Real-Time Conducting Animation from Sheet Music

By

Kseniya Rychkova

CSC 498: Mentored Research
Date of Submission: May 18, 2020
Mentor: Dr. Salgian

ABSTRACT

MusicXML is a recently developed digital sheet music format that can be easily parsed using an XML parser. The information found in a MusicXML was used to create a conducting animation in Java, which presents the conducting components of the musical piece such as time signature, tempo, dynamics, and rest measures. The animation is displayed in sync with the music, and is created specific to the music piece upon inputting the MusicXML file. This can be used as a conducting assistant to student musicians, or as a visual augmentation to a live orchestral performance.

Keywords

Sheet music, Java Swing animation, MusicXML, Conducting animation

1. INTRODUCTION

With many aspects of daily life, professions, and entertainment turning to technology for enhancements, the musical industry began its advent into digitalization many years ago. Orchestras have begun incorporating digital visual effects like projection of images into their performances, using techniques such as Architectural Project Mapping [1]. In his article discussing video technology in orchestras, Brian Wise says, "Successful video projects broadly depend on two factors: Suitable visual content and the technical coordination with the music" [1]. This paper explores a way to combine sheet music with animation such that the animation is specific to the music piece and synchronous with the music and the gestures a conductor would make, and suggest a new possibility for uniting technology and classical music.

The conducting animation can be useful for helping student musicians practice their music without a conductor present. It displays many of the factors a conductor would be performing, including the motion of the conductor's right hand/baton to assist with beat timings, cues for the musicians to play louder or softer, and indications of rest measures and entrance cues. Students would find the assistance and direction they need from the animation, although a computer program could never truly replace a human conductor.

Another potential application of this conducting animation project is to display a colorful visual to add excitement to an orchestra performance. Typically, orchestral performances provide auditory entertainment to the audience, but not much visual. According to *Orchestra Facts: 2006-2014*, a study of orchestra finances and operations commissioned by League of American Orchestras, despite orchestras decreasing their prices over the years, 60% of the orchestras involved in the League of American Orchestras reported a drop in

overall attendance from the years 2010 to 2014 [2]. Adding exciting visual effects to the orchestral performances could make them more appealing to audiences who would otherwise quickly lose interest in long shows. The system presented in this paper is a starting point for the creation of a spectacular orchestral performance.

1.1 Related Work

[3] describes a project with a similar goal: to convert digital sheet music into conducting. The developers of the project, though, chose to use MIDI rather than MusicXML due to the former being an older, more widely-used format. MIDI, however, does not include many of the essential conducting components in its data, and works only with specific software packages. Dorien Herreman's open-source project called "musicXMLparserDH" [4] is a Java MusicXML parser that creates an ArrayList of Note objects, which include information such as pitch, duration, and start time. While useful for other applications, note data is not necessary for the conductor. Instead, a conductor needs to know the details pertaining to the timing and dynamics (volume) of the piece.

2. SYSTEM OVERVIEW

Upon starting the program, the user is prompted with a window that allows the user to select a MusicXML file to use with the program (see Figure 1).

