

Synthotron

Live Music Performance with Android

18-551 Spring 2014 Capstone Project
Michael Nye (mnye), Michael Ryan (mer1)
<https://github.com/thenyeguy/18551-project>

1 The Problem

Music performance applications on Android are virtually non-existent. Part of this problem is that historically, Android has had audio latency that is too large to facilitate responsive live performance. Latency also varies wildly between devices and OS versions. Some devices have a latency around 100-150ms, while others are closer to 300-350ms. This is not yet low enough for a traditional keyboard interface, but it allows for performance on an appropriate interface in real-time.

2 The Solution

Synthotron addresses the need for a real-time music performance and synthesis application on the Android platform. We avoid the usual pitfalls of long latency times by scheduling music instead of playing it as a response to user touch input. This allows the application to compensate for latency and provide synchronized visual and audio feedback to users. The interface lets users create individual measures by the sixteenth notes and string eight measures together in the sequencer. The sequenced track can also loop to allow users to compose or perform with a backing track.

Synthotron also presents a satisfying range of audio synthesis options. The subtractive synthesizer, FM synthesizer, and drum machine each have their own private effects as well as shared mastering effects. These include reverb, vibrato and tremolo LFOs, a ring modulator, and three-band EQ. These effects are important for users to feel like they are interacting with a full-featured synthesizer and vastly increase the range of tones and music that one can create.

3 Architecture

Synthotron is designed in two connected layers: a Java frontend, and a C++ backend built using the Android NDK. The frontend is responsible for user interface and scheduling notes, while the backend is responsible for generating and processing the audio signal. The two layers communicate using a MIDI style protocol.

3.1 Audio Data

Synthotron uses mono, CD-quality audio. Audio is processed at 44100 samples per second. Samples are 32-bit floating point numbers internally, and converted down to 16-bit unsigned

integers when passed to the audio driver. Audio is processed in system-defined buffers. On our target device (a 2013 Nexus 7) these buffers are 3300 samples long, or 75 milliseconds.

3.2 Frontend

The frontend provides a user interface for creating and scheduling one measure loops. For more details on the interface, see Section 4.

The interface code communicates with a timing manager. The `TimingManager` class tracks the key and notes for a given measure with sixteenth note resolution. Once triggered, it will play the measure on the instrument passed to it via MIDI messages passed to the backend controller for a specified instrument. The sequencer maintains one for each instrument as it plays measures.

3.3 Backend

The backend spawns into a separate thread and continuously generates audio output, feeding it to the audio driver for playback. In order to trigger notes or change effect parameters, a Java interface is provided to pass MIDI messages to the backend.

The backend is an adaptation of an audio processing framework developed by Michael Nye called ClickTrack[3]. ClickTrack provides a generic interface for modular processing blocks. These blocks can be one of:

1. Producer, which produces audio on its own (e.g. oscillators)
2. Consumer, which takes audio input and passes them to outside sources (e.g. speakers)
3. Filter, which takes in audio and produces new audio output (e.g. reverb)

These blocks are connected using channels which contain circular buffers. Blocks have support for an arbitrary number of input channels and an arbitrary number of output channels. This allows for modularity of components, and makes complicated topologies simple to implement.

When requested to handle the next buffer, an audio consumer ask their input channels for audio. The channel either has the requested audio and provides it right away, or asks its parent generator to produce the audio and push it to the buffer. This allows all audio timing to be managed by a single end point and lazily generate audio as needed to maintain seamless playback.

Our project makes heavy use of this modularity and composability; the final project contains 38 separate processing blocks. The majority of these processing blocks were implemented for this project.

4 Interface

4.1 Piano Roll

The piano roll is a common interface for computer synthesizers. It presents the keys of a piano with note name labels on the vertical axis and outlines a grid on which the user can place notes that follow time on the horizontal axis.

