Introduction

For my Advanced Audio Applications project I chose to create a Drum Replacement application in Pure Data. Essentially the application analyses percussion data input by the user and outputs a different sound as a replacement for the input.

Drum replacement usually arises out of the desire to change an existing sound in a recording without the hassle of rerecording the performance. Take, for example, a situation where after a drum recording session, the engineer realises that whilst the performance is spot-on, the microphone on the snare was unfortunately in an inappropriate position. Thus making the sound of the snare unsuitable for the style or theme of the production. Rerecording the drummer would not be favourable, as it would no doubt incur a considerable cost in time and money especially considering the recordings are only let down by a single track. This is where drum replacement comes in.

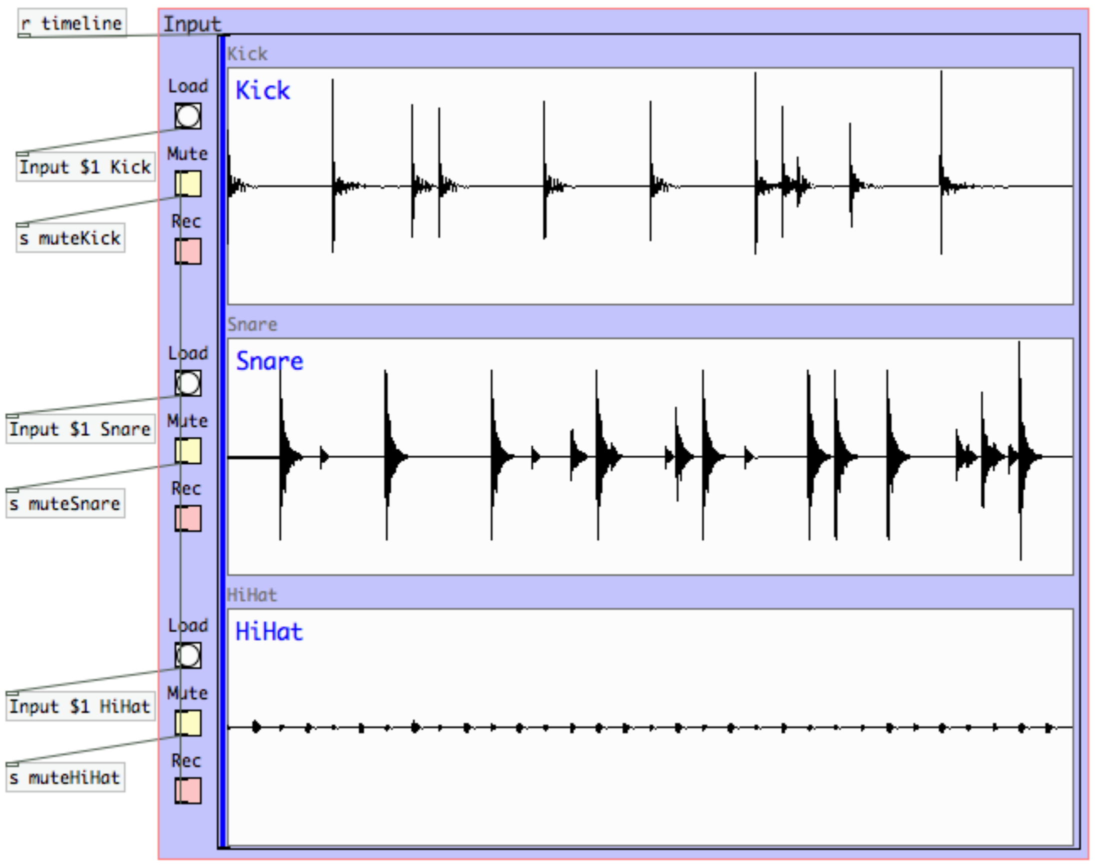
In the past, to rectify the situation the engineer would most likely have to go through the track and 'drop' a new sample in as replacement for the previous. However, as this is done by hand the processing time would be increased substantially. In these situations an automated process for drum replacement is a very valuable option.

Drum replacement programs are mainly found in synchronicity with Digital Audio Workstations, tending to appear as either plug-ins or as standalone applications. Pure Data is a great application to develop such a program as it enables the end user to utilise drum replacement techniques with little prerequisites.

