Audio Visualizer

By: Keith Kutsuma & Kelsey Valencia

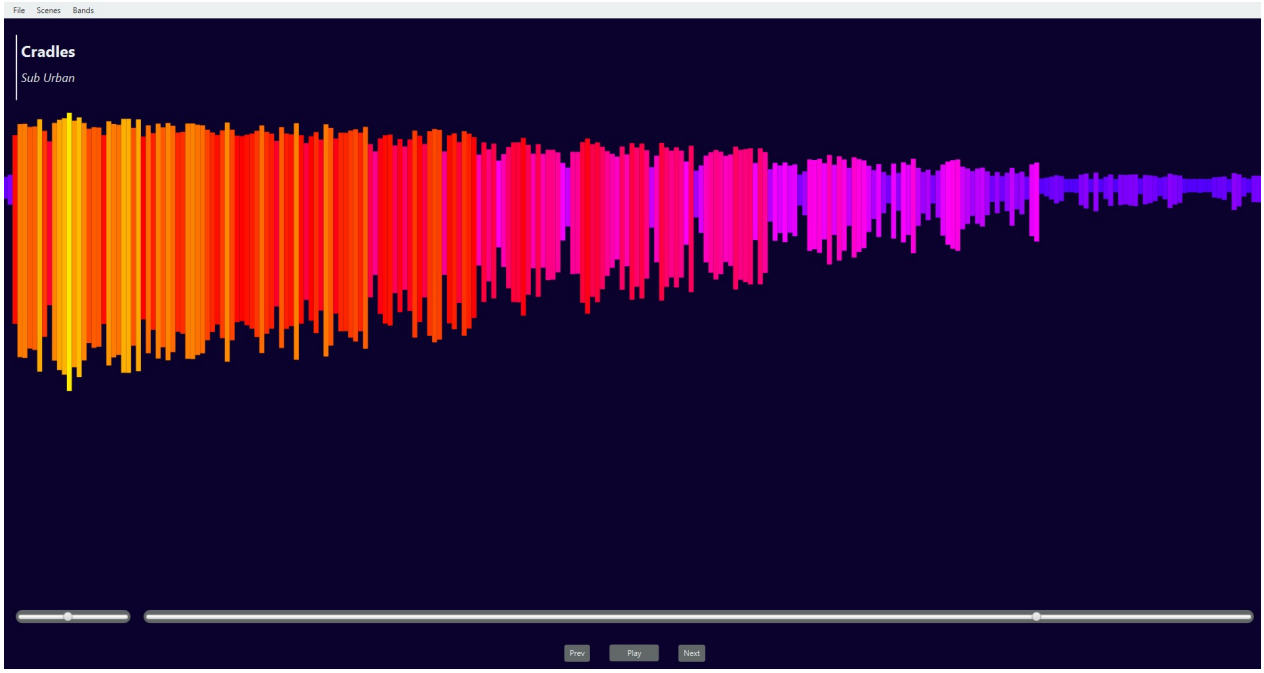
Created for CU Boulder CSCI 4448 SU21 Final Project

Demonstration

Demonstration Video

System Statement

Our Audio Visualizer is now capable of reading .mp4, .mkv, .mp3, .wav files, and more. Any valid file given will be played back with the addition of an representation of the audio being played back.



The Audio Visualizer is also capable of multiple color settings, different amounts of bands, and adding valid media files. This is all on top of the normal capabilities of a media player such as pause, resume, skip, and previous media selections.

