KARLSTAD UNIVERSITET

beatr

Real time, interactive Android app for creation and composition of electronic music.

Kristian Sällberg

8/19/2012

This document contains technical information and background reasoning about the beatr project.

Index

Introduction

First off, I would like to thank the University of Karlstad for providing this very open ended and free of constrains summer course. And I would like to thank Martin for accepting my request to make a sound application even though I didn't know anything about the field. I have a learned a lot this summer.

This project report will present a good deal of the reasons as to why I wanted to work on this project, a real time music production tool. In the background section, I will describe why I chose the Android platform and in later sections I will describe why I probably should've chosen a different platform.

Overview will give a tour of the app and it's different views. It will explain how to use the app and what different modes and views it has from an end user perspective.

The detailed description section deliberates on some important key implementations and gives insight into why I wrote some pieces on code the way I did. I explain how the audio itself is generated. I describe what controllers are the most important ones to know about.

Finally, I summarize the project. I write about what I've learned and discovered. I add some of the great things I've experienced this summer and some somewhat disappointing facts about the app.

Background

For a long time, one of the top projects on my I-really-want-to-do-this-list has been a music production tool. However, I had no knowledge of how to create audio, and I never felt I had time to learn the basics of audio programming. So with this course, I finally got a chance at creating some kind of tool to generate audio. I realized I would not have time to create the actual sound generating (synthesizer) code myself. I needed to find a software synthesizer that would work out of the box on the platform of choice.

At first, I wanted to build the application in C++ using some kind of open source framework such as openFrameworks or libcinder to build an old school desktop application. But roughly a week before the start of the course, it appeared to me that a mobile app would be very interesting for a many reasons. Mainly because the course is short, seven weeks of programming is not a lot of time compared to the years of development time that have been invested by major desktop music application creators.

Another reason is that now when everyone has a smart phone in their pocket all day long, it makes sense to provide some kind of simple music application for ordinary people to quickly prototype beats or just to kill time. Yet another reason is that smart phones usually support multi touch, making the application feel more like a real instrument. On top of this, there are not many real time music generation apps for the Android platform; this point will be further deliberated later on in this document.

I decided to move to Berlin to work on this project. Berlin houses many companies focused on sound development, Ableton among others.

Currently it's not possible to write sound bytes directly to the sound chips of Android devices via the SDK (Software Development Kit). There are some ways to get around this but these

solutions are not very easy to find, and when you find them, they are hard to compile and hard to get running within the Android framework. It took me a good week of searching through music forums and even software synthesizer forums before finding a couple of Android ports of existing software synths that did not lag too much and that weren't too awkward to compile.

The combination of Android being an awkward platform to develop sound apps for and it being reasonably difficult for developers to find the ported synths has scared away many developers from this segment. Most sound developers have chosen to create their apps for the IOS platform instead. Even major players like Propellerhead (creator of Reason, and many others) have chosen not to make an Android version of their IOS hit Figure. Their CEO stating the screen size fragmentation of the android ecosystem as a major obstacle for creating great applications.

All of the problems stated above combined, leave the market for real time, software synth backed, audio generating apps for low-end devices pretty much empty. And that's what I wanted to change with beatr.

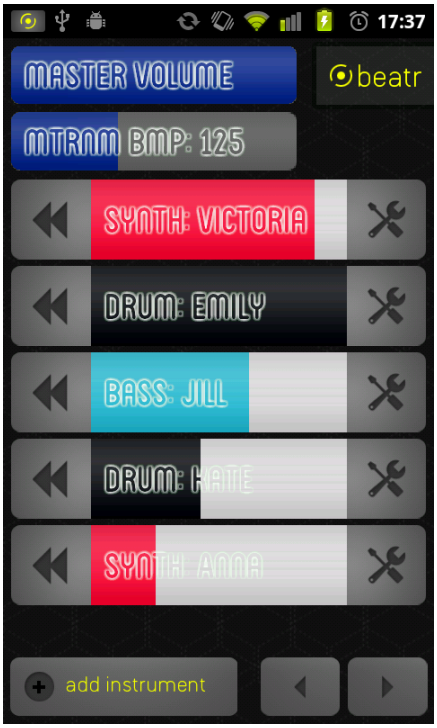
Overview

The application is basically a collection of instruments playing simultaneously. The user can add any amount of instruments. There are no restrictions as to how many instances of an instrument users can add, because different android devices will be able to handle different amounts of instrument instances without performance issues and there is no way I can know what each and every device can handle.

