DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ADVANCED JAVA PROGRAMMING

A MINI PROJECT REPORT

On

VIOLIN SYNTH

Submitted by

Pawan Bhandarkar
4NM16CS090

NITTE
EDUCATION TRUST

N.M.A.M. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution affiliated to Visvesvaraya Technological University, Belagavi)
Nitte – 574 110, Karnataka, India

NITTE
EDUCATION TRUST

N.M.A.M. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution affiliated to Visvesvaraya Technological University, Belagavi)
Nitte – 574 110, Karnataka, India
(ISO 9001:2015 Certified), Accredited with ‘A’ Grade by NAAC
☎: 08258 - 281039 – 281263, Fax: 08258 – 281265

Department of Computer Science and Engineering

B.E. CSE Program Accredited by NBA, New Delhi from 1-7-2018 to 30-6-2021

CERTIFICATE

Certified that the project work entitled “Violin Synthesizer” is a bonafide work carried out by P PAWAN BHANDARKAR 4NM16CS090 in partial fulfilment of the requirements for the award of Bachelor of Engineering Degree in Computer Science and Engineering prescribed by Visvesvaraya Technological University, Belgaum during the year 2017-2018.

Signature of Guide
Pawan Hegde
Assistant Professor
Dept of CSE
NMAMIT Nitte

Abstract

The main goal of this project is to provide casual users and musicians alike, a platform to experiment with different sounds. It presents a sandbox experience which provides a simple drag and drop interface that allows users to compose music.

This project is limited to only 12 basic frequencies played at 4 different time signatures each for a total of 48 different notes. From the 48 available notes, one can put together any number of notes in any order and by hitting the play button, the application will play those notes in sequence, effectively making music (?). Users have the freedom to experiment and try out new combinations of notes each time. Due to the lack of an audio set containing individual notes, each note has been recorded from scratch, resulting in a simplistic implementation of this "Violin Synthesizer"

Name of the note	No. of beats
Semi-quavers	1/2
Crotchet	1
Minim	2
Semibreve	4

Contents

Introduction

The inspiration for this project arose primarily from my love for violin music and my distinct lack of skill and experience in that regard. As I stared at my book of music notes, I thought to myself “If only there was a way to make reading music more fun!” and so I made one. A friendly warning to all those who would wish to put this project to the test, each sound you hear has been recorded using a cell phone and edited using an off the shelf sound editor I found via google search. So if the music sounds unpleasant and squeaky, that’s simply because I’m a terrible violinist.

Implementation

The Violin Synth was made from scratch using the following software:

1) IntelliJ IDEA – For all the Java codes and the GUI
2) Audacity – An audio editing software used to create the sound files
3) MS Paint – For making all the image resources required for the UI

The audio resources are all stored in the “Notes Audio” folder of the project and comprises of 48 different .wav files while all the image resources are stored in the “Notes Images” folder of the project and comprises of 107 .png files that represent the notes and the staff in various ways across the application GUI.

The JAVA code can be divided into 5 main sections as mentioned below:

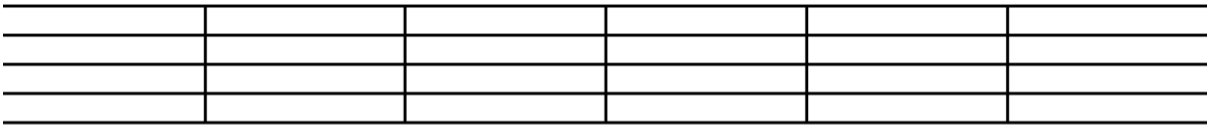
1) GUI
2) Helper methods
3) Sub-classes
4) Audio player
5) Timer

The Graphical User Interface

All of the GUI of this application exists in a single class `MusicFrame` which extends the JFrame class found in the package `javax.swing`.

The code for initializing and displaying the GUI is hence present in the constructor of the MusicFrame class. The three main parts of the UI are:

1) **<u>The staff</u>:** This is a JPanel present within a JScrollPane. In this Panel, each staff row is represented using a JLabel which behaves as a container and holds each note to be played as it is being dragged and dropped onto it. This staff without any notes looks like this:



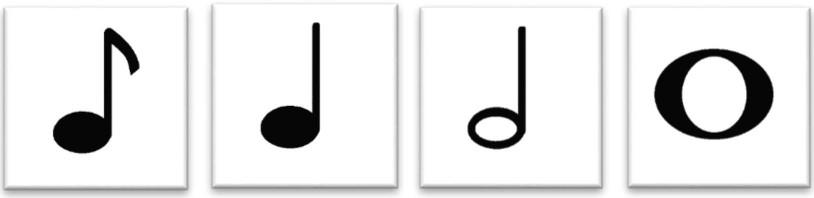
2) **<u>The buttons</u>:** There are two rows of buttons present right below the staff.

