ECE 485 Final Project
‘TuneLAB’

Davis Gossage
Emmanuel Shiferaw

Idea behind project

The idea for this project was to extend Lab 3: Digital Synthesizer by adding a user-friend GUI and allow for more customizability of sound. We wanted to extend the idea from Lab 3 of playing notes for certain durations to include more keys, finer control over note duration, the ability to combine tracks, different instruments, and the ability to add audio effects.

Materials studied to start project

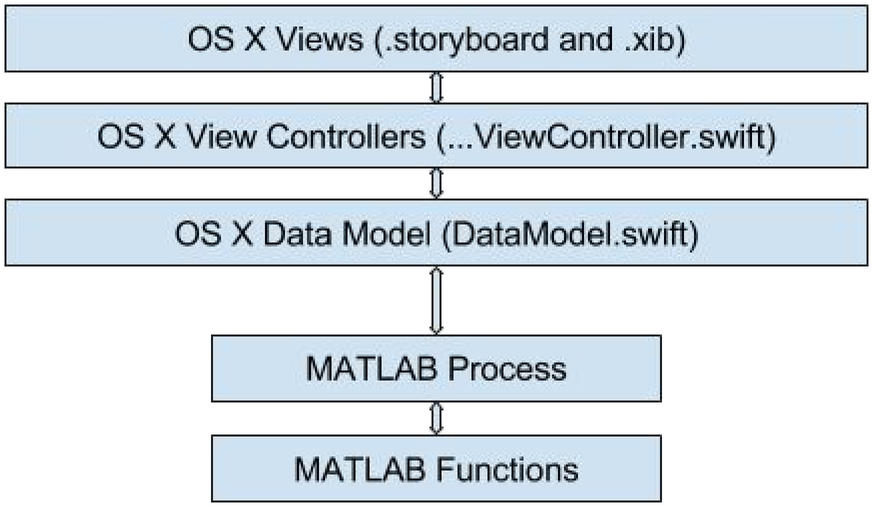
We began with Lab 3's approach, generating/computing signals for songs using DTFS coefficients, only given the desired frequencies/notes as input. Our initial idea was to simply connect our existing code (or a more cleanly-abstracted version of it) from Lab 3 to a user interface built in MATLAB, using the GUI programming API built-in to MATLAB. Dials/buttons/sliders provided by this interface would allow us to set parameters and build a song. We found, however, that the MATLAB UI toolkit is designed primarily for simpler projects like computational exercises that have an integrated visualization component - numbers are entered into text fields, buttons are pressed, and graphs are generated. We decided, instead, to build the UI outside of MATLAB.

Regarding the MATLAB/Audio processing side of our project, we wanted to add options for audio effects and for emulating different instruments. Luckily, we had the handout provided in class that gave the code for many functions.