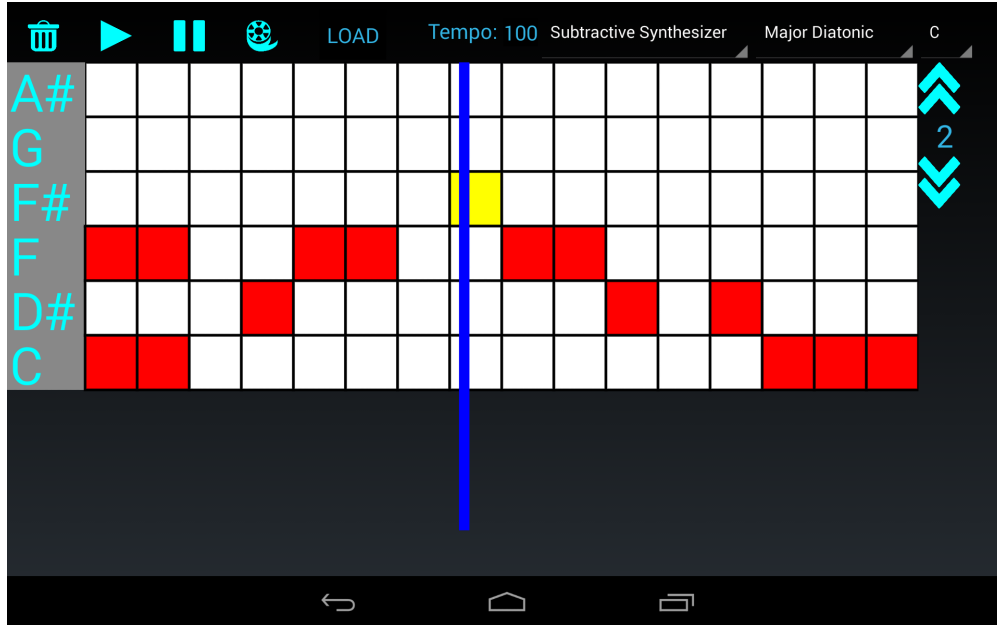


Figure 1: Piano Roll interface

Our piano roll has been simplified to limit the user to a specific key. In most cases, this is not only sufficient, but makes composition easier. The user can choose between diatonic, pentatonic, and blues major and minor scales with any tonic. Restricting notes to one key helps inexperienced composers generate satisfying music with only a few touches of the screen. The interface displays only one octave at a time for simplicity, but notes scheduled in all octaves play regardless of the current view.

Users can also save or load measures using the Android device’s storage. Once saved, these can be replayed from the piano roll or scheduled in the sequencer.

4.2 Sequencer

The sequencer functions similarly to the piano roll but with saved measures instead of notes. It has one channel for each instrument with eight measures of composition. Users can load single measures or entire songs into the sequencer.

The sequencer controls the tempo of each song, leaving the key up to individual measures for key changes. The sequencer can be set to play once or loop continuously so that the user can return to the piano roll and compose a new measure. This facilitates live performance and improvisation, a primary objective of the project.