Also it is interesting to note that these programs are not just limited to a total replacement of the target material, but they also as an enhancement of texture/timbre or even an incorporation of effects triggered by the source.

Implementation

Input Section



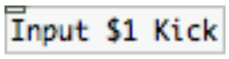
The **'Input'** section deals with importing user-selected audio and saving these files to tables. The tables are represented as the light blue boxes in the centre of the section, which graphically display the audio waveform. Additionally there is a timeline in dark blue that passes over the tables to indicate a point in time at playback. The timeline receives its data from the **'Timeline'** abstraction in the **'Playback'** section.
The options available to the user in this section are **'Load'**, **'Mute'** and **'Rec'**.

'Load' opens a prompt to the user directing them to select an audio file from their hard disk.

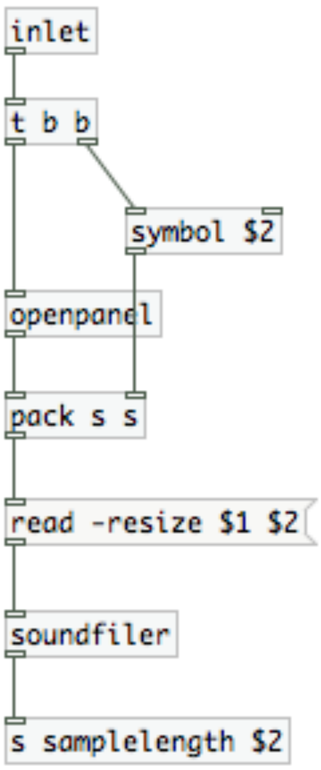
'Mute' silences the respective track. This button sends the information to the **'Playback'** abstraction of the **'Transport'** section and to the **'Replacement'** section.

'Rec', which stands for Record Enable, prepares the respective track for bouncing. It achieves this by muting the other two tracks.

