(i) - S_2(i) \right)^2$$

$P(S_1, i) \wedge \neg P(S_2, i)$
: Finger i should be released before time t_2. The time interval Δt that the finger can use to release is calculated similar to the one we use to calculate pressing time. Add $K_{\text{release}}/\Delta t$ to total cost.

For the index finger, which has three states, the cost calculation is similar, but a bit more complexed. It has 9 situations but some are overlapped with the normal fingers'. We are not going to list them here.

One other cost is the missing penalty. If a note is in both frames at t_1 and t_2, and is resolved by different grid or finger, then this note is considered missing. This is because we cannot keep the string vibrating when switching to another finger. This kind of missing is also penalized by $n_{\text{miss}}K_{\text{miss}}$.

4.5 Decision Process

From equation 4.2, we can see that once we know all $C(t_{i-1}, S')$ for any S', we are able to calculate $C(t_i, S)$. $C(t_0, S)$ is calculated by taking only the static cost. Using math induction, we can calculate $C(t_n, S)$ for any S. Finally we got Algorithm 4

Algorithm 4 Calculate Cost

procedure CALCULATE COST(T, F)
▷ T: key times, F: key frames
Generate valid states V_i for each frame F_i.
D ← empty table ▷ Choice table
C ← empty table ▷ Cost table
for $S \in V_0$ **do**
$C(t_0, S) \leftarrow C_{\text{static}(S)}$
for $i \in [1, n]$ **do**
for $S \in V_i$ **do**
$C(t_i, S) \leftarrow \min_{S' \in V_{i-1}} [C(t_{i-1}, S') + C_{\text{transition}}(t_{i-1}, t_i, S', S)]$
$D(t_i, S) \leftarrow \arg\min_{j, S_j \in V_{i-1}} [C(t_{i-1}, S_j) + C_{\text{transition}}(t_{i-1}, t_i, S_j, S)]$

Aside from calculating the minimum value, the corresponding choices are also saved. Using these choices, we can recover the corresponding play-state sequence, which is our goal to the fingering arrangement problem.

In this algorithm, both time and space complexity are $O(nm^2)$, where m is the average number of valid states for each frame. By experiment, we found that n is usually 500 to 1000, and m varies from 1 to 100.

One important optimization for the decision algorithm, is to rearrange the order of all available S' so that unnecessary are decreased. For each $C(t_i, S)$, we make a copy of V_{i-1}, then sort it by the delta index fret. When current minimum value is less than the minimum possible value of each remaining candidate play-state S', there's no need to compare for these candidates. The optimized algorithm is shown in Algorithm 5

Algorithm 5 Calculate Cost

procedure CALCULATE COST(T, F)

▷ T: key times, F: key frames

Generate valid states V_i for each frame F_i.
D ← empty table ▷ Choice table
C ← empty table ▷ Cost table
for $S \in V_0$ **do**
  $C(t_0, S) \leftarrow C_{\text{static}(S)}$
for $i \in [1, n]$ **do**
  for $S \in V_i$ **do**
    $C(t_i, S) \leftarrow +\infty$
    order ← $(0, 1, \ldots, |V_{i-1}|)$
    order.sort(key=lambda j: $C(t_{i-1}, V_{i-1,j}) + \text{deltafretcost}(V_{i-1,j}, S)$)
    for $j \in$ order **do**
      $S' \leftarrow V_{i-1,j}$
      if $C(t_{i-1}, S') + \text{deltafretcost(S', S)} \geq C(t_i, S)$ **then**
        break
      $x \leftarrow C(t_{i-1}, S') + C_{\text{transition}}(t_{i-1}, t_i, S', S)$
      if $x < C(t_i, S)$ **then**
        $C(t_i, S) \leftarrow x$
        $D(t_i, S) \leftarrow j$

4.6 Results

We choose 4 classical guitar pieces and have our method tested upon them. For each piece, we cut some measures out and run our program to get the fingering. Then we generate the tablature and put them together with the origin scores. In our method, each note is assigned a finger for it. However, due to the limit of tablature, the finger is not label in the result score.