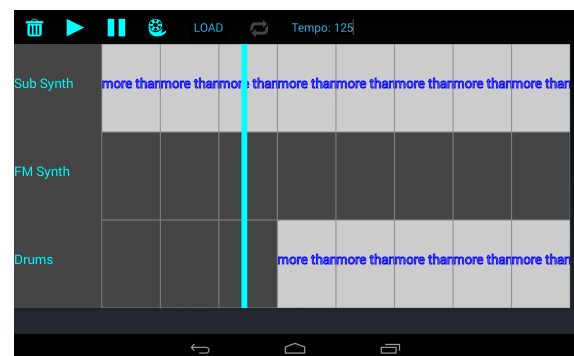


Figure 2: Sequencer interface

4.3 Synthesizers and Mastering

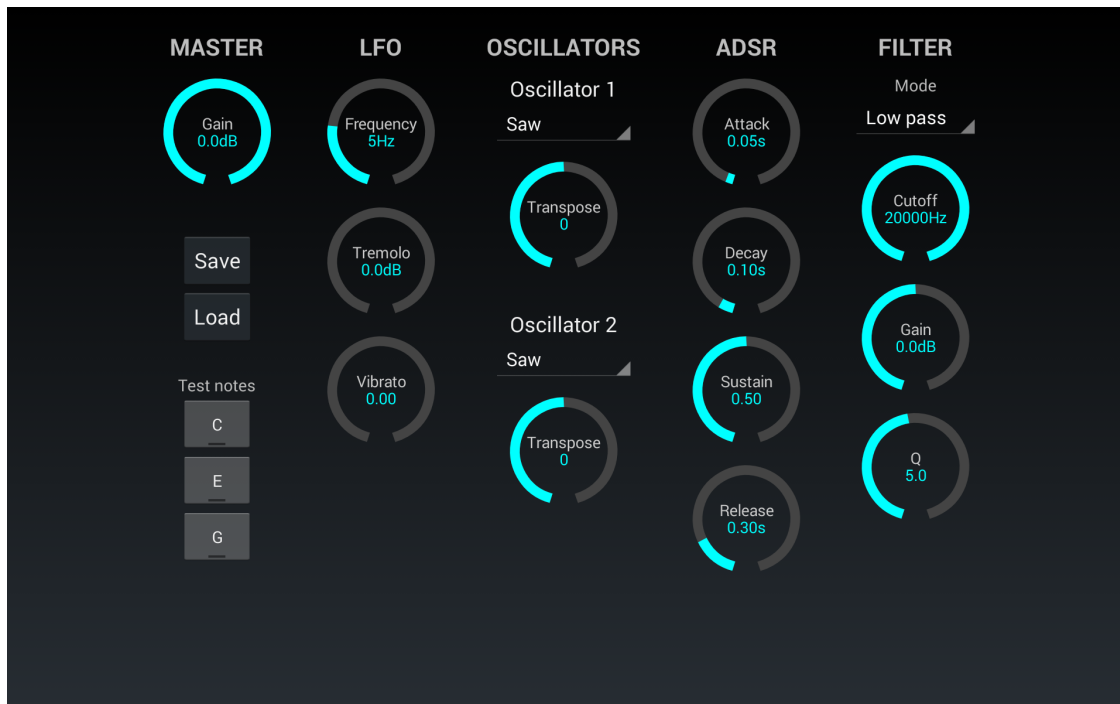


Figure 3: Subtractive Synthesizer

Synthesizer interfaces allow users to prototype synth tones before moving to the piano roll or sequencer. The effects of different configurations becomes immediately apparent through the test notes for each instrument, and the synthesis backend is updated in real time with the values of each knob and selector. Synthesizer control panels also allow for the saving and loading of favorite configurations.

The mastering interface contains controls for the effects on the master channel: EQ, reverb, and limiter.

The knobs on each interface mimic a physical synthesizer. The feeling of controlling knobs on a physical synthesizer is integral to providing a satisfying experience when controlling Synthotron. As any sound technician knows, a board is only as fun as the number of knobs it has.¹ Knobs implement Android’s `View` interface and are designed to act like other display elements. The `KnobReceiver` interface acts analogously to the `OnClickListener`, but can also scale the knob between arbitrary minimum and maximum values. Custom `KnobReceivers` also allow for logarithmic scales, which are desirable when dealing with frequency ranges.

¹Please note, however, that three EQ channels are more than necessary in almost all circumstances.

5 Audio Processing Algorithms

5.1 Oscillators

The most basic element of synthesis is the oscillator. An oscillator is a device that simply emits a certain waveform at an adjustable frequency. Our oscillators are designed to emit the set of fundamental waveforms: sine waves, saw waves, square waves, triangle waves, and pulse waves. These waveforms are desirable for music applications because they have rich upper harmonics, which gives lots of flexibility to carve different tonalities out of them.

Implementing these waveforms in the naive way (i.e. with sharp edges at transition points) implicitly decimates a waveform with arbitrarily high frequency content. The result is a waveform that has large amounts of aliasing; this is especially evident when playing high frequency notes.

In order to avoid this problem, our oscillators use a method called **Polynomial Bandlimited Steps**, or PolyBLEP[5]. PolyBLEP works by rounding the edges around a sharp transition point using a polynomial. If t represents the phase of our waveform normalized between 0 and 1, and dt represents the change in our phase each sample, then the PolyBLEP term can be computed as follows:

```
if (t < dt) {
    t /= dt;
    return t*t - t*t - 1.0;
} else if (t > 1.0 - dt) {
    t = (t - 1.0) / dt;
    return t*t + t*t + 1.0;
} else {
    return 0.0;
}
```

This method is incredibly effective, and can pull down the peak sidelobe aliasing by 50 to 100 decibels for a saw wave. See Figure 4 for a comparison of the frequency content of a naive saw wave, and a PolyBLEP saw wave.