Input Abstraction

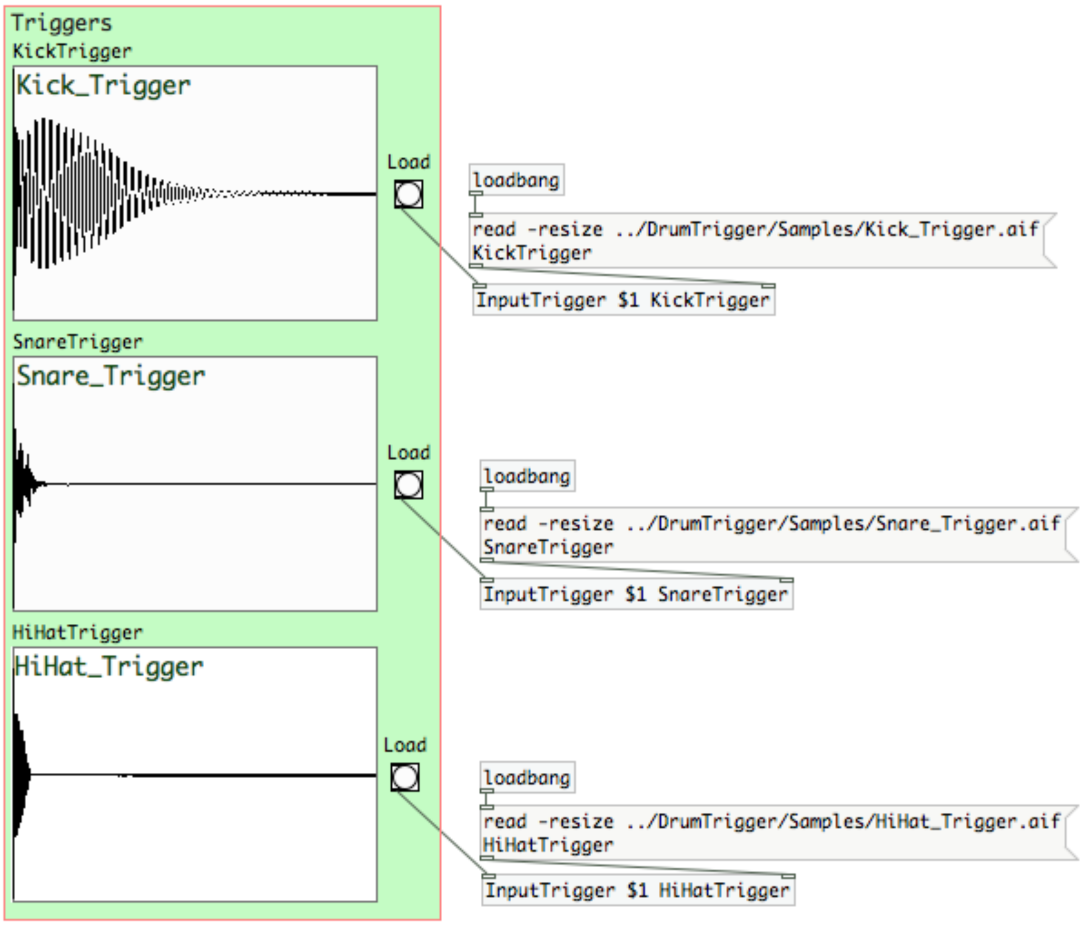

Input $1 Kick

This abstraction houses the method to get the Input data from the user and distributes information related to the audio files. It has two arguments: the first being the name of the file the user selects and the second being the table name it is to be put in. The `pack s s` object here handles both arguments and distributes them to the appropriate message box. It also bangs the message box so that soundfiler can deal with the file and input it to the relevant table. After processing the file, soundfiler outputs the size of the file in samples which is then sent via `s samplelength $2` to the **'TrackPlayback'** abstraction in the **'Playback'** abstraction of the **'Transport'** section. The great thing about using abstractions in this patch is that there is no need to create separate sends for each track. By using the '$2' variable, the send takes it's name from the initialiser of the abstraction. The example in this case would be 's samplelength Kick'. This reduces amount of sends needed to route the data and creates a modular environment.



Triggers Section

The **'Triggers'** section is almost identical to the **'Input'** section, the difference being it loads the audio files that will act as replacement sounds. There is only a **'Load'** button in this section as there is no need to mute or record enable the triggers. If the user mutes any of the tracks that are triggering the samples in this section then those samples will not play.

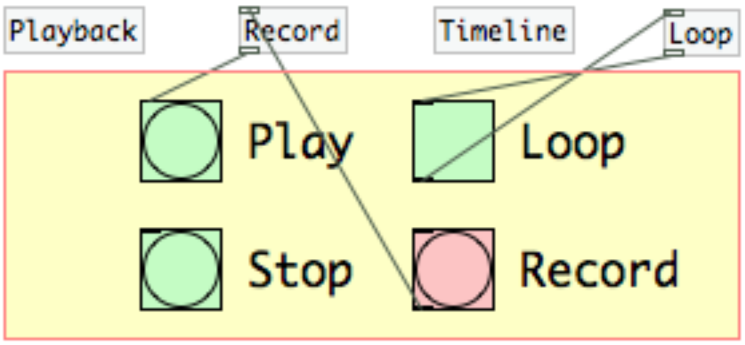


```
loadbang
read -resize ../DrumTrigger/Samples/Kick_Trigger.aif KickTrigger
InputTrigger $1 KickTrigger

loadbang
read -resize ../DrumTrigger/Samples/Snare_Trigger.aif SnareTrigger
InputTrigger $1 SnareTrigger

loadbang
read -resize ../DrumTrigger/Samples/HiHat_Trigger.aif HiHatTrigger
InputTrigger $1 HiHatTrigger
```

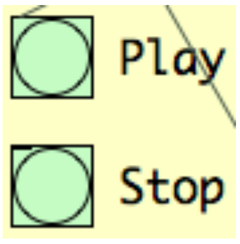
Another difference in this section is the inclusion of preset samples. Upon start-up `loadbang` bangs a message box containing the location of some samples that can act as triggers. This means the user can instantly start replacement of their input files. However, if they choose to use their own samples, the process is exactly the same as it is for the **'Input'** section. Clicking on **'Load'** in this section brings up a prompt to locate the sample of choice.

Transport Section

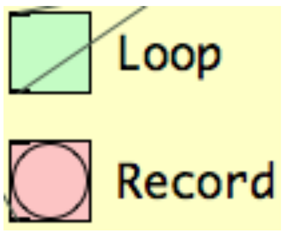
This section focuses on playback of the audio files and the recording of the triggered samples. It includes four function buttons and four abstractions.