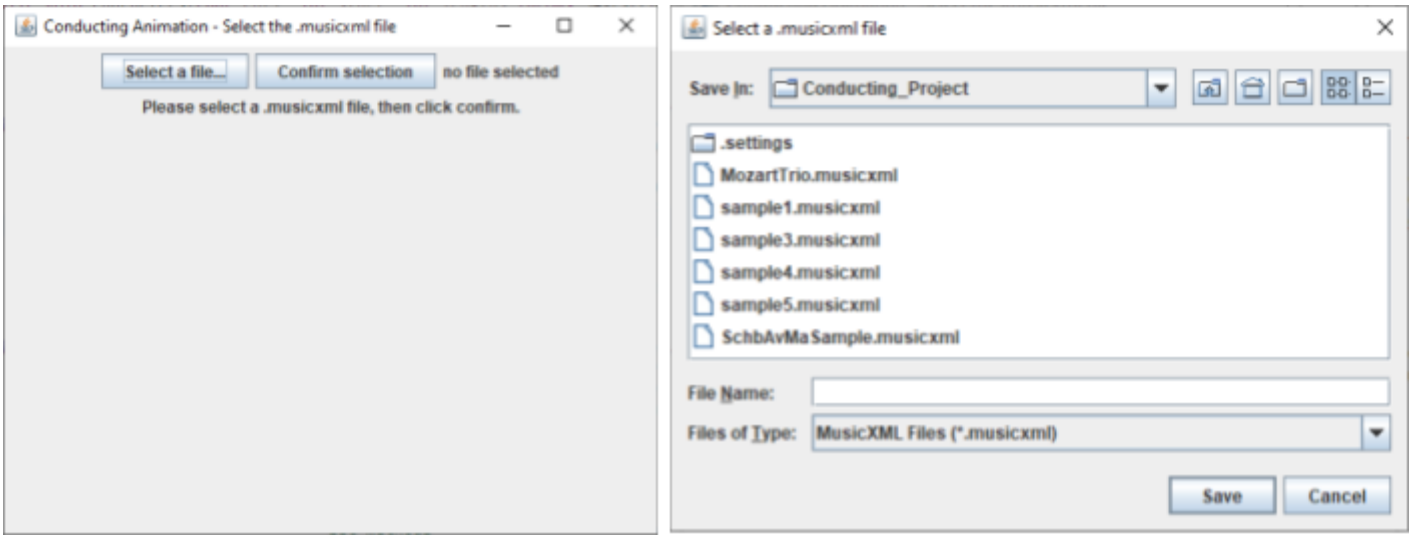

Figure 1. On the left: the window displayed to the user upon running the program. On the right: the file selection window allows the user to select a MusicXML file.

The user is then prompted to select a MIDI file to use for the program as well. This is used exclusively for the music playback part of the program.

After the MusicXML and MIDI files have been selected and confirmed, the program quickly parses the MusicXML file and presents a JFrame window labeled "Conducting Animation". The animation begins in the paused state: the user is able to click anywhere in the window to begin the animation, and can click again to pause it whenever they like. The paused state displays the piece's title and contributors at the top of the window, and a blue triangle

representing a “play” symbol in the center of the window. Once the user clicks to play, the program displays the following items (see Figure 2 for reference):

- the piece title and contributors, which remain at the top of the window throughout the animation;
- a row of colored blocks on the left, each representing an instrument part in the sheet music, labeled with the name of the instruments intended to play the part; these blocks fade out whenever the respective part encounters a rest measure, and fade back in when the part has non-rest notes to play;
- yellow arrows above or blue arrows below the blocks indicating an increase or a decrease in dynamics; these arrows include an appropriate label for the upcoming dynamics which fades out for a visual effect;
- a circle moving in a pattern according to the music’s time signature and tempo, displayed on the right side of the window, including a fading trail to emphasize the pattern;
- the time signature, displayed at the bottom of the screen;
- the current measure number and the total number of measures, below the time signature;
- the tempo in beats per minute, below the measure number.

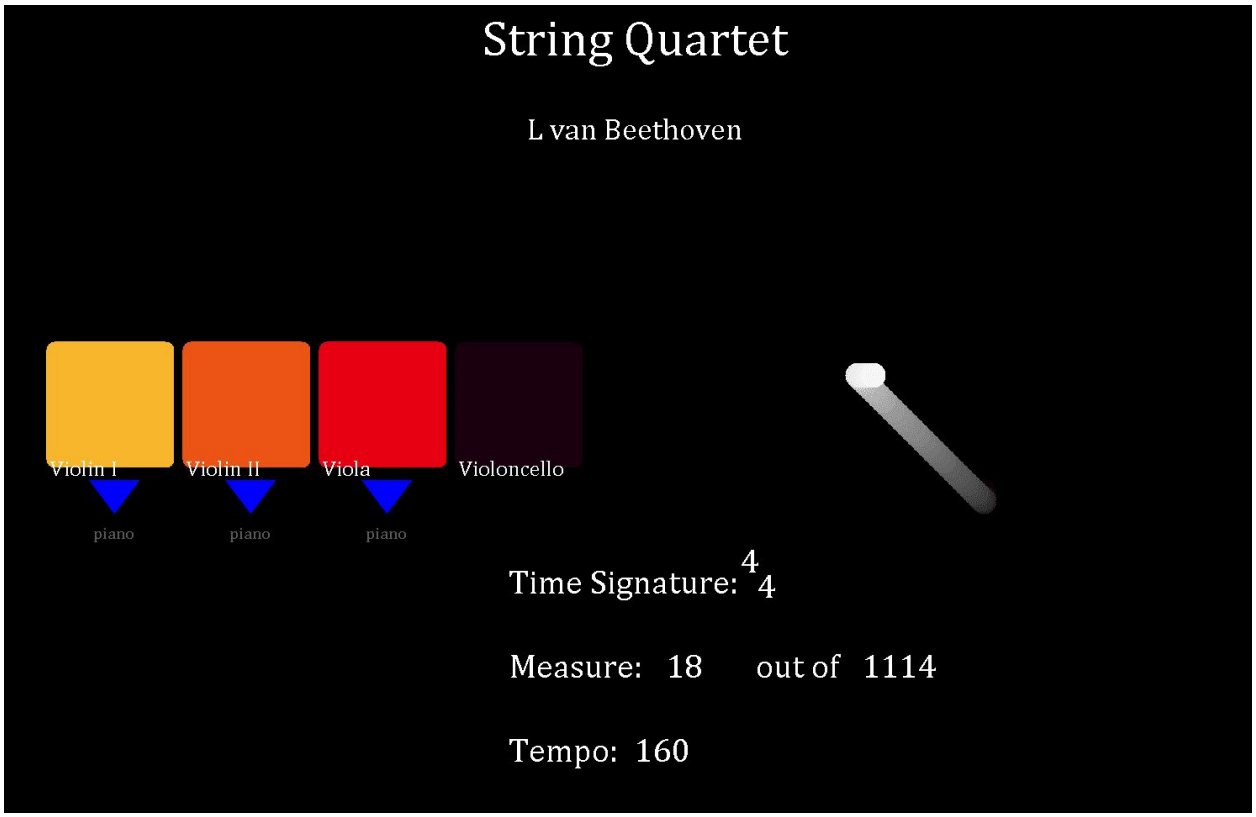

Figure 2. Screenshot of the Conducting Animation playing a String Quartet piece by L van Beethoven.