5.2 Low Frequency Oscillators

A typical oscillator is used to generate an audible waveform. Low Frequency Oscillators, or LFOs, are oscillators at sub-audible frequencies that are instead used to modify parameters of filters or other oscillators. Our application uses LFOs to create two different effects: vibrato and tremolo.

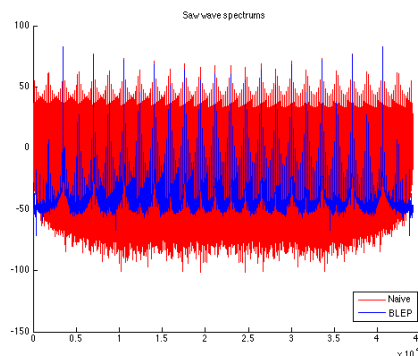


Figure 4: Frequency content of saw wave

Vibrato uses an LFO to modify the frequency a waveform oscillates at. For example, we can use the following equation to compute the instantaneous frequency of an oscillator from its stationary frequency as $f_i = f_s \cdot 2^{LFO(t) \cdot I}$, where $LFO(t)$ represents the LFO waveform and I controls the depth of the vibrato.

Tremolo uses an LFO to modify the gain of an input signal to create a wavering sound. The tremolo gain can be computed as $10^{LFO(t) \cdot I}$, where $LFO(t)$ represents the LFO waveform and I controls the depth of the tremolo.

5.3 ADSR Envelopes

ADSR Envelopes (short for Attack, Decay, Sustain, Release) are used to shape a note when it is triggered. A note passes through four states: the attack, decay, sustain, and release state. Attack, decay and release occur for a fixed increment of time, while the sustain state can last indefinitely.

In the attack state, the gain of the note scales from 0 to 1; this gives the note a fade in. In the decay state, the gain scales back from 1 to the sustain level; this backs the note off and makes the attack sound sharper. During the sustain state, the gain remains constant until the note is told to stop. It finally enters the release state, where the gain scales from the sustain level back to zero.

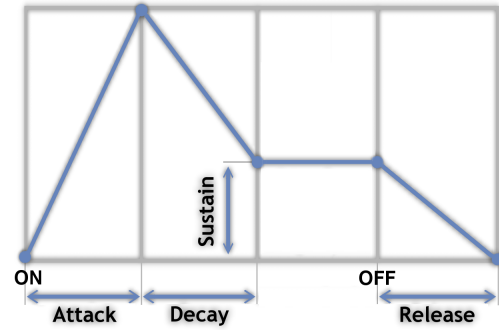


Figure 5: ADSR Envelopes

5.4 Compressors and Limiters

Compressors and limiters are a class of non-linear processors that respond to the signal level of its inputs. The devices first rely on an envelope estimation that determines the amplitude of the signal.

In order to estimate the envelope of a signal, we rely on leaky integration of our signal. This takes the form $env(t) = \alpha \cdot env(t-1) + (1 - \alpha) \cdot x(t)$. However, in order to rise quickly and fall slowly, we use two different values of α . A small value of α is used if $x(t) > env(t-1)$, and a larger value is used if $x(t) < env(t-1)$. A sample envelope estimation is shown in Figure 6.

Once we have an envelope function, a limiter and compressor can use to calculate a new gain. Both devices have a set threshold; in Figure 7 the threshold is set to 0.5. A limiter scales back a signal so that its output

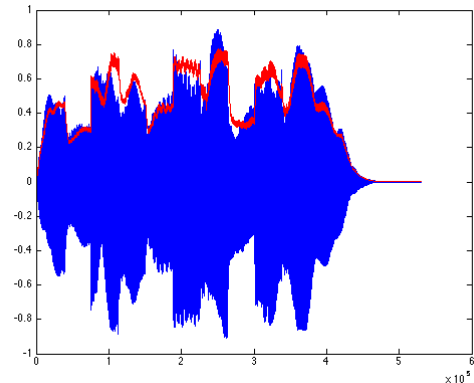


Figure 6: Envelopes Fitting

level never exceeds the threshold. A compressor scales back a signal by a certain ratio. In Figure 7, this ratio is set to 0.5; it reduces the signal by half the value it exceeds the threshold.