The **'Play'** button starts playback from the beginning of the tracks and stops after one pass. It has a send-symbol called *play* that is sent to the **'Playback'** and **'Timeline'** abstractions. It also includes a receive-symbol called *loopplay* from the **'Loop'** abstraction.



'Stop' halts any playback happening and brings the timeline indicator to the beginning.

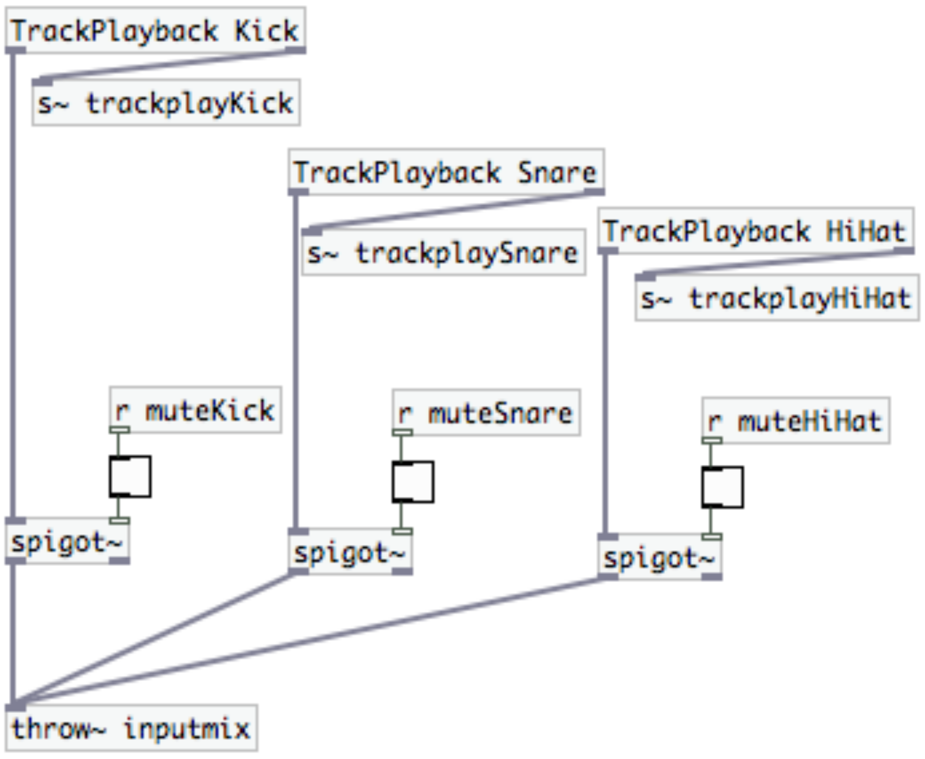


The **'Loop'** toggle creates an infinite loop that is only interrupted by either turning the toggle off or by clicking **'Stop'**. It has sends and receives to the **'Loop '** abstraction. The **'Record'** button allows the user to record the replacement sounds to disk.

Playback Abstraction

Playback

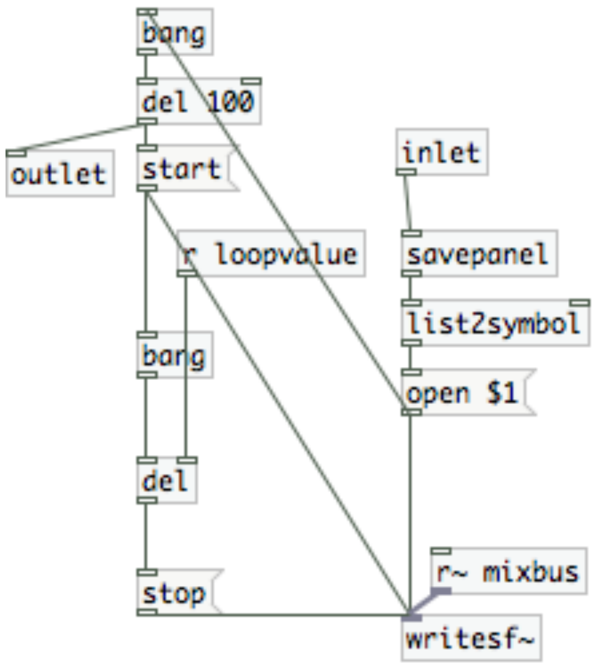
The **‘Playback’** abstraction enables playback of the material held in arrays. The actual method of playback is actually realised in the **‘TrackPlayback’** abstraction. This abstraction merely routes the signal to the input mix bus in the **‘Mix’** abstraction and receives the mute function from the **‘Input ‘** section and cuts the signal of the relevant track.