Except from the Main Menu, Credits View and Help View, beatr has two modes. The *Overview Mode,* (visible in the attachments section) which shows all existing instrument instances in a list is what users first see when opening the app and pressing *mixr*. The amount of instruments that fit into one window varies depending on what device the user is running beatr on. In the bottom of this view, there are arrows that allow the user to see the next or previous n number of instrument instances.

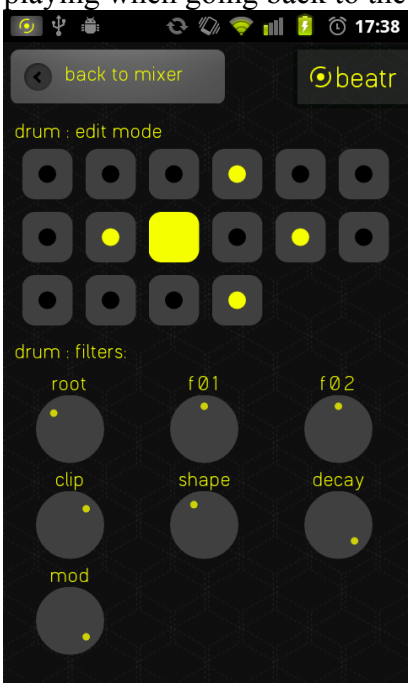
The second mode, *Edit Mode*, (also in the attachments section) allows the user to edit sound settings and record what they want the instrument to be playing. When recording something in the edit mode and then going back to the overview mode, recorded instruments will play back whatever was recorded.

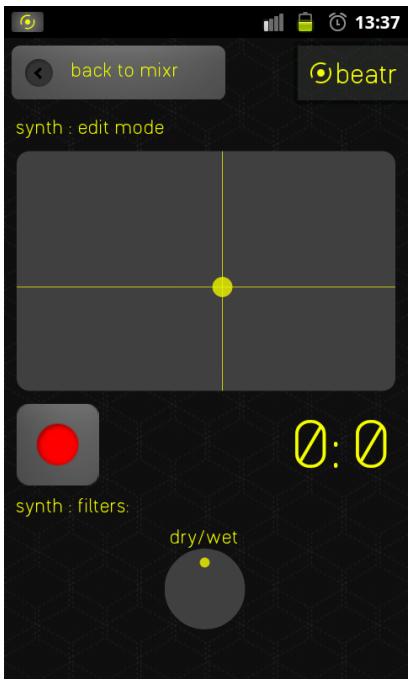
The Main Menu view, it has three buttons. I realized some things in the mixer view / Overview Mode are not very obvious to users, so I had to implement a Help View. I chose between linking these views in the Android auxiliary menu and linking them in some kind of app menu, and I decided on the latter alternative because the Android menu is deprecated from Android version 4 and onwards.



The Overview Mode (mixr) shows all added instruments and a volume control to them. Every instrument featured has a button to rewind the recorded beat (if any), and a button to enter the instrument's editor view. The Overview Mode also has a master volume controller that affects all added instruments and a slider that tells the metronome the BPM rate to play in. It has the "Add Instrument" button that opens a small overlay right over the button when clicked. Finally the view has the previous and next buttons that controls what instruments are visible in the list.

When entering a drum's editor view, you see sixteen drum pads, each able to be set on or off. Also, the view has seven knobs that affect different filters of the drum. The drum will keep playing when going back to the mixer/Overview Mode.





The synth and bass Editor Modes are essentially the same. It's possible to change the pitch by moving the XY-slider in X dimension. Higher pitches are to the right and lower to the left. The Y dimension controls a filter that makes the sound subtle and dull, it is completely off to the bottom and increases to the top.