The first row consists of the following buttons:

Play **Pause** **Stop** **Add**

The **Play**, **Pause** and **Stop** buttons are used to control the music (notes) as it plays while the **Add** button is used to add a new staff row to the staff panel i.e. make space for more notes to be added. (button icons are stored as .png files)

The second row of buttons looks like this:

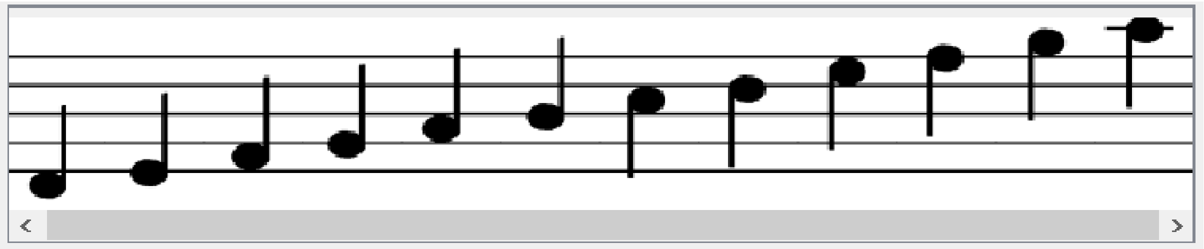


Semiquaver **Crotchet** **Minim** **Semibreve**

Clicking on any of the above will load the corresponding notes into the panel below it

3) **<u>The Notes</u>:** These are effectively the "sources" from where the notes will be dragged and placed onto the staff and will be loaded dynamically as and when one of the above buttons is clicked.

When loaded, the selected Panel will look like this:



So to sum up the user interface, you first click one of the note icons to load the notes below it, drag the required notes onto the staff and click play to hear the selected notes in the order in which they were dropped.

Helper Functions

These are the user-defined functions that take care of most of the heavy lifting in the application. The most important ones are:

1) `private void loadMap()`

   This is the function which is first during the execution of the program. It loads all the images into memory and makes them available for the application to use during execution. A combination for-loops, HashMaps and a Set is used to load all the data rather than a separate statement

2) `private void addStaff()`

   This function is responsible for adding a new row of staff to the staffPanel and updating the relevant ArrayList to reflect the added staff row.

3) `private NoteIcon getImage(String imageName, int width)`

   This function takes a String name which represents the name of the note required and an integer width which represents how wide it must appear on the staff. This is important because the width of a note affects how many notes can appear on a fully loaded staff without getting pushed off the screen.

4) `private void checkWidth()`

This function checks to see if the collective width of all the notes present in a particular staff row exceeds the total width of the staff itself. If it does, then it calls the **`compress()`** function.

5) `private void compress()`

This function is called when some notes are not visible (they have been pushed off the screen) and it "compresses" the staff, removing empty spaces and makes all the notes fit nicely onto the screen.

Sub-classes

1) myHandler

```
   class myHandler extends TransferHandler{

/**This class takes care of the heavy-lifting when it comes
to the actual drag and drop mechanism that governs the UI of
the note*/

   myHandler(String property){

/**This is the class constructor which in our code, simply
calls the constructor of the TransferHandler class.*/
 }

   @Override
   public void exportAsDrag(JComponent comp, InputEvent e,
  int action) {
/**This is an overridden method in which we can specify the
actions they take when the dragging is initiated. In our
code, we store a copy of the currently in-transit components
for later use. */ }
```

```
   @Override
   public boolean canImport(TransferSupport support) {

/** This function is used to allow or disallow dropping of
a component that is being dragged. We can check the
contents of the component currently being transferred and
decided whether or not the current drop location is
suitable to be dropped on.

In our code, we check to make sure that the drop location
is in fact, the staff and that it doesn’t already contain
too many notes. This function returns true if the target
allows the component to be dropped, false otherwise.*/
}

    @Override
    public boolean importData(TransferSupport support){
/**Here, we specify what actions to take as the item is
being dropped. In our code, we extract the dropped
component and add the note present in that component to the
overall ArrayList<> of components which contains the notes
to be played. This function returns true if the import was
successful, false otherwise. */
}

    @Override
    protected void exportDone(JComponent source,
Transferable data, int action) {
/**This is where we can specify what actions are to be
taken one the drag-and-drop transfer has completed
successfully. In our code, we call checkwidth() function to
ensure that the last drop didn’t cause any notes to be
pushed out of view.*/
    }
```

```
}
```