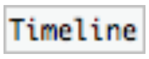
Record Abstraction

Record

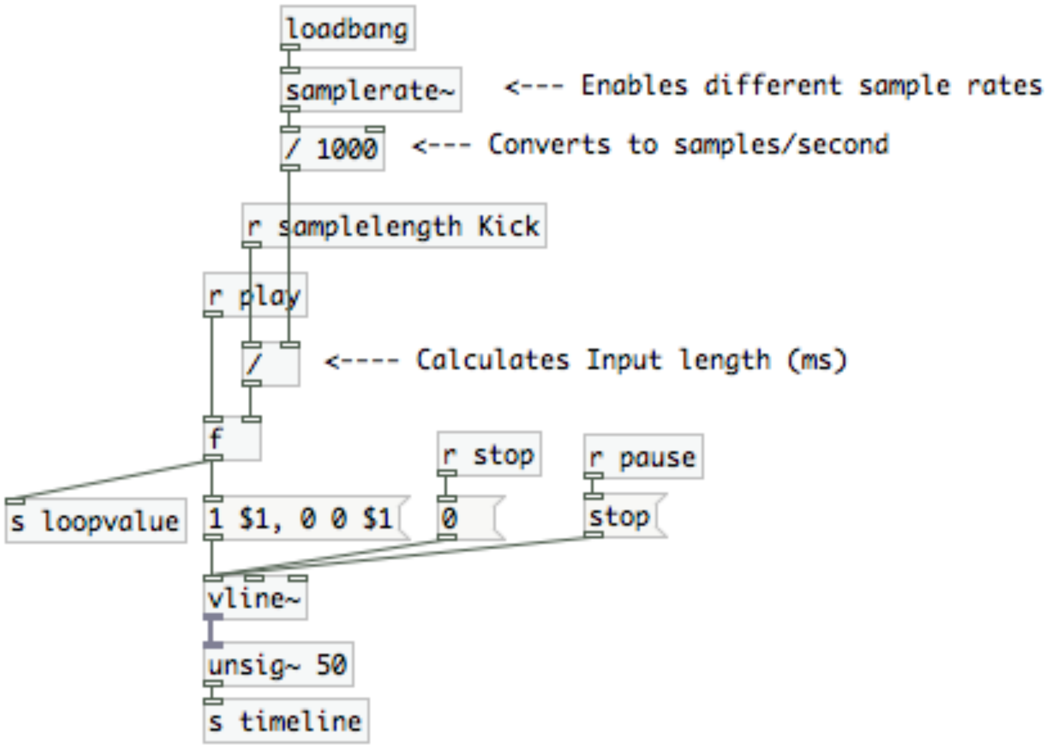
The **‘Record’** abstraction does the necessary routing to achieve bounces of the replacements. It’s inlet is a bang that when clicked prompts the user to select where they want to save the file and the name of the bounce. It also receives the output of the mix bus upon playback. Once the user has finished with the prompt the abstraction waits 100 ms and then tells to start recording while concurrently starting writesf~ playback. It then waits for a period set by the loop value before stopping the recording.



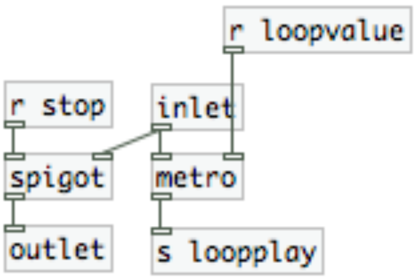
Timeline Abstraction



The **'Timeline'** abstraction enables to the timeline cursor in the main window to follow audio playback. It works by calculating how long the input sample is in milliseconds. This information is sent to a vline~ object that counts from 0 to 1 in the time calculated. By using the samplerate~ object it is possible to take different sample rates into account. The **'Play'** and **'Stop'** buttons from the **'Transport'** section stop and start the vline~ object. The sample length (ms) is additionally sent to the loopvalue receiver in the **'Loop'** abstraction.