These results are in Figure 4.3, 4.4, 4.5 and 4.6. These test cases are chosen to test the performance of our algorithm on different musical style.

Lágrima in 4.3 is slower piece. Since the cost functions are time-relevant, we should its performance on pieces of different speed.

Giuliani Op.50 No.1 in 4.4 has two part of sound. The treble is fast and its every notes can be group to a chord. The right way to play this style is to have the left hand keep pressing the chord for each group of notes. In our result, this is arranged properly.

Lute Suite No.1 in E Major in 4.5 is even faster than the Giuliani Op.50. However, its notes on treble part is not consists of chords. To play it properly, the left hand need to change quite a lot. Since it is so fast that the successive note with different pitch should better be played on different strings to decrease the burden of the left hand. If several

successive notes are played on the same string, the left hand finger will have less time to press. The Δt_{press} in our cost model can reflect this situation.

Pavane No.6 for Guitar by Luis Milan in 4.6 is a slow piece. However, it's actually more difficult to play since the notes on it are dense due to the extensive use of chords. For example, the first measure of it has 3 beats and each beat has a chord that consists of four notes. We can see that the fingering arrangement of our result is proper and playable.

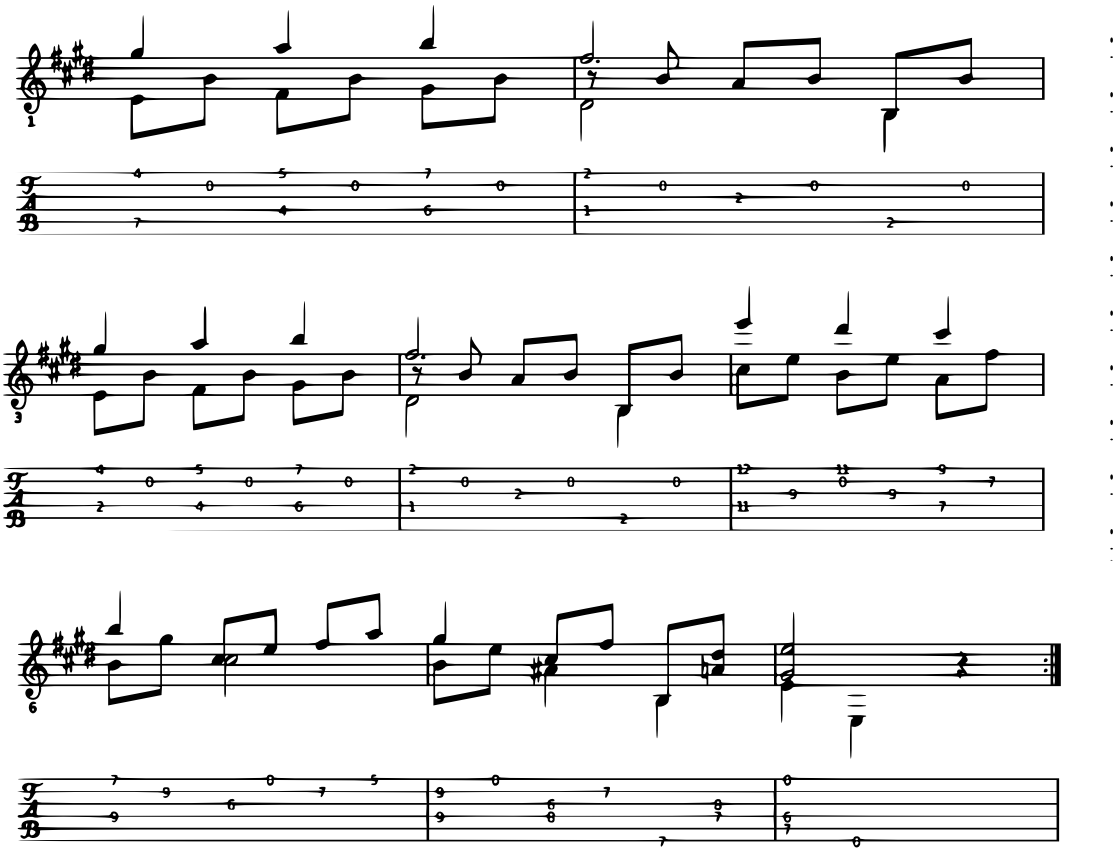

FIGURE 4.3: Lágrima by Francisco TÁRREGA

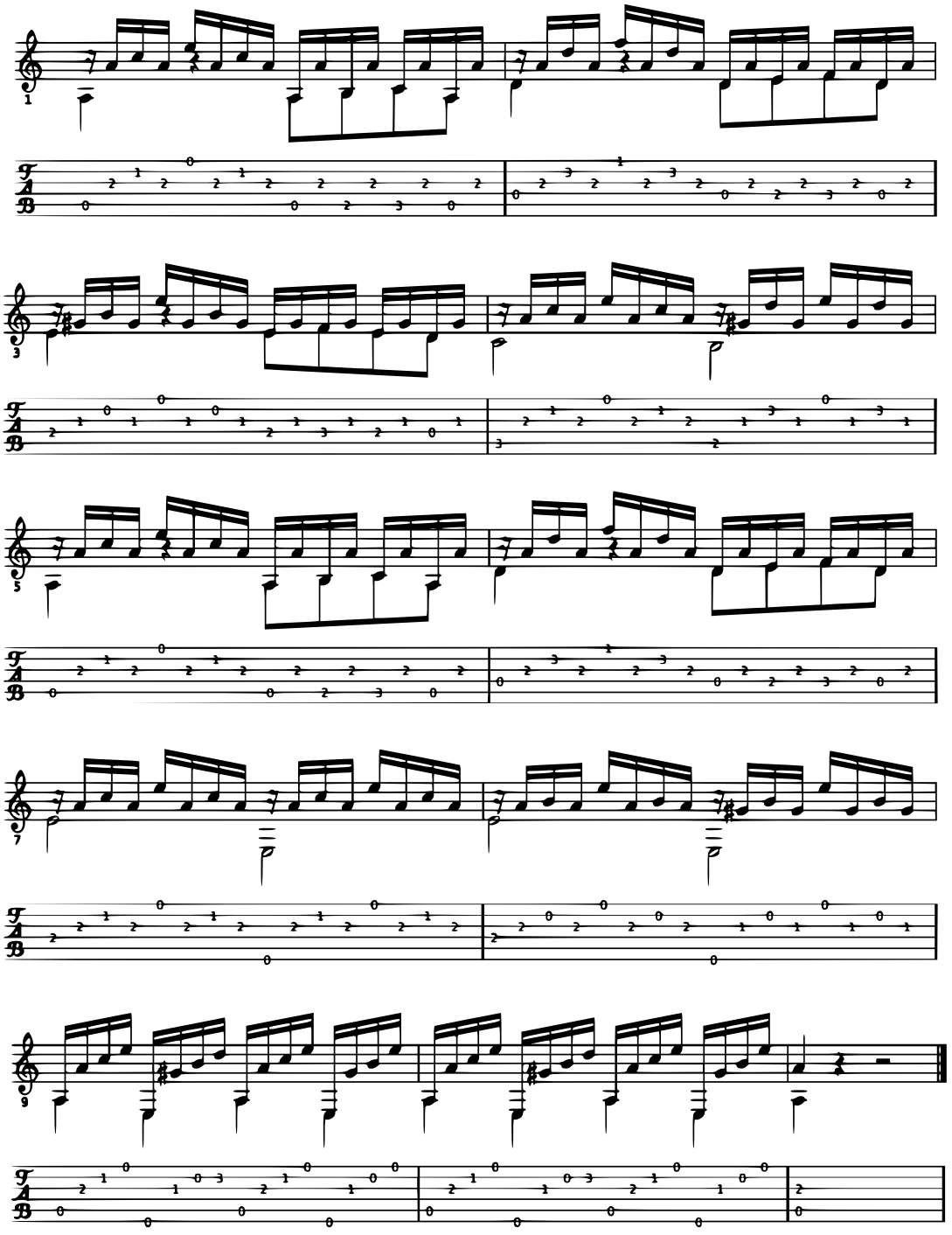

Figure 4.4: Giuliani Op.50 No.1

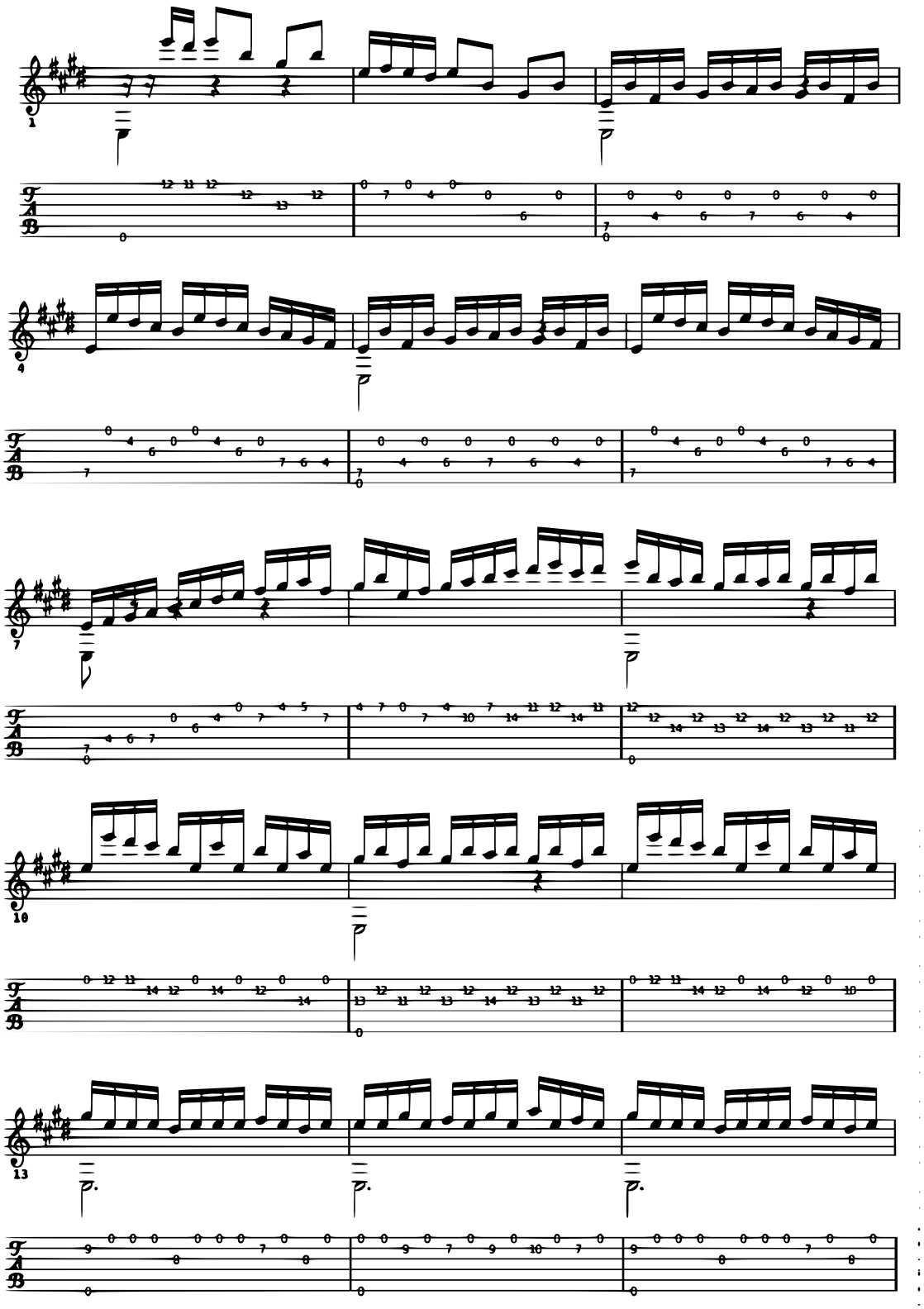

FIGURE 4.5: Lute Suite No.1 in E Major BWV 1006a by J.S. Bach

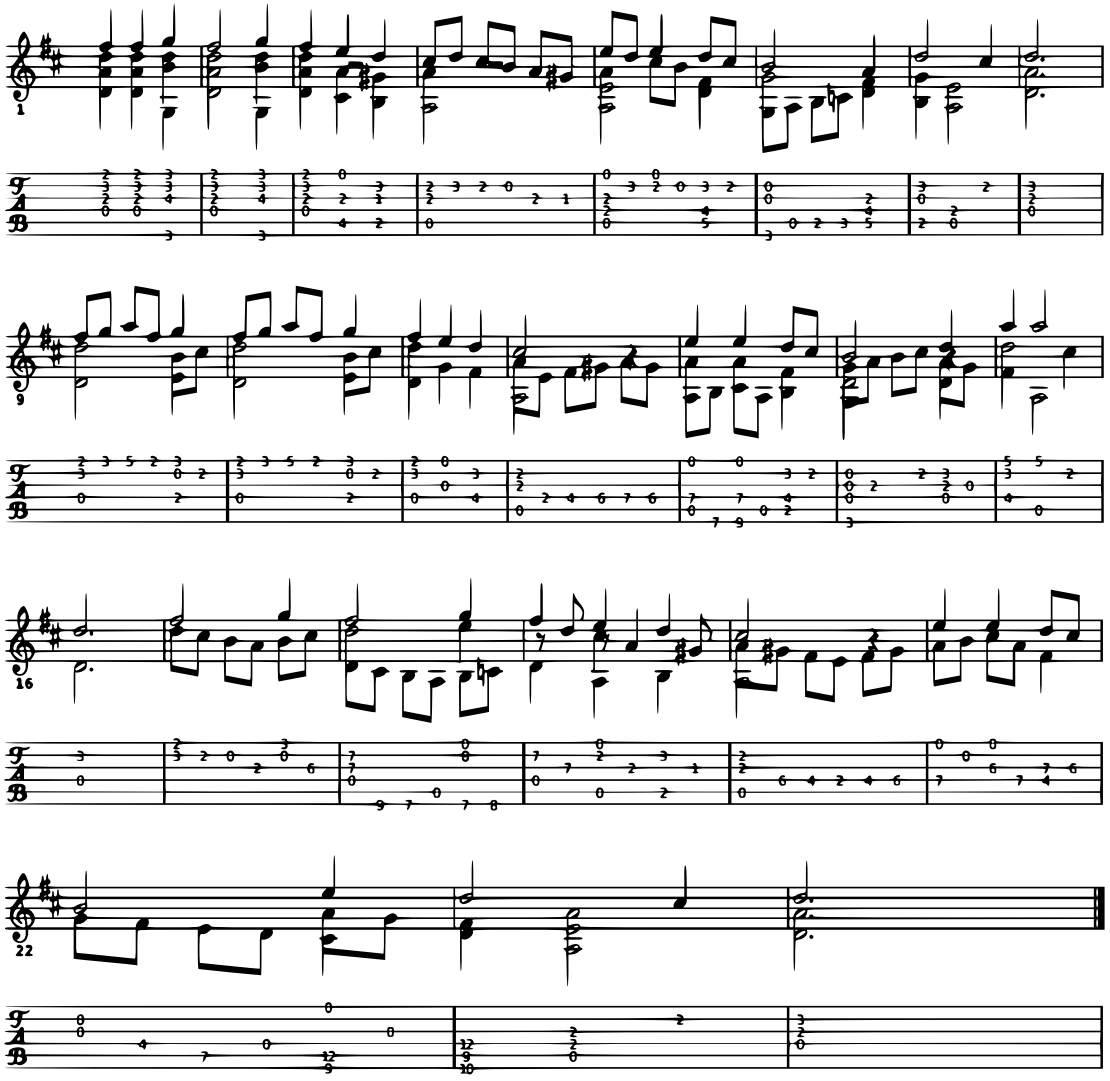

FIGURE 4.6: Pavane No.6 for Guitar by Luis Milan

Chapter 5

Summary

In the previous chapters, we described the most significant works in our application.

The first problem we solved is rendering of sheet music. Scores are read from MusicXML file, which provide the necessary information for rendering. However, not all information is given directly. Some information are context relevant and some need to be inferred by conventional rules. The most challenging part of the rendering problem is to determinate the slope and stacking for beams. It is solved b y sorting and iterating techniques. At last we got quite pleasing results.