2) noDragMouseListener

```
class noDragMouseListener implements MouseListener{

/**This is class which serves as the MouseListener for
the notes in the staff. The main purpose of this class is
to provide the "highlight" whenever the user moves the
mouse over the notes.

Since it is implementing the MouseListener interface, all
of the following methods have to be overridden:*/

  @Override
  public void mouseClicked(MouseEvent e) {

/**We can specify what action is to be taken when the
source is clicked. Note that the "click" refers to the
action of pressing and releasing the mouse. This is in
contrast with the next two overridden methods. In our
program, we do nothing here.*/

        }

   @Override
   public void mousePressed(MouseEvent e) {

/**Here, we specify the action to be taken when the mouse
is "pressed". This does not require the mouse to be
released once it is pressed. In our code, we start a
timer to record how long the mouse is kept pressed*/
}
```

```
@Override
public void mouseReleased(MouseEvent e) {

   /**This is where we put the code that executes when the
   user releases the previously pressed mouse button. In our
   code, we first check to see how much time has elapsed
   since the user pressed the mouse. I.e. how long has the
   user been holding the mouse button?

   If the duration is more than half a second, we play the
   (source) note that is being pressed. This provides a
   useful way to check what a note sounds like on the staff
   without having to wait for the player to reach that note
   from the start.

   If the duration is less than half a second, then we
   interpret that action to be an ordinary "click" and we
   simply remove that note from the staff. (This is how note
   deletions are handled in our code. */

    }
    @Override
    public void mouseEntered(MouseEvent e) {

    /** This function gets executed every time the user
   rolls his mouse over the source to which this listener is
   attached. In other words, when the user "mouseovers" a
   component attached with a MouseListener, this code gets
   executed. In our code, as soon as the use mouseovers a
   note, we highlight it.
Note: the “highlighting” here is simply being done by
replacing the image of that note with an alternate
(green) version of that note, hence making it appear
highlighted. This helps give a visual cue about which
note we are about to press.*/

    }

    @Override
    public void mouseExited(MouseEvent e) {

     /**Here, we specify what is to be done when the user
moves his mouse out of the note. (i.e., mouseover exit)
In our code, we simply replace the previously highlighted
version of the note with the regular version.*/
    }
}
```

3) NoteLabel

NoteLabel is simple a regular JLabel with an added feature that it stores the icon (in our case, the note that it represents) This is required to easily access the notes in other parts of the code.

```
public class NoteLabel extends JLabel {
    NoteIcon icon;
    public void setIcon(NoteIcon icon) {
        super.setIcon(icon);
        this.icon = icon;
    }
    public NoteIcon getIcon() {
        return icon;
    }
```

4) NoteIcon

NoteIcon, similar to NoteLabel is simply a regular ImageIcon with the added capability of storing the NAME of the note as a String as well as the time signature of that note in a Double variable.

```
class NoteIcon extends ImageIcon {
    private String noteName;
    private double time;
    NoteIcon(Image path, String noteName, double time ){
        super(path);
        this.noteName = noteName;
        this.time = time;
    }
    String getNoteName()
   return noteName;
    }
    double getTime() {
```

```
                return time;
            }
        }
```

Audio Player

The audio player class is the one which takes care of actually loading the required notes and playing them one by one. It does this by using a Clip object from the javax.sound.sampled package. It has play() pause() and stop() methods which make audio manipulation simple and easy in other parts of the code. Before the AudioPlayer can be used, it has to first be initialized by passing the string path of the audio file via the constructor.