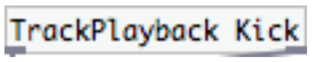


Loop Abstraction

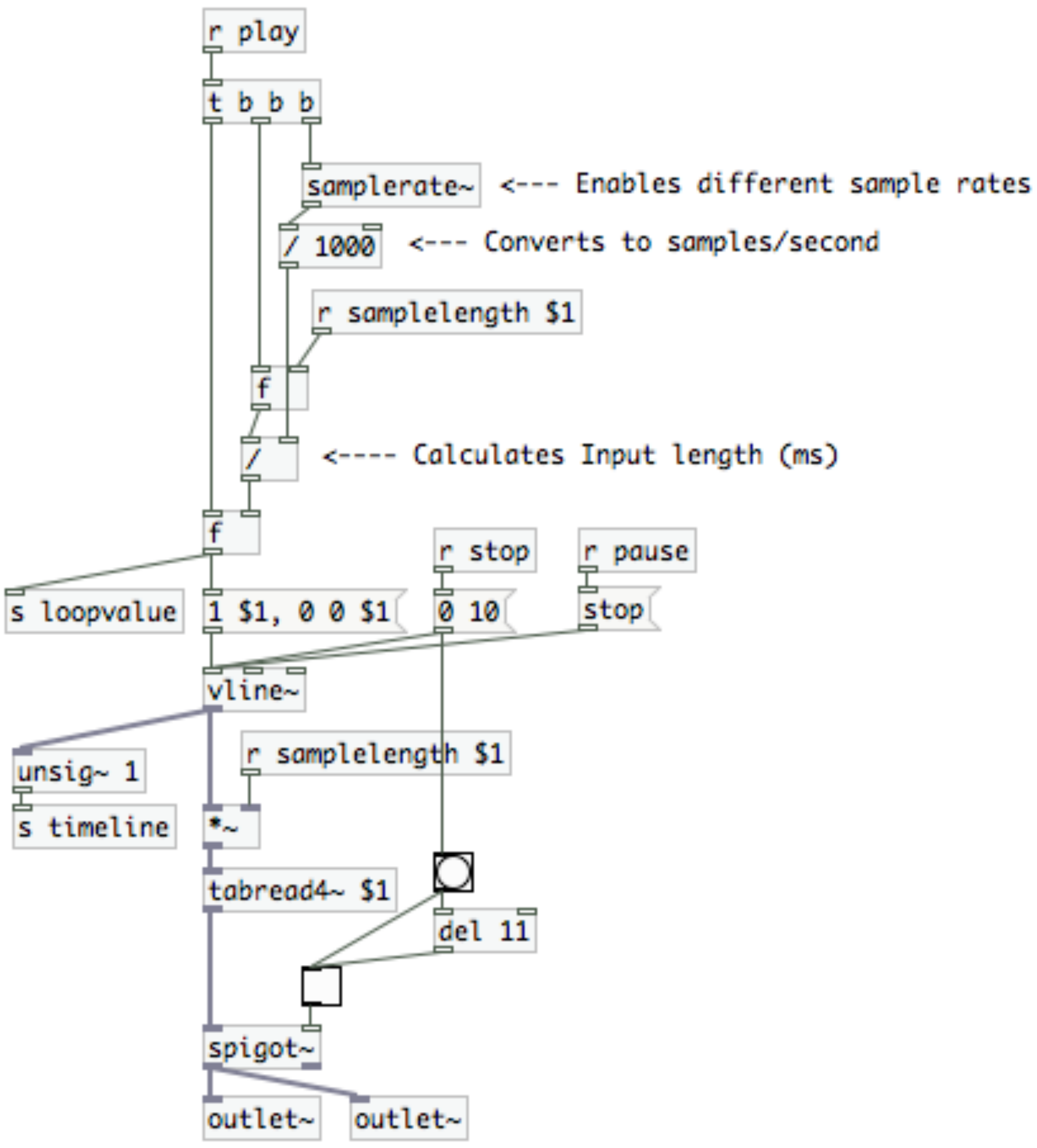


The **'Loop'** abstraction enables the looping function ability. The toggle in the **'Transport'** section starts a metronome that ticks away at the same speed as the length of the input audio. This means that at the exact point the playback finishes it is restarted by a bang from the metronome.