Beatr is focused on creating simple repetitive beats very quickly without any required knowledge of music and composition. To make it easier to control what sounds play when, I decided to abandon the timeline concept that's an integral part of almost every desktop sound production tool. Instead of a timeline, it is possible to control what instrument plays when by changing the volume bar that's connected to every instrument.

Beatr has a metronome that keeps playing a "tick" sound in the ranges of 80 to 200 beats per minute. All drum instances are synched to the metronome, they make their drum sounds when the metronome sounds.

There are currently three instrument types in the app; the *bass* and the *synth* are essentially the same instrument in terms of how they're used. The bass is restricted to low pitches and the synth starts from roughly where the bass ends and then continues to higher pitches. They both have a master volume property and individual volume properties as well as a property to control the frequency of the oscillator (the measure of how often the curve turns from low to high in the speaker). They also have a property that controls a value called wet/dry, it's something I've been inspired by from filters in the production tool Ableton Live.

The third instrument, the drum, is completely independent from the synth and bass. It has seven settings, on top of individual volume and master volume. I will not go into detail about what they do, because I have not designed the drum's synthesizer architecture myself. The drum machine in beatr currently features sixteen drum pads. The drum pads can be pressed to be activated and pressed again to be inactivated. The drum machine then then loops through the drum pads in the order they are added, 1-16, at each metronome tick. If a drum pad is activated when it's looped through, the drum plays a sound according to all the settings.

Detailed Description

In the Android platform, each visible layout in an app is usually associated with an Activity of its own. An activity is kind of a separate island on which you can have a piece of your program, isolated from other views. When an activity instance is paused, much of what it has stored in memory is removed from there, allowing one or several new activities to allocate a larger chunk of the system memory. This is important to make use of, because if an app is wrapped entirely in one Activity, changes between states and views will not remove unused data from memory.

I choose to use an MVC (Model View Controller) framework for everything that lives within my activities. I use a friend's MVC framework called aMVC (Android MVC). Usually I use the model layer to store different properties (such as the decay property of a drum beat) that are changed by the user and that shouldn't be removed if the Activity dies.

The aMVC framework makes the controller the most powerful actor. Controllers create models and views. And controllers create sub controllers, which in turn might have sub controllers themselves. It's important to note that Views in the aMVC framework are not Android platform views. They merely collect a bunch of Android views. This decision was

taken because as a developer you usually have a lot of views for which you might not need a controller; they might be too simple for that. Imagine having fifteen different LinearLayouts to allow some kind of wrapping to allow a certain placement on the screen. In that case it's nice to have one container that holds the many sub views.

The Main Menu, Help View and Credits activities do not use MVC. They are simple standalone activities with the model, view and controller type of behavior all in one class. I wrote the code like this because those classes are so short (less than one hundred lines) that I felt separating them according to MVC would just increase complexity without proving major scalability.

I'm using the software synthesizer library *Pure Data* to generate sound data. It writes data directly to the smart phone's sound card. In the Pure Data environment, developers define so-called *patch files*. They keep all properties and sound generators. Pure Data has support for many different types of sound generating units, including oscillators and phasors. It also supports a wide range of filters (such as a saw tooth filter) and other wave bending constructs. Pure Data patch files are usually created using graphical programming in the official Pure Data IDE.

I will not go into details about how the patch file creation works and I have mostly used already created patch files in this project. All communication to the Pure Data framework is handled by sending a String and a Float with the property name of what to change and it's new value. This is the raison d'etre of my app; remember all property names, and their current value, and when the user changes something – tell the synth!

In Pure Data it's not possible to change property names when patch files are already loaded into the environment. This was a big problem initially because I needed separate names for each instance of an instrument. I needed synthprop-a, synthprop-b, etc. (in this case they would represent two different instances of a synth). My solution to the problem is wrapped in the class FileModifier.java; it basically loads a patch file (file extension .pd), changes all property names to be unique for the new instrument, saves it as a temporary file and then loads it to Pure Data. FileModifier consults the InstrumentTracker class to get unique names for new instrument instances. Patch files are always loaded into the systems when the users presses "Add New Instrument" and then chooses what type to add. This bubbles an event up to the InstrumentMixerController, which consults FileModifier to generate unique names and then adds the modified file to the synth.