The MIDI audio is played alongside the animation, and is also paused when the animation is paused. Once the piece has finished, both the animation and the music stop. At any point in the animation, including after it has ended, the user can close the window and terminate the program by clicking the x in the top right of the window, just like any other window.

3. METHODOLOGY

MusicXML [5] is a type of XML (eXtensible Markup Language) format designed to be a digital representation of sheet music. Like other XML formats, MusicXML is a file containing readable text, formatted generally as such: *<tag>value</tag>*. The MusicXML file has specific tags for the different elements of the sheet music it represents. Overall, a MusicXML file has general music piece information listed at the beginning of the file, and then includes one music part after another (one for each instrument in the piece), with each part containing all of its measures in order.

3.1 Conducting Components

Some of the most important music components for conducting are the time signature, tempo, dynamics, and rest measures. A conductor needs to prepare the orchestra for the next musical moment as the previous one concludes [6].

The time signature affects which movement pattern a conductor needs to perform for the piece to keep in time with the music, as well as the length of a measure. It appears at the beginning of the sheet music as a number over another number; the top number defines the beats per measure, and the bottom number specifies the value of one beat. In MusicXML, the time signature appears in the first measure of each part, separately specifying the beats per measure and the beat value. For example, a 3/4 time signature (meaning that there are 3 quarter beats per measure) would appear in a MusicXML as *<beats>3</beats>* followed by *<beat-type>4<beat-type>*.

The music tempo affects how quickly the piece is performed, and how quickly the conductor needs to move their hand. A higher tempo means that the music is faster, while a lower tempo indicates slower music. This information is sometimes listed at the beginning of a music piece, shown as a specific note (e.g. quarter note) equaling a value representing the number of those notes per minute (see Figure 3). In MusicXML, this appears as one line within a measure, for example *<sound tempo="120"/>*. In this case, 120 is the number of quarter notes per minute. Notice that MusicXML considers it as quarter notes per minute regardless of the value of a beat, so there is a need to convert from quarter notes to beats if the time signature is not X/4.

Figure 3. Some components as found in sheet music. From left to right: time signature (3/4), tempo (120 quarter notes per minute), dynamics (*forte*), and a rest measure.

The dynamics involve how loud or soft the music is at any point. The conductor would typically make an “up” or “down” gesture with his or her left hand to signal a change in

dynamics to the orchestra. In sheet music, this is represented by a symbol such as an *f* (*forte*) or a *p* (*piano*), labeled below the section of the piece where the respective instrument should play louder or softer. Another way sheet music represents dynamics is with a crescendo or diminuendo wedge. The dynamics component does not have a specific value associated with it in the real world: it is up to the conductor or musicians how loudly or softly the music should be played, and is therefore ambiguous. In MusicXML, the dynamics can be represented in three ways: as an explicit value representing the MIDI note velocity, inscribed like this: *<sound dynamics="54"/>*; as a dynamics symbol written like this: *<dynamics default-x="110" default-y="-72" halign="left"><p/></dynamics>* (where the *p* represents *piano*); or as a wedge symbol: *<wedge type='crescendo' number='1' default-y='-63' relative-x='12'/>*, which describes the wedge as a crescendo or diminuendo and includes the positioning information of the element on the page of sheet music. The user manual for MusicXML explains that the dynamics representation is inconsistent in music scores, and whilst MusicXML captures what appears in the sheet music, it does not attempt to be optimal for analysis or synthesis of dynamics [7]. This makes it difficult to work with dynamics, since the component is very subjective to the interpreter of the sheet music.

It is also the conductor's job to signal entrance cues for after rest measures. This could be achieved by anything from a subtle glance towards the section, to a bold gesture to ensure they do not miss the moment to begin playing [6]. In sheet music, rest measures simply do not have any playable notes in them. MusicXML incorporates this into the format in one of two ways: by explicitly defining the measure as a rest measure as such: *<rest measure="yes"/>*, or by having only rest notes within a measure. Rest notes are presented like this: *<rest/>*, and are regarded similar to regular notes in that they include descriptors like duration and type (note value). Both ways to describe rest measures are easy to work with.