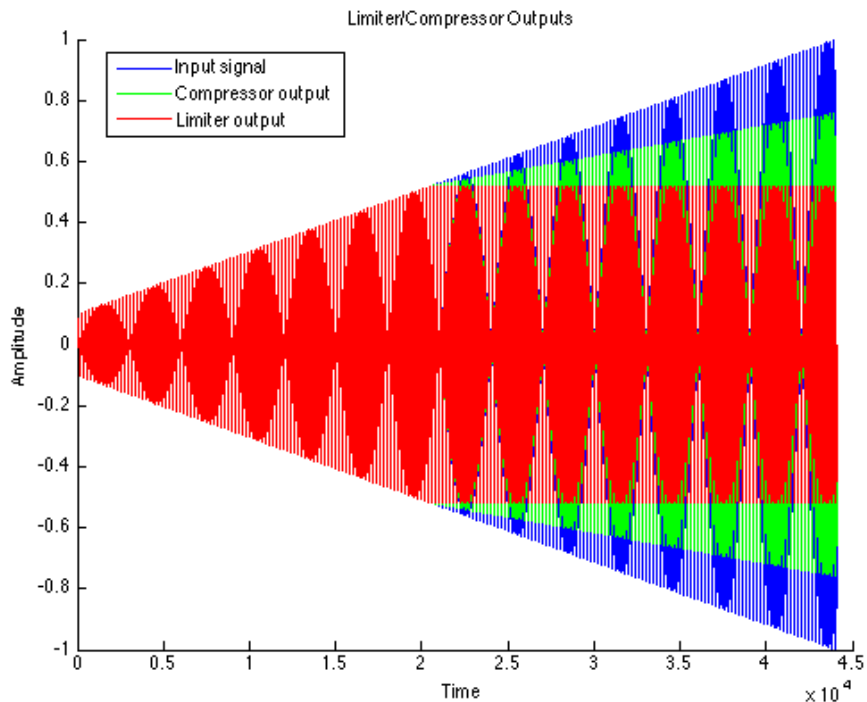


Figure 7: Compressor and limiter outputs

5.5 Reverberator

A reverberator is used to simulate the echoing of a room, in effect placing a synthesized signal in a real space. How to simulate this effect is still a widely discussed topic. For this project, we chose to implement a seminal method described by JA Moorer in the late 1970s[2]. The Moorer reverberator takes on two stages in series.

The first stage is a tapped-delay line. This attempts to simulate the first few arrivals of a multipath propagation. The individual delays should be relatively prime, in order to accidentally create strange frequency responses. Our first stage has 17 taps, the last of which arrives about 80 milliseconds after the input signal.

The second stage is a set of parallel comb filters. The comb filter attempts to simulate the random arrival of a large number of faint multipaths, in order to make the reverberation sound more full. The gains of the comb filter can be set to change the decay time of the filter. Our second stage uses 6 comb filters with delays ranging from 50 to 80 milliseconds.

5.6 Filters and Equalizers

An important part of synthesis is filters that allow carving of the frequency spectrum. While in many applications, sharp filters with low sidebands are more desired, musical applications actually prefer less extreme filters. To do so, we implement first and second order filters, as described in Reference [1].

First order filters support low-pass and high-pass filters, with 12dB/octave roll off. Second order filters support low-pass and high-pass filters with 24dB/octave roll off, as well as shelving and peaking filters.

A shelving filter allows you to specify a gain below or above a certain cutoff, while maintaining 0dB gain outside the passband. A peak filter allows you to specify a gain at a certain frequency, with a roll-off set by an adjustable Q-factor, again with 0dB gain outside the passband.

By composing several of these filters in different modes, we can create multi-point equalizers.

5.7 Ring Modulator

A ring modulator is a very simple device. It simply multiplies a signal by a fixed frequency sine wave. The resulting signal output resembles the input, but shifted in frequency space. It can be used to create interesting pitch shifts and aliasing effects, especially on instruments that aren't inherently pitched such as drums.

6 Timing and optimization

One of the largest challenges for developing on Android is the audio latency time. Each device has a different optimal buffer size and delay between the audio command being sent and output from speaker. In this project, we optimized for the Google Nexus 7 2013, which had an end to end latency of 200-250 ms. After further delays introduced by our processing chain and user interface, final latency was approximately 300ms.