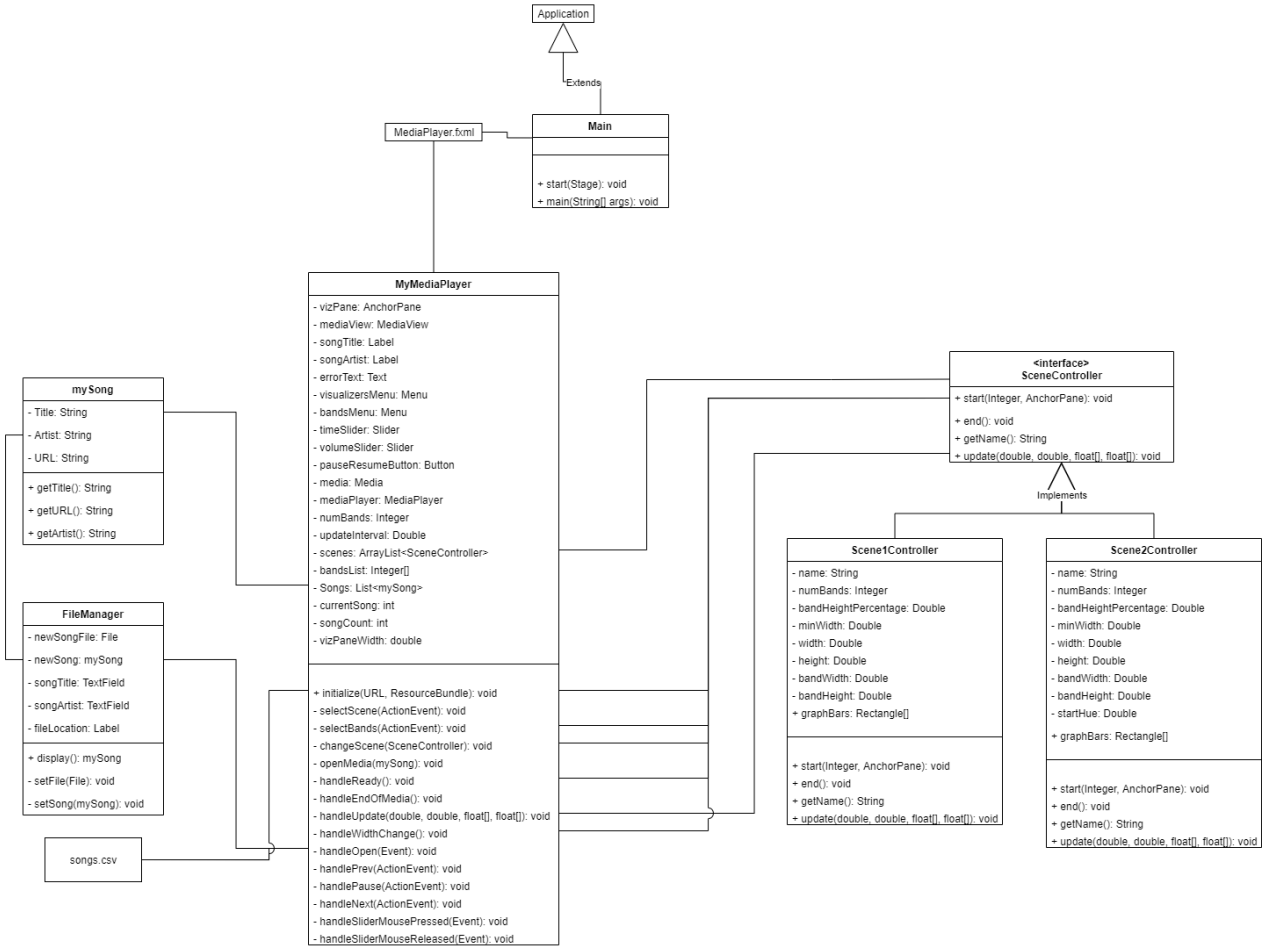
Some of the changes made from project 5 that did not make it into the final project are saving songs to one folder, deleting songs, and choosing which songs to play. This is because we realized it would take a lot more formatting and UI design to properly list and include the delete functions that we would want so it wasn't included for this sprint.
The largest reason to this is because we changed our requirements towards copying songs and we will get to that in the next section.

From project 6 we made major changes. Starting with the file system we added the capability to play more than one hard coded song, have the songs run continuously, skip/return to another song, and adding new songs. Also we added two types of visualizers and the option to change the amount of spectrum bars you can observe. We also worked on the UI and redesigning the program to look better. This took more time than expected since we changed our file structure and how we set-up the classes due to this.