3.2 Algorithm

Our program was created in Java, using SAXParser [8] to parse the MusicXML file, several object classes to help organize the information, and ArrayLists to manage the objects. The animation is created in a JPanel managed by a JFrame. The JPanel implements Runnable in order to utilize a while loop to iterate at a rate of exactly 100 times per second, representing the frame rate of the animation. The components of the MusicXML such as time signature and tempo are considered together with the frame rate to ensure the animation is synchronous with the music.

The program is organized follows: the Main class creates and utilizes the MusicXMLHandler, which uses object classes PieceInfo, MusicPart, and Measure to organize the MusicXML data. The Main class then creates a MyFrame class, which creates and manages a MyPanel class. MyPanel generates and oversees an Orchestra and a RightHand. The Orchestra is comprised of Instrument objects, and uses those objects to create the display of the orchestra parts on the left side of the animation window. The RightHand requires an instance of a RightHandPattern, which is an abstract class extended by the specific pattern types. Figure 4 shows a simplified class diagram of the source code.

3.2.1 *MusicXML Parser*

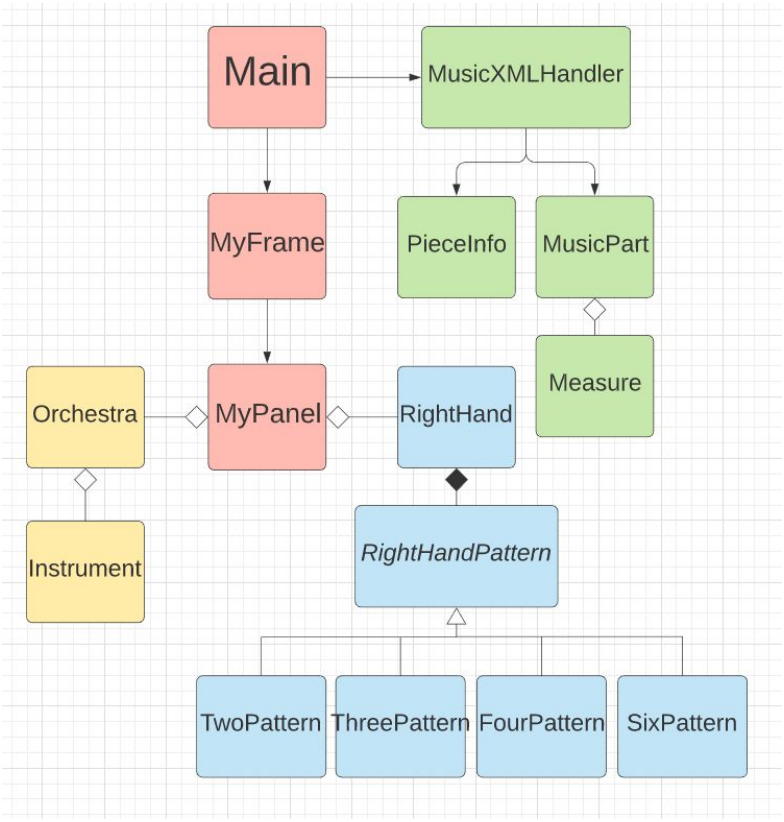

Figure 4. A simple class diagram of the program. The main classes running the program are shown in red, the classes to do with the parsing and organizing the MusicXML data are in green, the Orchestra and related classes are in yellow, and the RightHand along with its components are in blue.

The parser is defined in the MusicXMLHandler class, which extends DefaultHandler. It creates two data structures: PieceInfo, which is an object containing the information about the piece such as the work title, work number, movement title, movement number, and the creators; and PartList, which is an ArrayList of MusicPart objects. MusicPart encapsulates the ID of the music part (a string provided in the MusicXML), the instrument name, and an ArrayList of Measure objects. The Measure class includes information about the measure, including the measure number, tempo, dynamics, and a Boolean value for if it is a rest measure.

MusicXMLHandler overrides of DefaultHandler's methods startElement, endElement, and characters as recommended by Pankaj Kumar in his example on JournalDev of utilizing the Java SAX Parser [9]. When the parser is run on an XML file, it goes through the file from start to finish, checking for tags and elements. The startElement method looks for the starting tag of an element; if the string it finds matches the one provided in the code, the MusicXMLHandler either grabs the value found inside the tag (e.g. *<part id='P2'>* would obtain the string "P2"), or raises a flag in order for the endElement method to acquire the data between the two tags (e.g. *<beats>4</beats>* would obtain the string "4").

The MusicXMLHandler creates and defines an instance of PieceInfo class by checking for the element tags "work-number", "work-title", "creator", "movement-title", and "movement-number". Some MusicXML files may not include all of that information; PieceInfo decides which information to include from the information present, and sets the title to "Unknown Title" if the piece does not have a work title nor movement title. The MusicXMLHandler also constructs an ArrayList of MusicParts. Upon encountering the starting "part" tag, it creates an instance of MusicPart class. Then, upon encountering the starting "measure" tag, it creates an instance of Measure class. Between the starting and ending "measure" tags, this handler looks for keywords relating

to the measure attributes necessary for the conducting animation program. Table 1 below shows which tags pertain to which pieces of information within a measure.