Another challenge of developing for Android is performance constraints. While modern Android devices are fairly powerful, the OS itself is incredibly conservative about allocating CPU time. As audio demands real-time performance bounds, the inconsistency of Android's resource allocation places a heavy limit on the amount of processing we can do. In practice, our application was never allocated more than 35-40% usage of one core on average. As a result, careful optimization of the audio code was required to meet real-time constraints.

The first optimization was to provide heavier use of buffering. While ClickTrack was designed to generate small buffers of audio in order to keep latency low, it was necessary to make use of larger buffers in order to increase cache performance. Additionally, in many cases multiple simple blocks were merged into more functional single blocks, in order to cut down on processing overhead of the generic block code. Finally, we were very careful about the number of processing blocks we allowed to run on the device, including scrapping some effects and reducing the number of voices in our instruments.

7 Live demonstration

The demo included the free use of Synthotron with a variety of presets to showcase all functionality. Attendees toyed with all features of the system while being guided on functionality and were encouraged to loop a rock drum track while experimenting in the piano roll, the main method of live performance with the application. Completed demos included several synthesizer presets and two songs – “More Than a Feeling” by Boston and “The District Sleeps Alone Tonight” by The Postal Service.

Feedback at the demonstration was mostly positive. Users enjoyed the simplicity of creating music over a drum track, and limiting the piano roll to a specific key and scale ensured that compositions sounded pleasant even under a largely random selection of notes. Progress could be made with the interface, which intimidated people less acquainted with synthesizers and digital music production tools.

8 Potential Future Work

- Interface cleanup and polishing: friendlier interface or instructions bundled with the application.
- Additional effects: phaser, distortion, etc.
- Stereo Sound
- Optimization of save data - move from Java Serialization to a custom representation
- MIDI interface for loading/saving songs.
- Recording live audio performances
- Variable loop lengths and note resolution

9 Libraries used

- OpenSLES: Industry standard embedded audio API, provided by Google as the core Android audio driver. Used to send all music to the system for playback. [4]
- Guava: Google Java Collections library [6]
- Gson: Google Java JSON Serialization library [7]

10 Final Work Breakdown

Work was split primarily between frontend and backend. Michael Nye was primarily responsible for maintaining the backend C++ code, while Michael Ryan was primarily responsible for the frontend Java and interface code.

Week 1 (Feb 16 - Feb 22)	Initial proposal and presentation Feb 18 Experiment with Android audio latency Feb 22
Week 2 (Feb 23 - Mar 2)	Nye - Begin porting ClickTrack to Android Ryan - Develop barebones sequencer, piano roll, or keyboard interface on Android
Week 3 (Mar 9 - Mar 15)	Spring break!
Week 4 (Mar 16 - Mar 22)	Nye - ClickTrack functionality on Android, subtractive synth Ryan - Combine interface with ported audio production
Week 5 (Mar 23 - Mar 29)	Nye - Finish port of ClickTrack with Java interface Ryan - Refactor and make robust interface
Week 6 (Mar 30 - Apr 5)	Updates Begin expanding functionality for additional instruments and effects
Week 7 (Apr 6 - Apr 12)	Nye - Additional effects, FM Synth Ryan - Develop sequencer interface
Week 8 (Apr 13 - Apr 19)	Load and save configurations and songs Tone control interfaces
Week 9 (Apr 20 - Apr 26)	Nye - Additional effects and cleanup of audio engine Ryan - Additional effects and cleanup of interface Both - Final systems integration
Week 10 (Apr 27 - May 3)	Completed project integration and working demo
Week 11 (May 4 - May 11)	Final report

References

- [1] lzer, Udo. “DAFX digital audio effects.” 2nd ed. Chichester, West Sussex, England: Wiley, 2011. Print.
- [2] Moorer, James A. “About this reverberation business.” *Computer music journal* (1979): 13-28.
- [3] <https://github.com/thenyeguy/ClickTrack>
- [4] <http://www.khronos.org/opensles/>
- [5] <http://www.kvraudio.com/forum/viewtopic.php?t=375517>
- [6] <http://code.google.com/p/guava-libraries/>
- [7] <https://code.google.com/p/google-gson/>