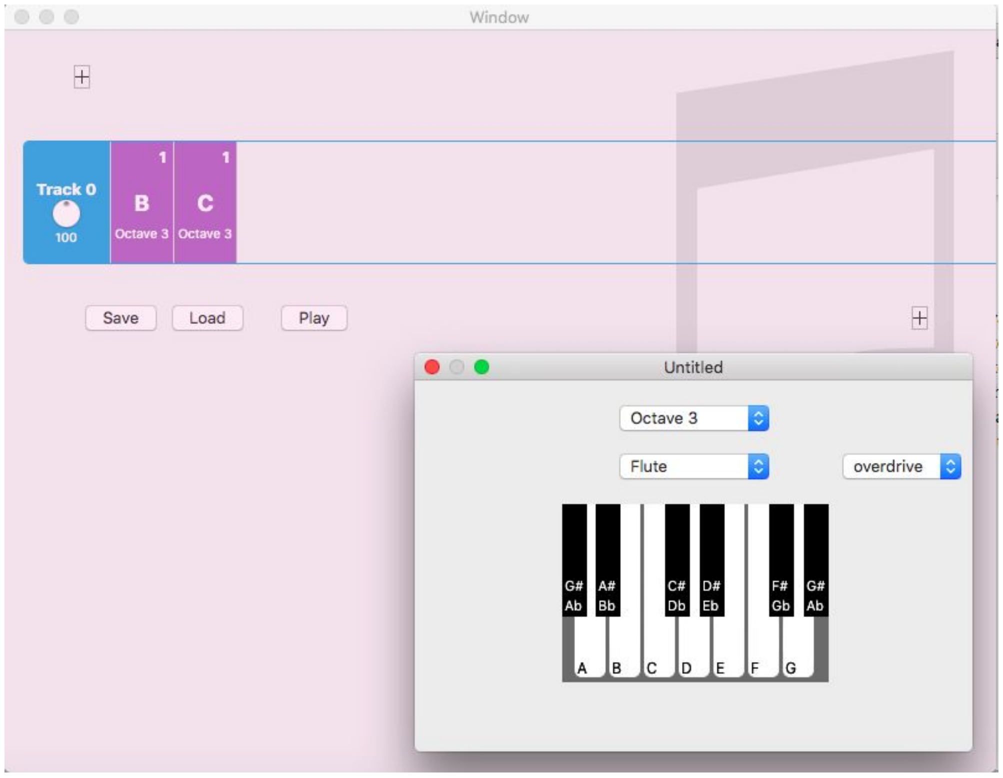
Materials created to accomplish project

TuneLAB is a Mac OS X application powered by MATLAB. We chose to write an OS X application because we did not believe MATLAB's UI components would be robust or flexible enough to implement the interface we had in mind. Both of us have experience in Swift, so we chose to use Xcode and Apple's AppKit framework to build TuneLAB. On startup the app launches the MATLAB executable as a subprocess in headless mode (without showing the MATLAB GUI). The app then maintains an input and output pipe with the MATLAB process, sending and receiving information as the user interacts with the app.

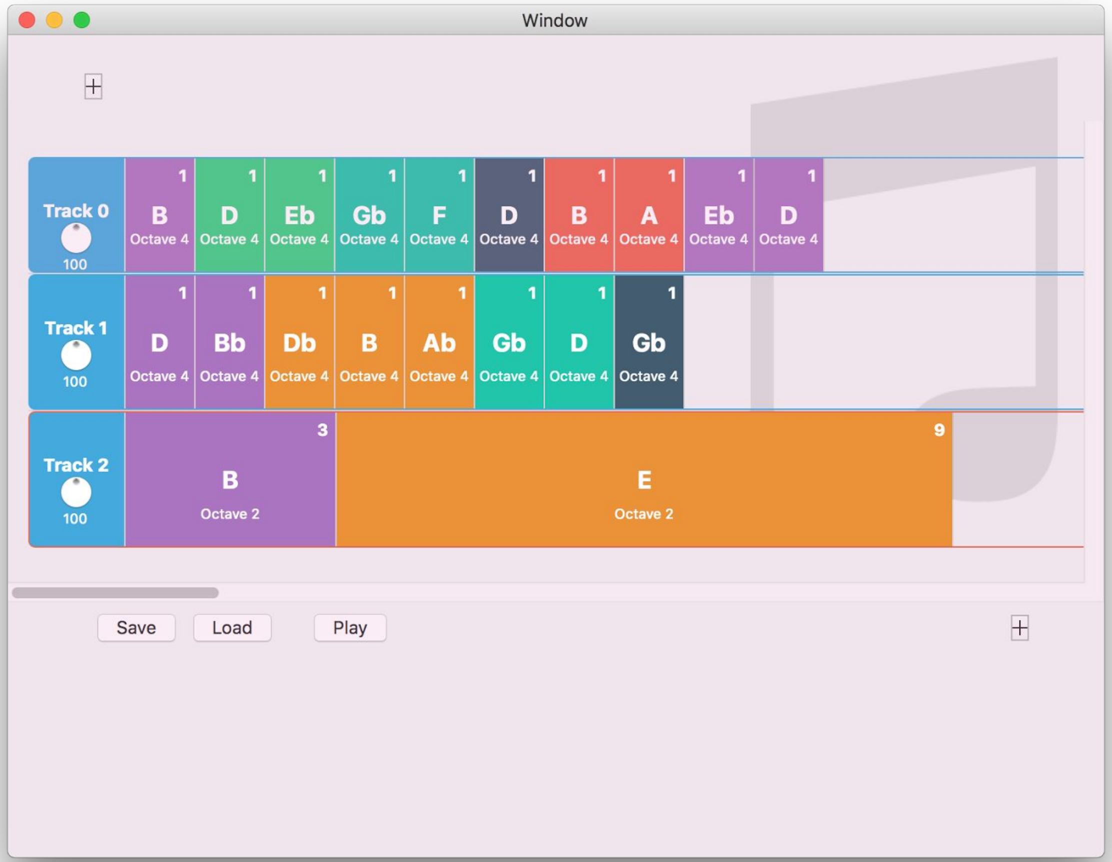
The end user sees the top level of the app hierarchy, which are the OS X Views. These views have controllers which communicate with the app's data model to ensure that the views are an accurate representation of the data. Certain actions (such as the play button) trigger the data model to send input to the matlab process, compiling the sound signal and playing it using MATLAB's *audioplayer* function. We used several MATLAB functions, some similar to our lab 3 code, and some from our effects handout, which serve as the lowest level of the app hierarchy, actually turning our data into a sound signal vector. These functions are found in 'ECE485_final_project/Matlab Functions' and include functions for finding note frequency across multiple octaves, generating audio at a frequency, and several audio effects.