Tag	Information
"number"	The measure number
"beats"	The number of beats per measure (top number of time signature)
"beat-type"	The value of one beat (bottom number of time signature)
"sound tempo"	The tempo
"sound dynamics"	The dynamics, written as an explicit value
"rest measure"	If attribute is "yes", then this is a rest measure
"implicit"	If "yes", then this measure is not a full measure and should not be counted

Table 1. The information obtained by the handler when it encounters the respective tag.

Implicit measures are used for handling pickup bars and measures with a repeat symbol not at the start or end of the measure. A pickup bar is an irregular measure at the very beginning of the piece, and it may not have the same number of beats in it as a regular measure [10]. This needs to be accounted for in order for the animation to be synchronous with the music.

After the MusicXMLHandler reaches the measure end tag, it adds the measure to the current MusicPart's ArrayList of measures. Similarly, once the handler reaches the "part" end tag, it adds the MusicPart to the part list. When the handler reaches the end of the MusicXML file, it makes some minor necessary adjustments to the completed part list, such as setting the tempos of measures that didn't list a tempo to the same tempo as the previous measure. At that point, the part list is ready to be utilized by the other classes.

3.2.2 Right Hand

The RightHand class oversees the movement of the white circle on the right side of the animation's window, which corresponds to the right hand of the conductor and indicates the tempo in a pattern determined by the time signature. When MyPanel creates an instance of RightHand, it supplies it with the frame rate, tempo, beats per bar, and the starting beat (this value is 0 unless the piece includes a pickup bar). RightHand utilizes a switch case, using the beats per bar value to determine which conducting pattern to use. The RightHand class defines two important methods: update and draw. RightHand's update method calls its pattern's update method to obtain the *x*- and *y*-positions of the circle and the measure number. The draw method uses the Java Graphics fillOval method to draw the circle in the location. The trail displayed by the animation is simply a Queue data structure that saves the previous 50 positions of the circle in order to iterate and display them with a fading color.

RightHand merely defines its update and draw methods, and does not call them itself; MyPanel is the class which runs the animation, so it is able to call RightHand's update and draw methods within its own methods.

3.2.2.1 Right Hand – Conducting Patterns

The RightHandPattern is an abstract class made to calculate and manage the position of the RightHand's circle. It is extended by four patterns: TwoPattern, ThreePattern, FourPattern, and SixPattern. Each child class represents a different beats per measure value, and displays a different animation based on the actual conducting motions (see Figure 5).

The only two pieces of information a pattern class needs are the number of frames per beat and the start beat (for the case of a pickup measure). Since RightHand decides which pattern to use for the music piece, each of the patterns have the beats per bar predefined. The RightHandPattern abstract class prepares an abstract method update for its child classes to override, and defines the method incBeatNum, which is to be called by each of the child classes within their update methods. The incBeatNum method not only increments the beat

