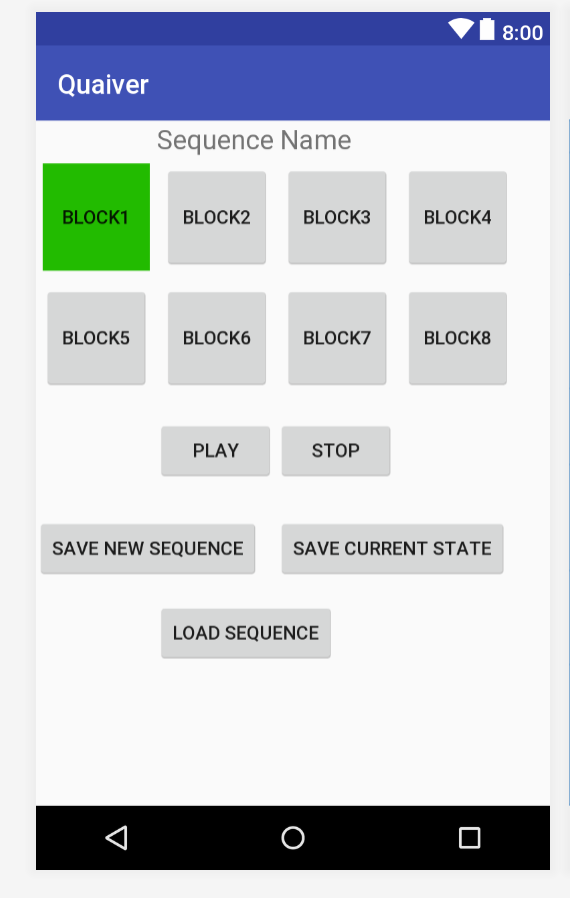
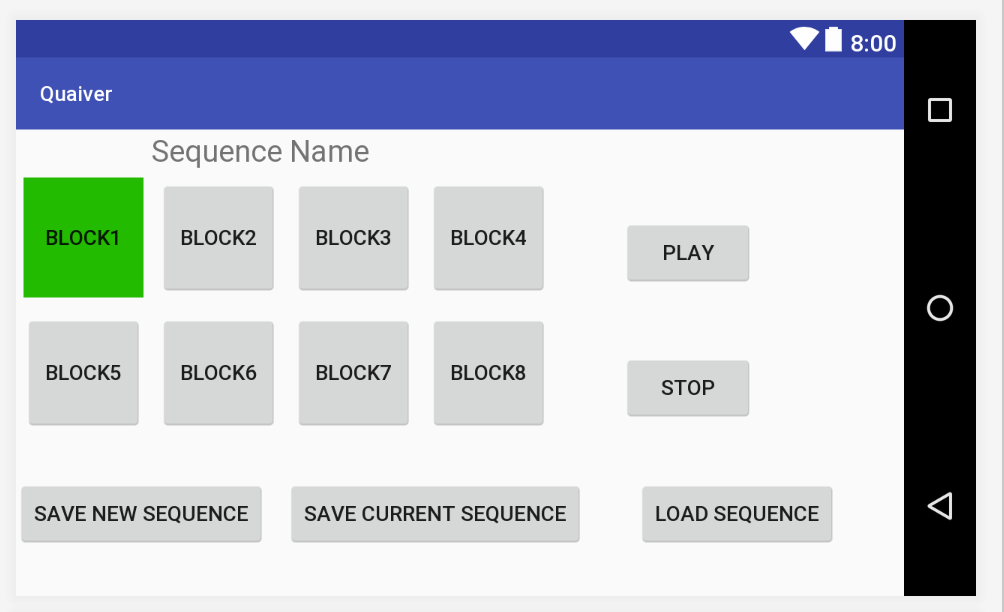
**Quaiver Proposal**

The original vision for Quiaver was a polyphonic sequencer emphasizing a unique customization and flexible control of the synthesizer voices. Due to a series of technical difficulties, most of the original plan was scrapped in favor of an 8-block monophonic sequencer. Issues with sound synthesis halted the tone controls, and a crude sine-wave was implemented instead to play back the sequence of notes. We will hereafter refer to this as Quaver V 0.1, indicating the app is in an early development stage. The existing features of V 0.1 will be outlined; as this version is functional but bare-bones and short of the project goals, we will also discuss those features which we aim to modify or implement in the next iteration, V 1.1, which will hopefully come closer to the project’s vision.