Having audio playback in musical applications is always helpful. The second problem we solved is playing sounds. The most important step is to generate the note sequence from the score. The generated note sequence is also used to solve the fingering arrangement problem.

The next problem we solved is the fingering arrangement problem. We proposed a expressive model and than used dynamic programming to solve the problem. The most challenging part of this method is to calculate the cost functions. We set up essential rules to formulate the calculation of cost functions. In the cost model we have several parameters that were unknown at the beginning. They are found out by experiment later. The results we got are usually properly arranged and playable. However, they are not always the same with the published version. It is due to that the rules in our cost model can not reflect some conventions.

To further improve the work, we should add tie note and special playing techniques to the model. Special techniques such as hitting, picking and sliding may affect the fingering arrangement. Taking them into our model can improve the fingering arrangement result.

Appendix A

Setup Instruction

Guitarist, the application, is written in the Python 3 language. The rendering engine is powered by OpenGL and sound playing is powered by the midi module in Pygame.

The whole application is splited into three projects, namely:

- raygllib
- pysheetmusic
- guitarist

.

And here's the third party library dependencies:

- PyOpenGL
- Pyglet
- Pygame
- Numpy
- Cython

For now, the application has been tested on Linux only. To install, use `pip install` or `easy_install` to install the third party dependencies. Then, install `raygllib` and `pysheetmusic`, which are normal python packages.

To run the application, just run the `main.py` in project `guitarist` as python script.

Bibliography

[1] Masanobu Miura, Isao Hirota, Nobuhiko Hama, and Masazo Yanagida. Constructing a system for finger-position determination and tablature generation for playing melodies on guitars. *Systems and Computers in Japan*, 35(6):10–19, 2004.

[2] Samir I Sayegh. Fingering for string instruments with the optimum path paradigm. *Computer Music Journal*, pages 76–84, 1989.

[3] Aleksander Radisavljevic and Peter Driessen. Path difference learning for guitar fingering problem. In *Proceedings of the International Computer Music Conference*, volume 28. sn, 2004.

[4] Daniele Radicioni, Luca Anselma, and Vincenzo Lombardo. A segmentation-based prototype to compute string instruments fingering. In *Proceedings of the Conference on Interdisciplinary Musicology, Graz*. Citeseer, 2004.

[5] George ElKoura and Karan Singh. Handrix: animating the human hand. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 110–119. Eurographics Association, 2003.

[6] Michael Good. Musicxml for notation and analysis. *The virtual score: representation, retrieval, restoration*, 12:113–124, 2001.

[7] Michael Scott Cuthbert and Christopher Ariza. Music21: A toolkit for computer-aided musicology and symbolic music data. In *ISMIR*, pages 637–642, 2010.

[8] Han-Wen Nienhuys and Jan Nieuwenhuizen. Lilypond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*, pages 167–172. Citeseer, 2003.