OS X Views (.storyboard and .xib)
OS X View Controllers (...ViewController.swift)
OS X Data Model (DataModel.swift)
MATLAB Process
MATLAB Functions

To give an example of the entire process flow, say a user selects a note from our piano view. The user has the option of selecting an octave, and selecting whether to produce a square signal tone, triangle signal tone, or a tone resembling that of a flute.



This note is then added to the track timeline where the user may drag to extend the note length or add a new note. To preview this track, the user can double click on the blue part of the track. It is at this point that the data objects representing this track are sent from the view controller to the data model, where they are converted and outputted as a MATLAB command using MATLAB tone generation and tone manipulation functions we wrote, and then played using *audioplayer*. This all happens as a background process with nothing shown to the user. The user also has the ability to add and listen to multiple tracks, adjust the volume of individual tracks, and add color-coded audio effects to individual notes. These effects are taken from the code samples in the effects handout and made into functions that output signals of the same length as the input.



The audio effects currently implemented are:

- No effect (purple)
- Flanger (green)
- Wah-wah (orange)
- Vibrato (red)
- Overdrive (turquoise)
- Fuzz (asphalt)

One of the effects (vibrato) involves computing delayed versions of the signal, and actually generates a signal longer in samples than the input. Simply using the `vibrato` function from the handout would not fit with our model for representing notes, as it wouldn't fit into the fixed-length units we have for notes. To solve this, we simply take the first N samples of the output from `vibrato`, where N is the number of samples in the input. This keeps everything the same length.

Overall summary of what was learned

Overall, after completing this project, we feel we have a much better understanding of how audio effects are implemented in MATLAB and how these functions can be implemented and used in real-world applications. We also feel we have a better idea of how more robust and professional music creation apps like garageBand are implemented: with a large number of real-time audio effects and samples, more customizability in terms of rhythm, melody, and sound, and more portability/compatibility with standards like MIDI. The abstractions required to build a sort of one-dimensional music making app like TuneLAB are very simple, but with all the options necessary for modern DAWs, it becomes increasingly difficult to present the user with an interface that doesn't feel cluttered with countless knobs, dials, and extra windows. Even our basic prototype requires a separate window for the piano input (which we actually felt was the best option as far as maximizing the speed with which someone could input a song).