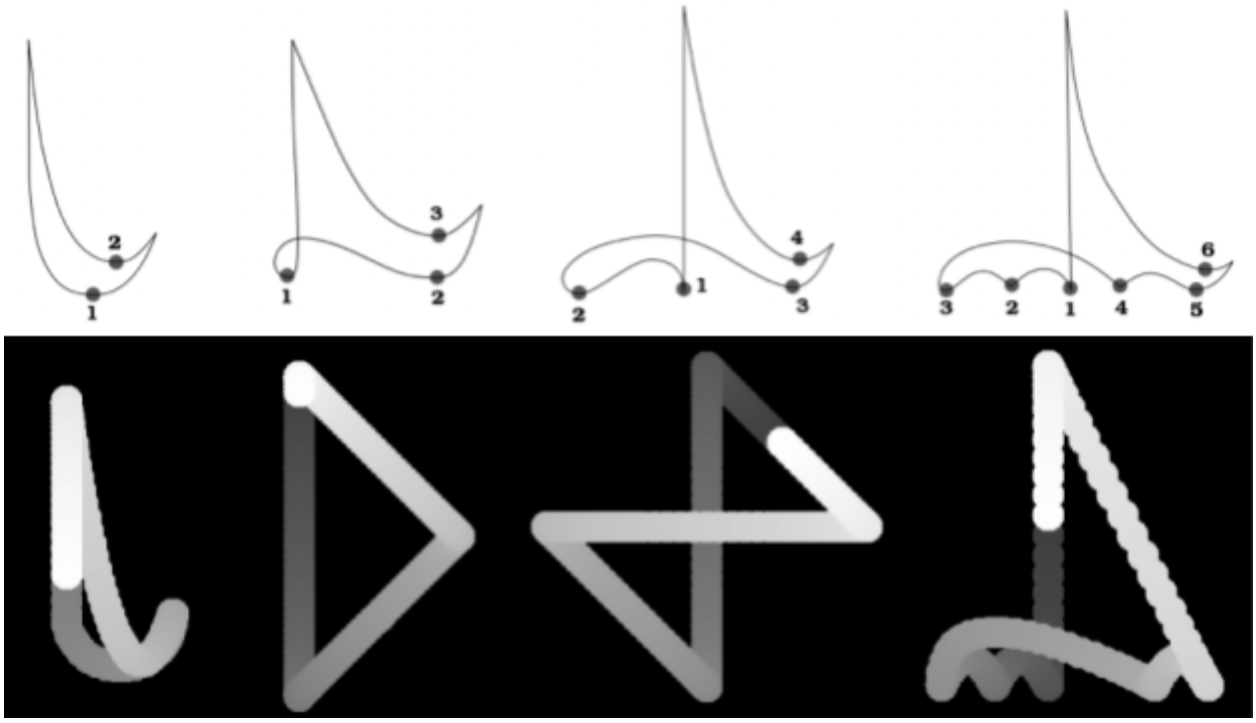

Figure 5. The conducting motions for four common time signatures with the animation paths shown below. From left to right: 2 beats per measure, 3 beats per measure, 4 beats, and 6 beats. [11]

number, but also accurately calculates the number of frames in the next beat. This is very important, since the frames per beat is calculated with the formula *(frames per second) * 60.0 / (beats per minute)*, and the result may not be an integer value. The program requires an integer number of frames per beat since it updates and draws frame by frame, and an unaccounted-for round-off error would result in a loss in frames over time.

Each of the child classes has its own update method to determine the position of the RightHand circle at any frame within any beat. The *x*- and *y*-values are calculated uniquely for each beat of a measure to provide the RightHand class with its circle's correct position on the program window. This is done with partwise mathematical functions, using a switch case for each beat in a measure. Firstly, a variable *t* is introduced to parametrize the

functions. *t* a value from 1 to -1 defined by the function *t = cos(pi * (current frame number) / (total frames per beat))* to create a smooth movement within a beat that speeds up at the start of the beat and slows down near the end. Then, the *x*-value is calculated as a linear function of *t*, giving the circle a smooth movement in the *x*-dimension. The *y*-value is then calculated as a function of *x* or *t*. Depending on the pattern and the beat number, the *y* function is anywhere from a linear function (to move in a straight line), to a quadratic function (to replicate a curve), to a function of the form $y = \sqrt{ax - b} + x + c$ (to imitate the asymmetrical movement from beat 3 to beat 4 in SixPattern). All of the functions are written ensuring that the appropriate movement is performed within the number of frames of a beat, regardless of what that number may be, and making sure that the position at the end of a beat is the position at the start of the next beat.

3.2.3 Orchestra

The Orchestra class handles the colored blocks on the left side of the animation, as well as the dynamics and rest measures aspects of the program, since both aspects are specific to each part of the orchestra. Firstly, Orchestra constructs arrays with data for the blocks based on the parts in the music piece, the data including the blocks' locations on the screen, colors, and instrument names. It also prepares arrays with the triangle locations to be used when the dynamics are increased or decreased for each instrument. The colors of the blocks are semi-randomized: upon creation, Orchestra randomly selects one of the 12 predefined colors to use for the first instrument part, and the consecutive parts match to the colors following the randomly selected ones (see Figure 6).

Orchestra requires the ArrayList of MusicPart objects in its constructor, and its update method requires the current measure number. Since RightHandPattern calculates the progression of the animation in sync with the music and is ultimately overlooked by MyPanel just like Orchestra is, MyPanel provides Orchestra with the current measure number based on RightHandPattern's information. Orchestra uses the ArrayList of MusicParts and the current measure to keep track of the current and upcoming dynamics and rest measures for each instrument part. Since a conductor would signal instruments in advance, the Orchestra class displays changes in dynamics and rest measure based on the upcoming measure. Orchestra has a draw method to do so, which renders first the blocks, then the names of the instruments, then the arrows (if there is a change in dynamics for the instrument). Orchestra also renders the time signature, tempo, and measure number displays. The rendering of the blocks takes into consideration whether the instrument will be encountering a rest measure in the upcoming measure, and if so, its alpha value [12] is decreased at a rate of -3 per frame, with a minimum cap of 30 out of 255. Likewise, if the instrument is currently in a rest