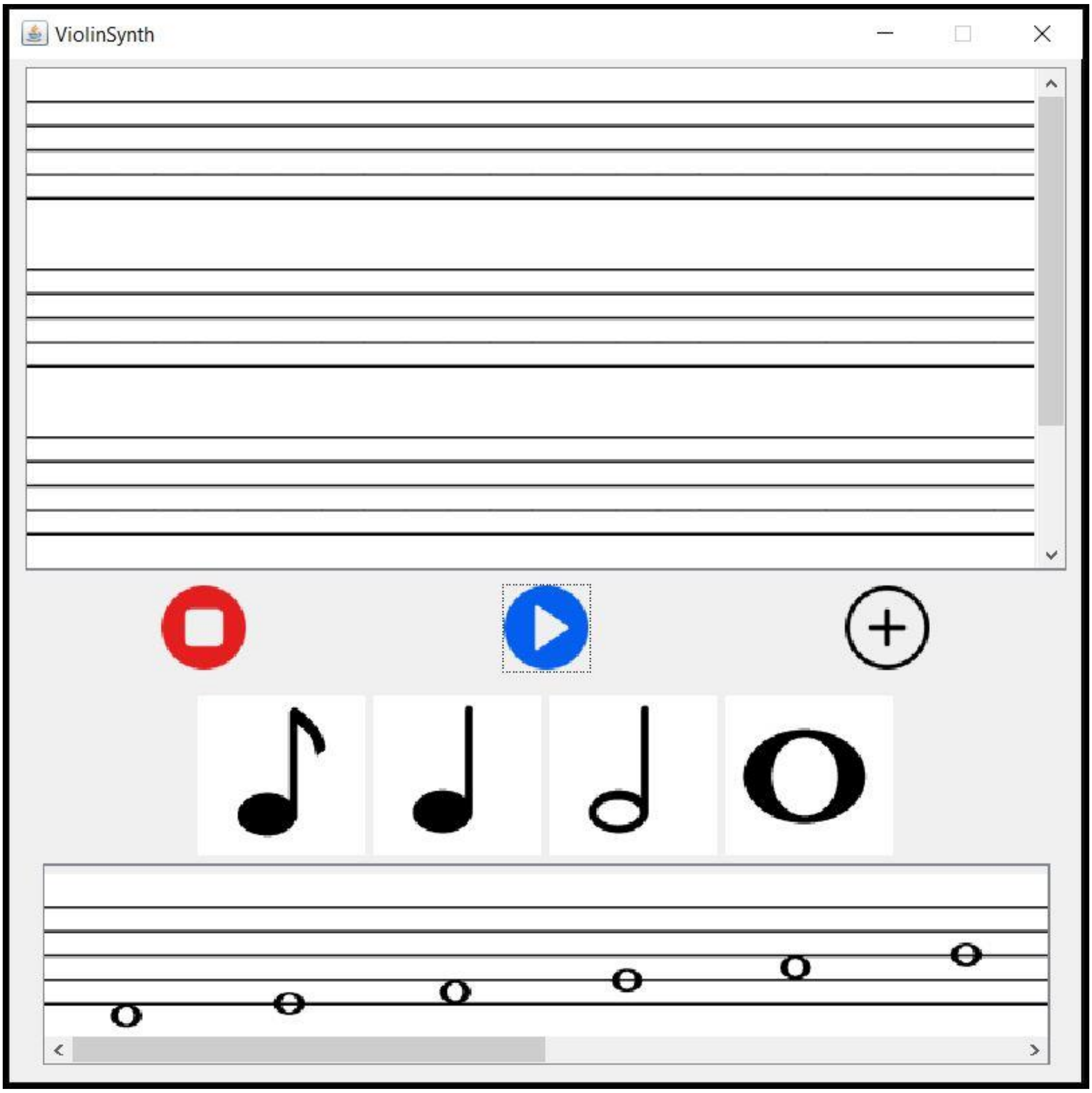
Timer

One of the important concepts used in the program is the use of a “Timer” object present in the java swing Library. A Timer is basically used to perform a certain action repeatedly, once every <specified time interval> seconds. The syntax for using a timer is as follows:

```
Timer t = new Timer(delay, new ActionListener() {

@Override
public void actionPerformed(ActionEvent e) {
      /**This is where we specify what has to be done at every
‘delay’ intervals of time. In our code, we handle the playing
and pausing of the notes on the staff by means of the timer.
This is made possible because all the notes to be played are
stored in a global ArrayList. This is important because if the
ArrayList was declared/initialized inside the timer, it would
be initialized to the same value each time since a Timer
simply REPEATS and action and doesn’t ITERATE. So we increment
a pointer to the ArrayList of notes to fetch the next note for
every interval. */
});
t.start(); //Starts the timer
```