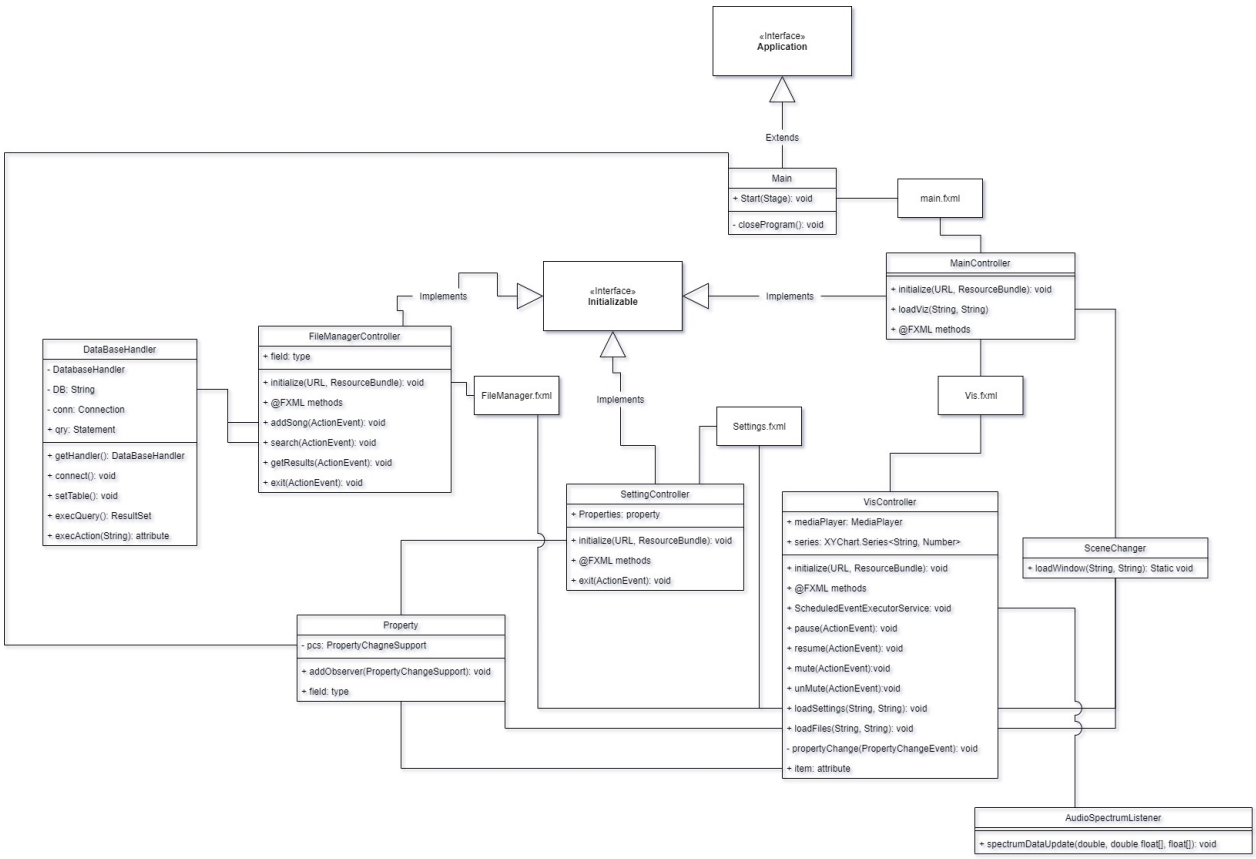
Comparison Statement

Key Points & Changes

Here are the Project 5 and the Final Class Diagrams.



Final Class Diagram



Project 5 Class Diagram

We can see that the final class diagram looks like a trimmed down version of the project 5 class diagram. And that is because it is. Many of the classes such as the SceneChanger, DataBaseHandler, SettingsController, Property, and the extra .fxml files weren't needed. In fact most of the .fxml files were excluded from our Project 6 because it didn't give us enough flexibility to handle the visualizations that we wanted.

As such Kelsey took Project 6 and slimmed it down to the version now. Now there is one MediaPlayer.fxml that handles the visualizations of the main page. This includes the placement of the bars, visualizer, buttons, etc. In addition to handling the styling of this page.