From the point of getting loaded into the Pure Data system, all patch files are constantly generating sound. So it would not be possible to turn instruments on and off out of the box. I solved this problem by defining several different multiplication units right before the data is sent to the sound card. Generally, I have a few different volume multiplication units, one for master volume and a couple other individual ones for each instrument. These units are able to steer the volume of the instruments, and to disable an instrument. To "disable" sound streams, I simply set the volume of one multiplication unit to 0%. Each time users hear an instrument sound; the volume is quickly toggled between 100% and 0%.

A central feature of the app is the ability to record and play recorded audio. This following section will explain how I choose to implement this functionality. The flow for the bass and the synth instruments are identical. The drum cannot be recorded, instead, the user turns a set of buttons (I call them Drum Pads) on or off, and when the button is validated (they are

validated one at the time, in a consecutive order, based on the Metronome's tick event) it will send a signal to the Pure Data drum instance that it's time to make a sound. If the drum button is turned off, nothing will happen.

I never record audio byte data, instead I record basic events dispatched from user interfaces. When playing them back, I simply pass the data obtained from those events to Pure Data at the appropriate point in time.

The synth and bass instruments make use of the se.purestyle.beatr.helpers.beatplayer package to record and play recorded instructions. The package houses the Recorder class among others, which implements Runnable and is intended to run in it's own thread. When started, it basically starts counting from 0, and is then waiting to receive lists of messages of a pair of String and Float for each millisecond running. The String is the message to give Pure Data (it is unique and contains the name of the instrument instance and the property to change) and the Float is the setting to that property.

The Player class also implements runnable, the reason to have the playback functionality in different Threads is because all recorded beats are not the same length. Some beats are longer and some are shorter, they all need to be reset at different points in time. Each Player keeps track of what millisecond the current recorded instrument is playing, and then sends the recorded values to Pure Data when needed and resets the individual current time when needed.

Beat.java is the data structure that keeps track of all recorded properties and their value at certain points in time. It uses a map where the key is a long, so that each millisecond can have it's own list of instructions, and where the value is an array of pairs with Strings and Floats.

BeatPlayer keeps track of all Player instances and also has a method to quiet everything down or to play everything (or to play just a single instrument). BeatPlayer keeps both bass/synth instances and the drum instances and it calls the proper methods in those instances to make them be quiet or to make them play. Everything is always activated when entering the Overview Mode. And everything except the current modified instrument is always quieted down when entering a view that is part of the Edit Mode.

When a user adds a new instrument from the "Add New Instrument"-button in the Overview Mode, two important things happen at once. One AbstractEditorModel subclass is created, note that the model of editors is only ever created once, but new controllers and views are created every time the editor is entered. This is because the model needs to store values for as long as the instrument is still in the app. You want your old settings to remain even if you've stepped out of the editor and want to enter it again. Editor controllers then load all values needed from the model and set initial values of their UI components to match what's in the model. The second important happening is the creation of the InstrumentController, together with its model and view companions, they are only created once and never removed as long as the instrument is still alive.

AbstractEditorControllers are responsible for all communication to the Pure Data framework concerning all properties other than the master volume and individual volume. The InstrumentControllers are only responsible for sending master volume and individual volume messages to Pure Data. The reason as to why I created this separation of flow is because all instruments have the master volume and individual volume properties. So the

InstrumentController can be identical for all instruments. But the editors differ in what properties they have. Both controller types receive a string called internalPdName when instantiated, this is the unique name of the instrument they are controlling.

InstrumentMixerController is the most important piece of code of the Overview Mode. It basically creates everything in this mode, and also listens to events in all of its children. Its view (InstrumentMixerView) has references to many of the buttons and other UI components deeper down in the nested hierarchy of layouts that were needed to place everything where I wanted it to be.

The InstrumentHolder MVC group is responsible for holding all instrument representations in the Overview Mode, i.e. all volume sliders with links to the instrument's editor. Its view class wraps functionality to measure the device's screen height, and thereafter making sure a maximum amount of instrument views are visible at the same time on screen. Its controller listens to the next and previous buttons in the view, and when an event is fired, the view calculates what instruments to show based on where in the list of instruments the first instrument to show is.