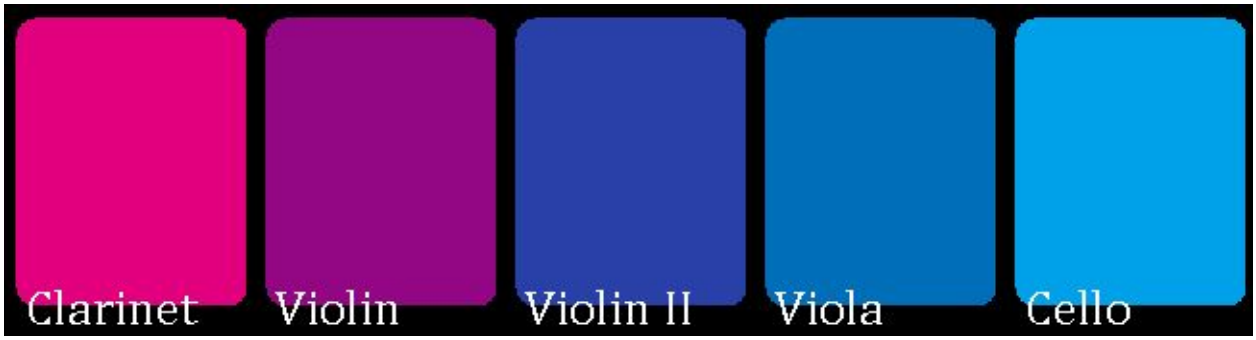

Figure 6. Each block represents an instrument part of the piece, and the colors are consecutive but begin with a random one each time the program is run.

measure and will begin playing again in the next measure, the alpha value is increased at a rate of +3 per frame to bring it back up to the maximum value of 255. The dynamics are handled similarly, using arrays to keep track of the current dynamics and upcoming dynamics. The Orchestra class draws the arrows based on the change in dynamics, which is equal to the upcoming dynamics minus the current dynamics for each instrument. Along with the arrows, Orchestra displays the full name of the dynamics symbol that matches the upcoming dynamics. Table 2 shows the values found in MuseScore [13] and the interval of values used in the conducting animation program.

Dynamics Symbol	Name	Velocity provided by Wikipedia	Interval used in the program
ppp	pianississimo	16	< 26
pp	pianissimo	33	26 - 41
p	piano	49	41 - 57
mp	mezzo-piano	64	57 - 72
mf	mezzo-forte	80	72 - 88
f	forte	96	88 - 104
ff	fortissimo	112	104 - 120
fff	fortississimo	127	> 120

Table 2. The symbols and names of the dynamics, as well as the values found in MuseScore and the intervals used in the program to determine which dynamics name to display with the arrows.

Similar to RightHand, the Orchestra class does not call its own update and draw methods; the MyPanel class calls those methods when appropriate.

3.2.4 MyPanel

MyPanel is a JPanel that implements Runnable. It predefines the frame rate using a delay value (set to 10 milliseconds for a frame rate of 100 fps), and the positionings of the separate elements on the panel. Upon construction, MyPanel uses the MIDI file provided by the user to generate a MIDI Sequencer [14] to play the audio. The MyPanel class instantiates a RightHand and an Orchestra using information from the ArrayList of MusicParts originally constructed by the parser. MyPanel includes the methods update, paintComponent (an override of a Java UI method), and run (an override required when a

class implements Runnable). The update method first calls the RightHand's update method, then uses RightHand's current measure number to call Orchestra's update method. MyPanel checks the tempo of the current measure, and if it has changed, notifies RightHand of the new tempo. Lastly, the update method checks if it is the end of the music piece: if the current measure is the last measure, MyPanel raises a flag to halt the animation.

The paintComponent method calls RightHand's and Orchestra's draw methods, and is automatically invoked when repaint is called. Repaint is called in the run method after the update method, so that the program draws based on the newest information. The run method uses a timed while loop decided by the frame rate to run the animation. It keeps track of the current system time in milliseconds, and uses the time difference between the previous frame's system time and the current frame's system time to calculate how long to wait until the next frame. Upon entering the loop, MyPanel starts the MIDI Sequencer to begin playing the music. Each iteration of the loop, MyPanel calls its update and repaint methods, and then calculates the time difference between the end of the previous iteration and the current system time. MyPanel then calculates the sleep time by subtracting that time difference from the predefined delay value, and uses *Thread.sleep* to wait for that time [15]. This way, each frame of the animation is allocated exactly the same amount of time (in this case, 10 milliseconds), so the animation runs smoothly.

MyPanel also includes a method to pause or unpause the animation. Pausing the animation raises a flag, which the run method uses to temporarily stop its frame loop as well as the audio.

3.2.5 Main/MyFrame

Main is the highest level class of the program, and handles the presentation of the windows. The file selection windows were made with JFileChooser, and Main incorporates threading to ensure the windows are presented in the correct order. After the files are selected, Main uses SAXParser together with the created MusicXMLHandler to parse the MusicXML file the user provided. The MusicXMLHandler creates an ArrayList of MusicParts, which Main then passes to the MyFrame upon creation.

MyFrame is a JFrame that implements Runnable and MouseListener, and creates an instance of MyPanel. Runnable allows for the window to appear on the user's screen and to close when the user closes it, and MouseListener allows for the frame to be able to react to mouse clicks within the window. Upon *mouseClicked*, MyFrame tells MyPanel to pause or unpause the animation by utilizing MyPanel's reversePaused method.