TrackPlayback Abstraction

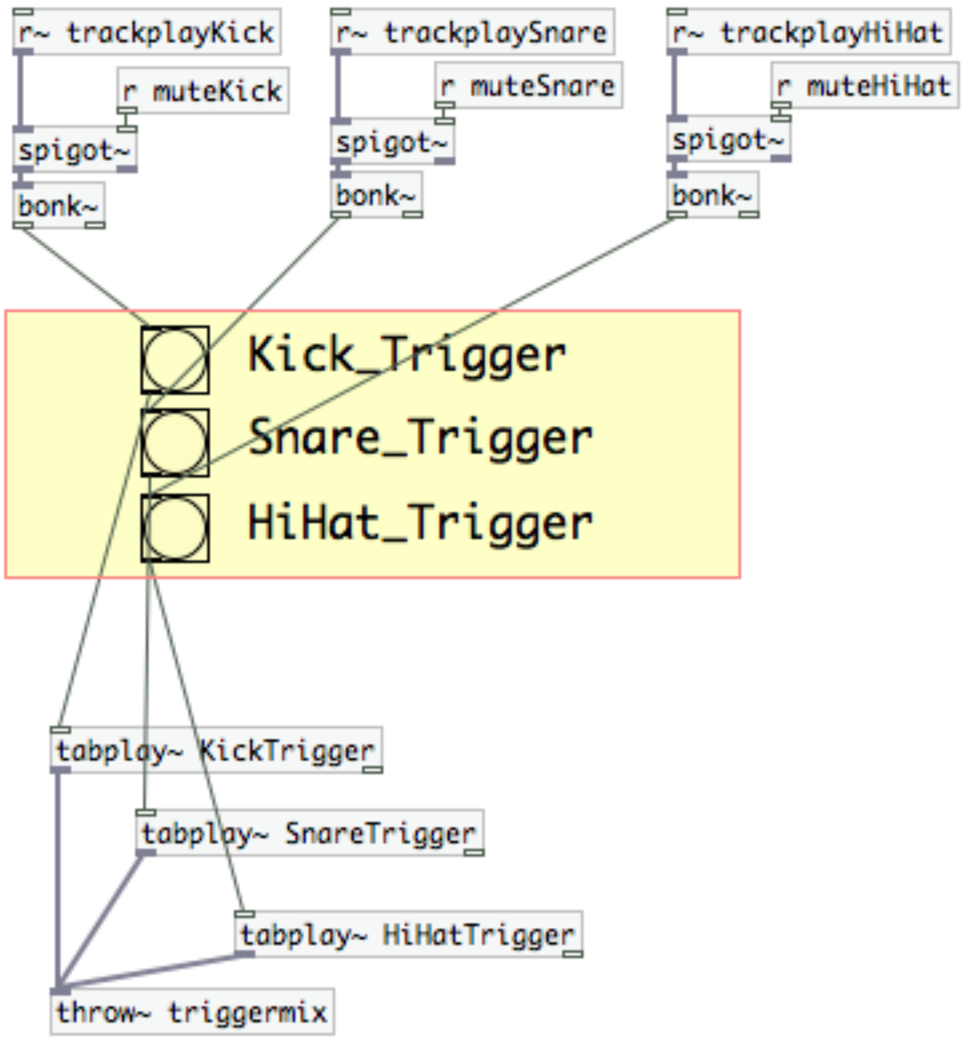


This abstraction is core of the playback function. It works in the same way as the **'Timeline'** abstraction with a vline~ object that is set by the sample length of the input file. The **'Play'** button in the '**Transport'** section bangs the inlet of this abstraction. The main difference in this abstraction is that it plays back the arrays of the input files instead of the timeline. This is achieved through the use of the tabread4~ object. The effect of multiplying the vline~ (running from 0 to 1) by the sample length gives an index for the tabread4~ to output the audio. I also added a spigot~ object to output. This is because when playback was stopped a click could be heard as the counter returns to zero. Therefore I muted the output of the tabread4~ object for the length of time it took to return to zero.