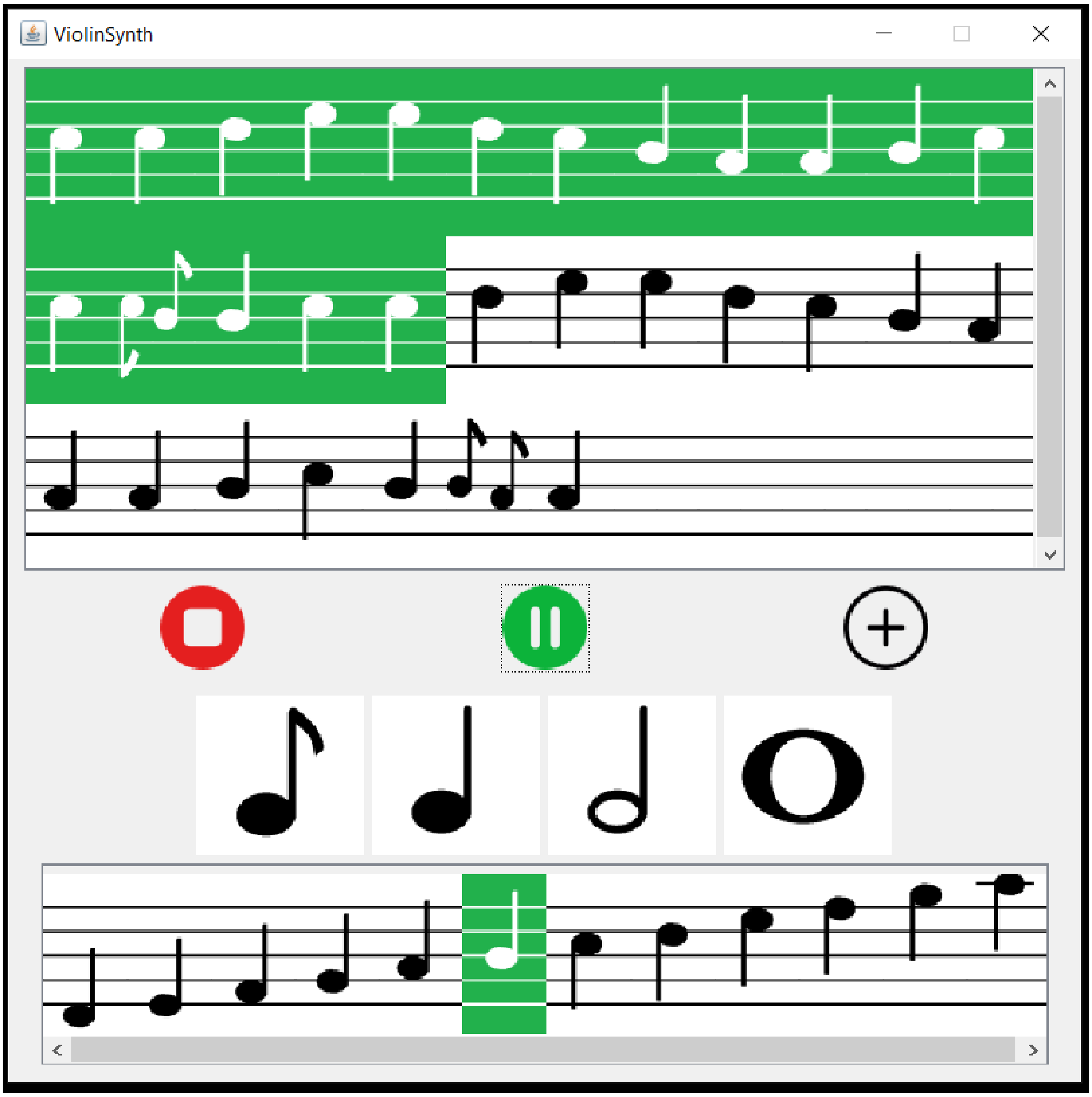
Screenshots:



This is what the main Application looks like when the program is run.

From top to bottom:

1) Staff Panel
2) Button Row 1
3) Button Row 2
4) Selected Notes



The first verse of “An Ode to Joy” by Beethoven, being played on the app.

Things to notice:

1) The music notes get highlighted as they are being played
2) The Play button is being replaced by a pause button
3) The Bottom panel has been loaded with semiquavers.
4) The note in the bottom panel is being highlighted due to mouseover

Conclusion

Although the all the code works and the application itself runs reasonably well, it must be said that there is scope for significant improvements in terms of design, code and audio quality. Although it started off as a “Violin Synthesizer”, the code is structured in a way that makes it possible to switch from Violin to any other instrument by simply changing the value of a String. (In particular, the string that holds the path of the audio files). This is possible ONLY if we are able to obtain the audio files of the corresponding notes played on the instrument we wish to switch to. From the conception of the idea to the writing of this report, this project has been an absolute pleasure to work on and I come out saying that I’ve learned quite a bit!

References:

1) DnD in Java: https://docs.oracle.com/javase/tutorial/uiswing/dnd/intro.html
2) Audio Player: https://www.geeksforgeeks.org/play-audio-file-using-java/
3) Java Layouts: https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
4) UI Design with IntelliJ: https://www.youtube.com/watch?v=G1Zo3UKzB4A
5) https://stackoverflow.com/questions/1513178/jlabel-on-top-of-another-jlabel