Then there is the MyMediaPlayer which is the controller for MediaPlayer.fxml. This handles all of the information needed to play the MediaPlayer. It sets the listeners and delegates the work to the SceneController how to update the visualizer. In addition the MyMediaPlayer handles any of the menu buttons which will change the visualizers. Finally this class will delegate work to the FileManager to get a new song from the user and add it to the current list of songs.

Otherwise the Implementations of the SceneController create the visualizations that we want to have. In our cases they're rectangles to represent the bars. start(Integer, AnchorPane) will set the bars position/values. Then update(double, double, float[], float[]) will change the bar heights depending on the values that the media player's observer is giving it.

Going back to the FileManager we felt that it wasn't needed to have a sql database to handle the information for this project. It was overboard, and we didn't want to make instillations for the team and others to be more difficult. So we went with handling the song information in a csv, and using java's io writer/reader to work with the file. It didn't require any more libraries and was a simple implementation.

Design Patterns

Since Project 6 we took out the command pattern since it wasn't needed, *(more isn't always better)* and added a strategy pattern. This is so that the MyMediaPlayer can simply run update on the current scene and it will change the values of the bars, *(or what we are using as to represent a band)*.

We thought that this was more important because while it makes the scene itself more difficult to change, it makes the visualizer much easier to customize and design. This was more of the headstone and main goal we wanted which is why we took this approach.

Otherwise we are also using the built in observer pattern to update the visualizer to any changes in the music. Also both the volume bar and timing bar use observers to update their values or the songs value.

Third-Party Code vs Original Code

Firstly the front end API of JavaFX and any java libraries were not coded by us. Although we did use them heavily in this project.

In addition the .fxml file, while it was made by us we did use SceneBuilder which is a UI based tool to generate the code in order to create the scene visuals we want.

The file system used Java's Read/Write libraries to read, write, and find the .csv file. We didn't use an external library for the csv and used really simple read/write statements. Resources we used for this can be found here.

The controls of the project such as the volume slider, timer bar, and buttons use basic functions of the JavaFX API and we use multiple techniques. For both slider's we use different ways to control them because they each have slightly different needs. The volume bar used a way that Oracle handled a slider, but didn't handle the timer bar how we wanted so Kelsey found another way to do that. Most of the standard practices can be found on Oracle's website and this was a resource that helped.

For the visual graphic design portion of our project, https://docs.oracle.com/javase/8/javafx/api/javafx/scene/shape/Rectangle.html was a great resource used to help understand the properties and methods that make up the Rectangle shape Javafx class. Additionally, https://www.tutorialspoint.com/javafx/javafx_colors.htm was also a useful resource to look at when trying to implement different color patterns such as, the linear gradient pattern used in the scene, "Rainbow".

Final Statements on OOAD

1. Keith found it difficult during the designing phase especially around JavaFX since we didn't know much about it. It was difficult to determine how the code needed to be handled and what was best practice. Since working with more packages/libraries, they can be designed in a way that isn't Object Oriented Friendly. Or they have an aspects that have difficult give or takes. In our case learning how to deal with .fxml files was difficult and took many prototypes until a decent design was even made.

2. Another point Keith saw was that our group worked in more of a scrum style. In the end when we started to understand more about JavaFX and working together our results were quicker and seemed to work better. Given more scrums the project was on track to have some impresive results. The issue being that we don't have continuous amounts of time to work and should have treated it more like a waterfall style since we have a hard deadline for the class.

3. Kelsey encountered numerous problems when trying to implement the file chooser feature included in our final draft of our project. The main problem being the excess use of multiple .fxml files and loading all of them using the FXML loader in our previous draft of our project. Kelsey found a way to work around this issue by restructering the overall program layout and removing all extra .fxml files except for our main one.

Project Set Up

To work on this project we are using JavaFX 16 & Java 16. You can find the latest JavaFX download here. To set up JavaFX you can find instructions here. You can then run the main function if you want to run the program there.

There is also a built .jar file **AudioViz.jar** which can be executed with the vm options

```
--module-path ${PATH_TO_FX} --add-modules javafx.controls,javafx.fxml,javafx.media
```

Where the ${PATH_TO_FX} is your own path to the JavaFX lib folder.