The Metronome is the only object that produces sound as hearable by the user that is not played by Pure Data. The metronome sound is a simple MP3 file that is repeated over and over again when it should be according to the current beats per minute property defined in MetronomePlayer. It uses a java Timer to be able to know when to dispatch the tick event, and it's possible to add observers to the metronome, so other instruments can be synched to it. Currently only drum type instruments listen to the metronome for instructions on when to play.

I will not describe what the classes in the se.purestyle.beatr.view package look like. They are all subclasses of the LinearLayout or View classes in the Android SDK. And frankly, they do not do much of interest. They simply place stuff on different positions on the screen, and redraw my custom UI components. Android XML layout files could’ve replaced most of these classes. Neither will I describe any specific model classes any further. All they do is keeping states and properties.

Results

Beatr is published to the Google Play market. Please download it from the link below.
https://play.google.com/store/apps/details?id=se.purestyle.beatr

Summary

I have succeeded to make Android phones generate sound based on input received from user interface components I have designed and programmed myself. That was my basic goal. It seemed to be a rather easy goal, and I thought it was going to take a lot less time than it actually did. I was wrong. After spending over a week of production time trying to find anything that would make a sound and almost giving up and picking something else for a project, I finally found Pure Data. Also, the Android framework took longer time to learn than what I had expected. I knew java well before the project started and I can't really say I learned anything new in terms of the language. But I have learned an incredible amount of other stuff. From audio terms to how the file system works in Android devices, this summer has been very rewarding from a learning point of view.

I was very naive and clueless about the Android platform's sound API. I did not know about any of its limitations. If I had – I would probably have chosen to do something else, or at least to do it in another platform, probably IOS. But after finding Pure Data, I felt that being one of the few developers of audio apps for the Android platform turned into an advantage compared to if I would have developed for IOS, now that the app is on Google Play, it will probably face less competition than if it had been published to the IOS counterpart.

Sadly, the bet to skip having a timeline and instead have a lot of instruments and just activate them by using their individual volume controllers wasn't a great idea, at least not in my development device, a ZTE Blade. The processor is simply too slow to handle more than three instruments, and it starts lagging too much for the app to be usable when having as many instruments as one would need to create something that sounds like music. On the other hand, I have used the app on more powerful devices and then it worked out nicer. But I still see this as a failure because I wanted beatr to run on old and slow devices.

If I had started from scratch with this project, I would probably have wanted to team up with someone who has insight in music production to help concept the app. I had absolutely no idea what to implement other than that I wanted to create some kind of synth and some kind of drum machine. For instance, a couple of weeks before the end of the course, a friend who is producing tracks on his spare time complained about the fact that I had no metronome. So I had to add it, and synch the drum machine I was working on to be aligned with the metronome. On the other hand, knowing little about the subject was a good practice in being dynamic and working in an agile manner.

If starting over, I would definitely have chosen to implement most Layouts as Android XML files. I have implemented all views (the view part of my own UI components and all layouts) in Java right now. I did that to get total control of everything. But later on I realized I could have gotten the same level of control with plain XML files. That way, I would have saved many lines of repeated code. All layout functionality and settings provided by the Java SDK is supported in the XML mode anyways.

Another failure was not getting multi-touch working. I simply ran out of time before getting the grips of how to use it. So now it's not possible to adjust several controllers at once, which is a shame.

The move to Berlin was a risky bet as I had nowhere to live and nowhere to work during the daytime. But after having fixed all the everyday issues such as a place to live and an office space, it turned out Berlin gave a lot of inspiration back. Weekly programming user groups for a multitude of programming languages and its many techno clubs gave inspiration I hope will last for a long time after this course has ended. I also made new friends and possible business contacts I hope will prove valuable in the future.