3.3 Testing and Results

The MusicXML website [5] has some sample files available that were hand-edited to include features not automatically generated by export software. There are some other sample files downloadable from Project Gutenberg [16]. The two sources make the pool of sample files more varied, with different features appearing in different files. The program was tested on a variety of sample MusicXML files to ensure that every aspect works for every variation. Some inconsistencies found between different files were the lack of title and/or contributors in some pieces, the various expressions for dynamics, the two different ways to describe a rest measure, the presence or absence of a pickup bar, and changes in

tempo throughout the piece. All of the sample files used for testing were in valid MusicXML format; the score-editing software MuseScore [17] was used to verify the validity of the sample files, as well as for exporting the corresponding MIDI files to use for the audio. MuseScore also displays the visual sheet music of a MusicXML file as it would look on paper, which was used to help test and debug the program.

The resulting program runs as expected. The animation and music are synchronous, the pause/unpause function works as expected, and the data retrieved by the parser is correct. The noticeable changes in dynamics in the audio correspond to the correct arrows display, and the instrument blocks fade out and in when appropriate. The movement of the right hand animation is similar to the conductor's motions for the appropriate time signature. The animation properly halts once the current measure number reaches the total measure number, and the music stops then, too.

4. CONCLUSION AND FUTURE WORK

MusicXML is a recently developed file format for digital sheet music, and had much potential for its usage. It contains useful information that other music file formats don't include, and parsing it is simple and does not require any special software packages. This project is an example of a way to use digital sheet music to synchronize live music with technology, and can be used as a tool to help student musicians or as a visual supplement to an orchestral performance. Some potential future additions to this project could be implementing a way to rewind the animation, adding a way to reselect files without needing to restart the program, and including more features from the sheet music such as repeats, codas, and individual notes.

5. REFERENCES

[1] Brian Wise. 2013. *Orchestras Use New Video Technology, Courting a Younger Crowd*. WQXR Blog. Retrieved May 3, 2020 from https://www.wqxr.org/story/311596-orchestras-video-technology-younger-crowd/

[2] Zannie Giraud Voss, Glenn B. Voss, Karen Yair, and Kristen Lega. 2016. *Orchestra Facts: 2006-2014*. League of American Orchestras. Retrieved May 2, 2020 from https://www.arts.gov/sites/default/files/Research-Art-Works-League.pdf

[3] Andrea Salgian, Lawrence Agina, and Teresa M. Nakra. 2014. *Teaching Robots to Conduct: Automatic Extraction of Conducting Information from Sheet Music*. International Computer Music Conference.

[4] Dorien Herremans. 2015. "An easy way to parse musicXML files in java". Retrieved September 17, 2019 from http://dorienherremans.com/content/easy-way-parse-musicxml-files-java

[5] MakeMusic, Inc. MusicXML. Retrieved May 12, 2020 from https://www.musicxml.com/

[6] Jerry Nowak and Henry Nowak. 2002. Conducting the Music, Not the Musicians. Carl Fischer.

[7] MakeMusic, Inc. 2015. *Element: Dynamics*. MusicXML User Manual. Retrieved May 12, 2020 from https://usermanuals.musicxml.com/MusicXML/Content/EL-MusicXML-dynamics.htm

[8] Java Oracle. “Class SAXParser”. Retrieved December 3, 2019 from https://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/SAXParser.html

[9] Pankaj Kumar. 2013. *Java SAX Parser Example*. JournalDev. Retrieved December 3, 2019 from https://www.journaldev.com/1198/java-sax-parser-example

[10] Robin McClellan. 2016. *Behind the Notation: Pickup Bars*. Noteflight Notes.

[11] Lumen Learning and Natalia Kuznetsova. “Conducting”. Lumen Music Appreciation. Retrieved May 14, 2020 from https://courses.lumenlearning.com/musicappreciation_with_theory/chapter/conducting/

[12] Java Oracle. “Class Color”. Retrieved March 2, 2020 from https://docs.oracle.com/javase/7/docs/api/java/awt/Color.html

[13] MuseScore. 2020. “Dynamics”. Retrieved May 19, 2020 from https://musescore.org/en/handbook/dynamics

[14] Java Oracle. “Using Sequencer Methods”. The Java™ Tutorials. Retrieved March 23, 2020 from https://docs.oracle.com/javase/tutorial/sound/MIDI-seq-methods.html

[15] Jan Bodnar. 2018. “Java 2D Games Tutorial - Animation”. ZetCode Tutorials. Retrieved October 2, 2020 from http://zetcode.com/tutorials/javagamestutorial/animation/

[16] Project Gutenberg. 2016. “The Sheet Music Project”. Retrieved March 1, 2020 from https://www.gutenberg.org/wiki/Gutenberg:The_Sheet_Music_Project

[17] MuseScore. Retrieved March 15, 2020 from https://musescore.org/en