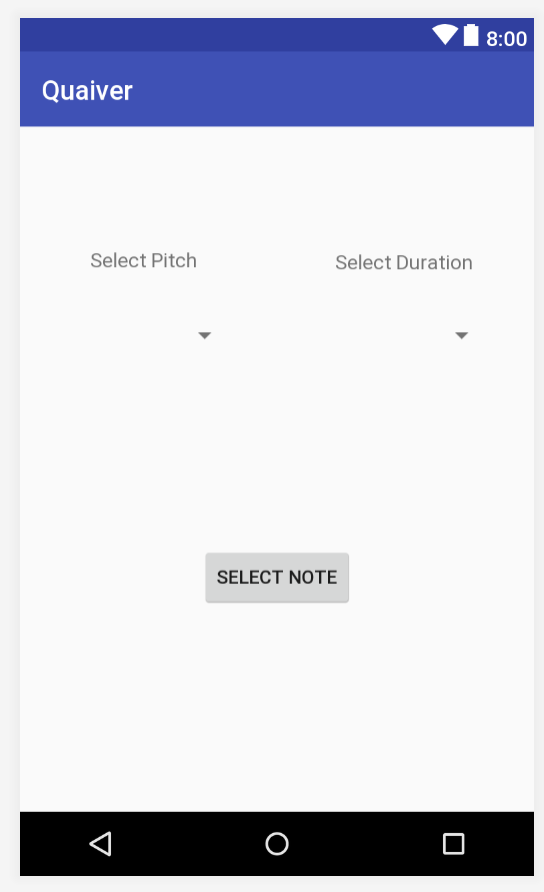
The project is an 8-block sequencer, which means that it stores 8 pieces of musical information and then plays them back in order. The musical information stored is duration (similar to a drum machine) which specifies how long to play that block before moving on to the next, and the pitch (frequency) of a note, which specifies the pitch of a note in terms of the octave and name in the musical scale. Any number of blocks from 1 to 8 can be set when the sequence is not in the middle of being played back, but blocks must be set in order. For example, if one has set 3 blocks, block 4 must be set before setting blocks 5-8. However, After block 4 is set, any of those blocks which have already been set, blocks 2-4, can be altered. When playback is started, the sequencer runs through the blocks, playing a sine wave at that blocks pitch for its designated duration. After playing the last block, it returns to the first and repeats until playback is stopped. The set of blocks and their note information comprise a sequence, which must have a name, though the name can be updated, saving a new copy of the sequence under a new name. Block data is saved automatically for the current sequence. Previously saved sequences can be viewed, and a selected sequence can be loaded, which sets all the blocks to their configuration for that sequence.

**Layout of the MainActivityBlockView**

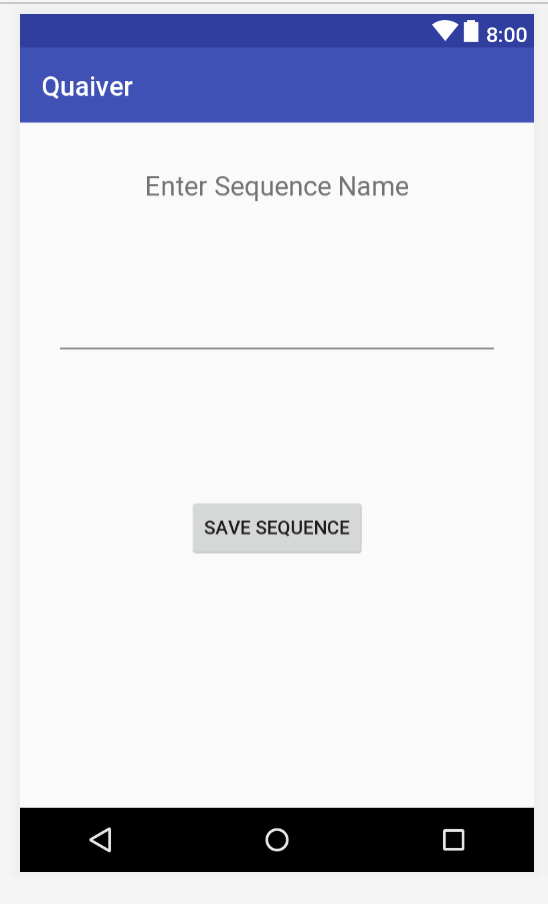




The main activity shows the name of the sequence, and a visual representation of the 8 blocks of sequencer data. When Quaiver first starts, the main activity “MainActivtyBlockView” launches, but there is no sequence data present. The next available block slot is colored green; touching this block starts a “NoteSelect” activity, where the duration and pitch can be set. There are spinners for the selection of both values; possible pitch values range from A2 to G#5, and the corresponding frequency in a standard 12-tone equal temperament. Duration is selected by an integer value between 1 and 16; in this case the value represents the inverse of a note’s duration in 4/4 time – 1 represents a whole note, 2 a half note, 4 a quarter note, 8 an eighth note, and so on. When the desired values are selected, the “select note” button sets a note in the respective block and returns to the main activity. The block will now appear darker to indicate that a note has been set in that slot, and the next open slot moved to the next block and colored green to indicate that it is the next available block slot.