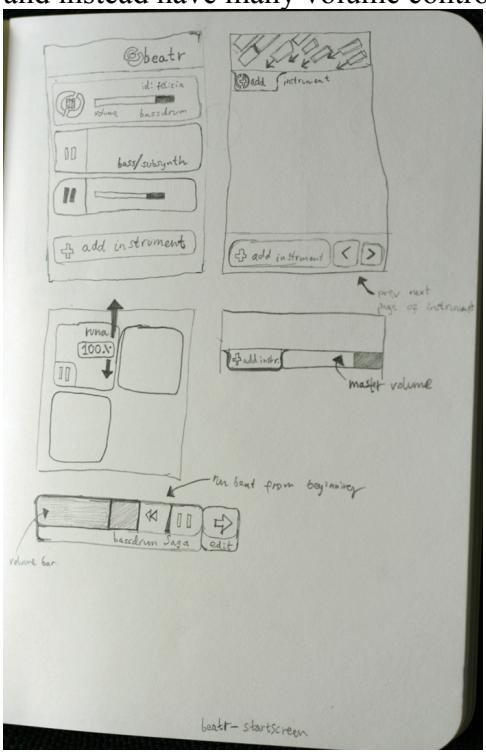
All in all, this course gave me so much back. I finally got to try some audio related coding out, got to move to Berlin. I got to learn so much about the Android platform that I could probably work on simple commercial apps from now on.

List of References

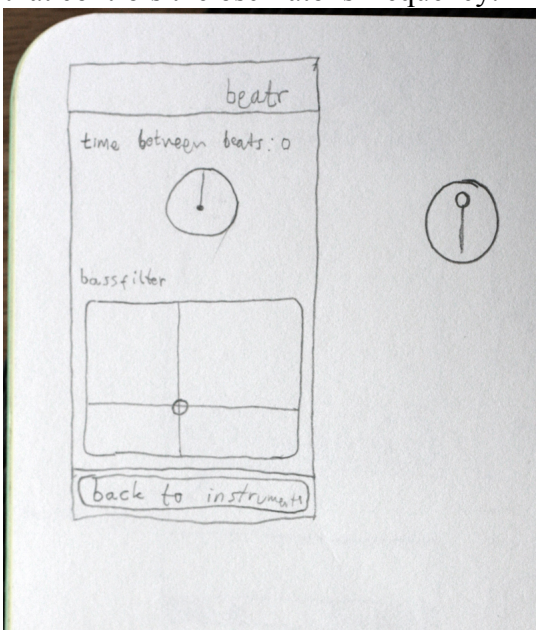
http://www.synthtopia.com/content/2012/05/07/propellerhead-ceo-ernst-nathorst-boos-on-why-they-dont-make-figure-for-android/ [2012-07-13]

Attachments

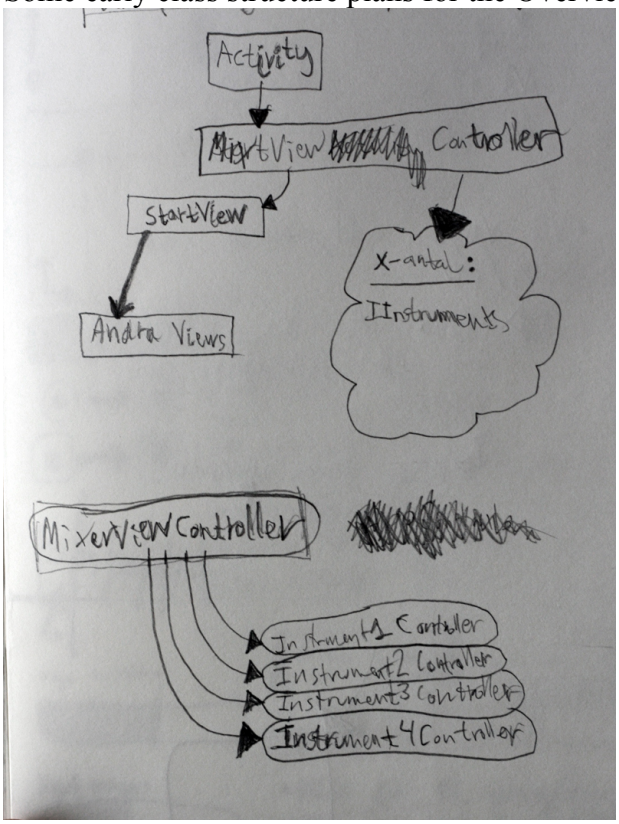
Let's show some of the sketches that I drew the weeks before the project started. First off, this photo shows the initial idea of how to strip the timeline away from a music production tool and instead have many volume controllers.