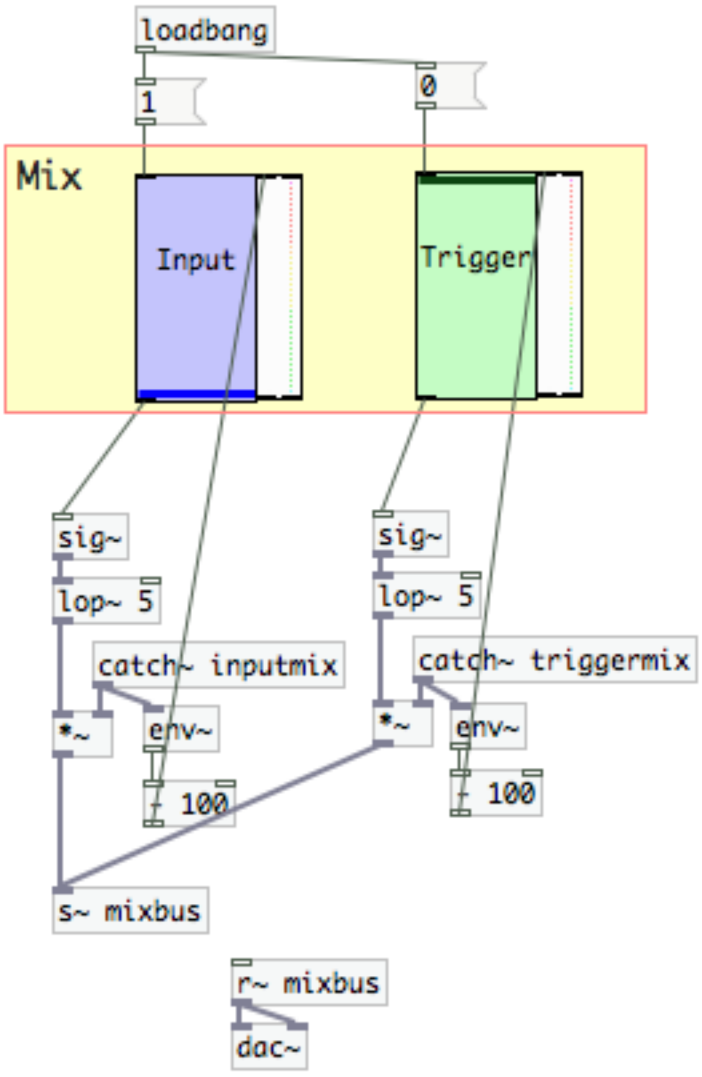
Replacement Section

While being fairly simple this section is the core of the trigger function. Each bonk~ object is fed a send from **'Trackplayback'** abstraction in the **'Playback'** abstraction of the **'Transport'** section. Essentially as soon as bonk~ detects there is a *hit* in the audio it triggers a bang in it's output. This in turn triggers a one-hit playback of the samples stored in the **'Triggers'** section. The mute button from the **'Input'** section also makes another appearance here. The output of this section is sent to a mix bus for the triggers in the **'Mix'** section.



Mix Section

This is the final section in the Sample Replacer project. The **'Mix'** section basically gathers all the audio output in the application and sums them together before going to the dac~. However, to add more functionality for the user I have included a mix slider to fade in either the input mix or the trigger mix (To avoid clicks when using the slider a low pass filter has been implemented to the output). A case where this is useful could be if the user wishes to hear how the triggers are affecting the original audio also, if recording, the user can add varying amounts of either mix to the bounce. A VU meter has been included to give a visual representation of the signal.



Considerations

I think the Sample Replacer works very efficiently and achieves its purpose. However, through the task of documenting the project I have come across a number of possible bugs that might want to be dealt with in the future. When pressing the **'Play'** button while in playback the audio plays at half the play speed. If pressed again it would halve that speed and so on. This is because the vline~ is being told to count at the sample rate again. However as it is not starting from the beginning the amount of samples are being miscalculated. I have also noticed a bug that causes the timeline cursor to twitch. This is possibly caused by the fact that the timeline is being calculated in multiple places. To amend this the timeline should have its own centralised function.

Some further modifications to the project in future would also increase the usefulness. At the moment the application only really deals with single tracks. Drum replacers are used in the majority to correct multi-tracked drum recordings. Therefore an ability to input and process multiple tracks would be imperative. Added functionality would also include velocity layers to the triggered samples and a clearer GUI for the **'Mix'** section.

Furthermore, the real intention of this project was to achieve beat detection of audio through analysis of data. However, time issues meant that what was a possibility now looked unlikely. Therefore as a last minute decision I decided to use the bonk~ object for audio analysis and concentrate on drum replacement.