**Layout of NoteSelect**

Once at least one block has been set, the current sequence can be saved under a new name by touching the “save sequence” button, which launches the SaveSequence activity. Text for the name must be chosen, and then if the sequence name has not already been used, the sequence name is saved in the database along with all the current block data, returning to the main activity which now shows the chosen name. The current sequence which has been saved now also has a sequenceID associated with it, which is the primary key generated when a new row is added to the Sequence table in the database. At any point after the sequence title has been saved, the block data can be updated by clicking the “save current state” button. There will be a sequenceID associated with the current sequence, so that is used along with an ArrayList of Notes for all currently set blocks; first all Notes with that Sequence ID in the database are deleted from the Note table, then the new Note values from the ArrayList are inserted in the Note table.

 **Layout of SaveSequence**

Any sequence which has been saved can be loaded at any time. A previously saved sequence can be selected to load in the “LoadSequence” activity, which brings up a list of all sequence titles which have been saved in a RecyclerView. Clicking on a title loads that sequence and goes back to the MainActivity with all the block data from that sequence.

All sequence data including the note values chosen for each block is stored within a static variable that is an instance of the VoiceController class in MainActivityBlockView. This allows it to be accessed from any activity, and re-loaded when orientation changes for the main activity. The VoiceController class is an intermediary between the sequencer functions of sound playback and the storage of current sequence data. It has a hashmap of Voice instances (in this implementation, there is only one voice, but the classes and several layers of composition were designed to accommodate a more complex system not yet implemented in which several voices, each with its own sequence, can play back simultaneously) in the “voices” variable. The instance of Voice has a Sequence, and the instance of Sequence actually contains an ArrayList of Note objects which hold the block data, the name of the Sequence, and its ID. When a block is set, the value is changed in memory via the static variable, an instance of Voice which has an API to get and set sequence data. It also accommodates the interface to another class, SequenceController, which regulates the audio playback. Thus all sequence data for the current sequence is within the VoiceController object, which is static, allowing the database methods to get data that is stored there, and load data into that object.

The database is organized with a 2 table schema. One table holds Sequence information, mainly the Sequence Name and an integer ID serving as a primary key. The other holds Note data, which has a primary key integer for ID, an integer for note Length, a real (decimal) number for Frequency, and an integer for Sequence ID serving as a foreign key back to the Sequence table. There is a one-many relationship between Sequence and Note, since any one Sequence has many Notes. When a sequence is inserted into the Sequence table, all of its note values are inserted in the Note table and having their Sequence ID set to the newly inserted sequence’s ID. Similarly, when an existing sequence is updated, all of the Note values which have their Sequence ID equal to the sequence’s ID are deleted, and then the updated note values inserted. Loading a sequence is done by selecting a sequence with a given ID from a list of sequence names in the recycler view and returning all notes from the Note table with a matching Sequence ID.

**Features for Version 1.1**

The first feature which needs to be changed is the sound generation engine. The sine wave must be replaced with an actual synthesizer; optimally this would be a synthesizer built out of components so that different aspects of the sound can be controlled. Android has no default library for this, and while playing back samples is relatively easy, synthesizing sound is much more involved. The goal is to find a synthesis engine which allows for the selection of a basic waveform (sawtooth, triangle, sine, rectangle), control over an ADSR envelope, and control over a filter’s low pass frequency and resonance level. The capacity to compose additional modulations would also be nice; generally this takes the form of an oscillator whose frequency and gain can be controlled, and set to modulate pitch, volume, or filter frequency. However, if the engine is built up from more primitive components, additional modulations could be offered. At first, this would be paired with an additional activity that allows the user to change some of these values, so that when the sequencer plays back the tone can be controlled and dynamically modulated in real time.

Hand in hand with the synthesizer engine is the link between the sequencer and synthesizer. Currently a primitive timer system is implemented to control when the sequencer plays (on/off), how long it stays on, and what frequency to play. A global timing system is needed to “schedule” the blocks for playback. The note on/off gate could easily be made to conform with MIDI standards, and note frequency converted to MIDI note numbers (there is already a class which can do this) so a synthesizer engine which uses the MIDI standard to schedule events would be a natural choice. The current class which controls the sequence playback, SequenceController, could be altered so that the timing system is integrated into a MIDI compliant event scheduler. This is internal, and not reflected in the GUI of the project. But it is crucial if the sequencing aspect of Qaiver is integrated with the sequencer. Currently the tempo of playback is hard-coded, but this is also something that the user should have control over. Optimally, we want the tempo to be variable during sequencer playback, which means that the timing system must scale the note values accordingly. So even though the sequence blocks are pre-set in pitch and duration relative to the playback loop, they must be dynamically scheduled during the playback loop.