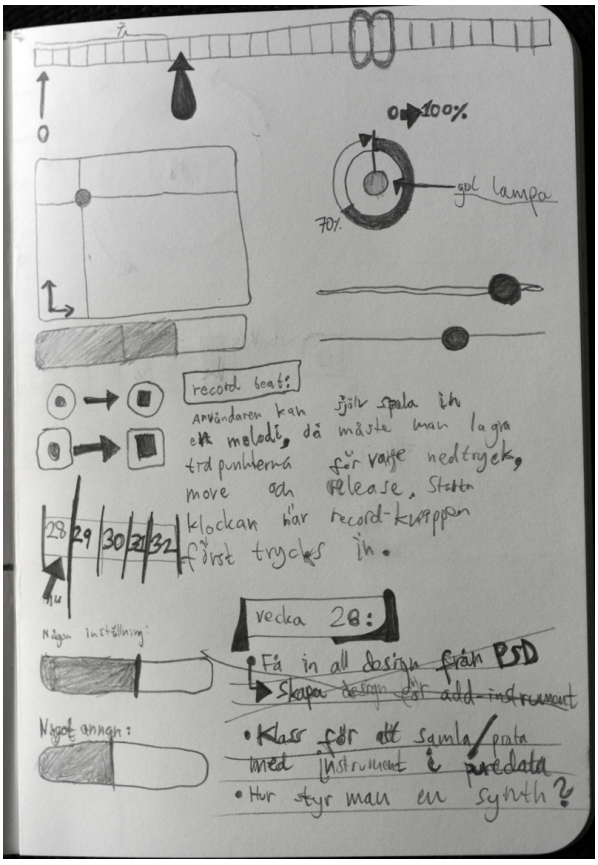
This is from when I conceptualized how to control the bass and synth instruments. I didn't want to create a classic piano so instead I created a simple XY slider (two dimensional slider) that controls the oscillator's frequency.



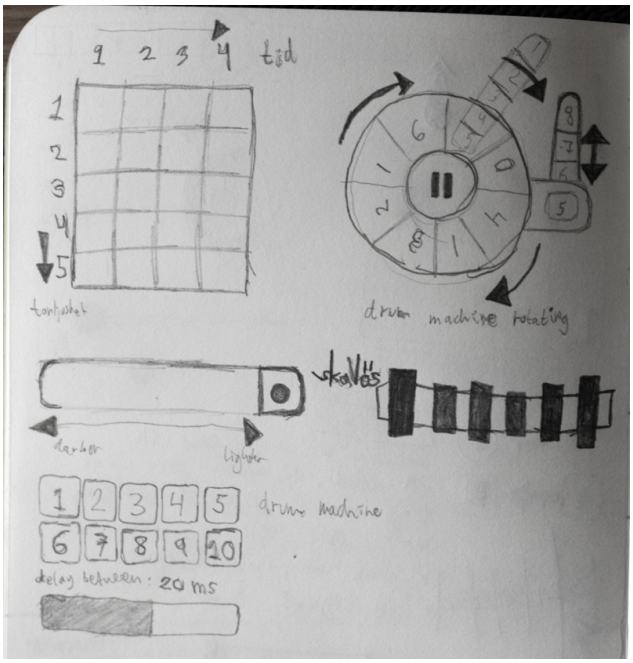
Some early class structure plans for the Overview Mode:



Different sketches and ideas for volume controllers, record buttons, the two dimensional slider etc:



I needed some kind of way to play drum sounds. First I wanted to create an advanced, flashy device that could schedule different sounds to be played after each other (the round thing with a pause button in the middle below). But I thought it didn't add much value and instead I created the drum pads that are in the drum machine now, they are in the left bottom corner below.



The last week of development was pretty intense, here's the focus list: