Soft050 Synth Report

10564011

12/05/2017

Contents

1 Background

The program is an audio synthesiser which will be designed to run on Linux operating systems. There is a lack of good audio software availible for Linux, however there is a definite demand for it. The program will provide a configurable musical instrument for the user.

2 Method

Haskell with the stack eco-system has been used as the programming language and environment. Functional Reactive Programming techniques with the Reactive Banana libraries have been leveraged in order to provide GUI and MIDI event handling as well as managing the state of the instrument. Reactive Banana was chosen over Wires as it does not require being the "Main Loop" of the program, instead reacting to calls from other functions.

The GUI has been built with Glade, which is a graphical GUI design frontend for GTK. Glade produces an XML file which the program reads to create the GUI and define events and behaviors relating to GUI elements.

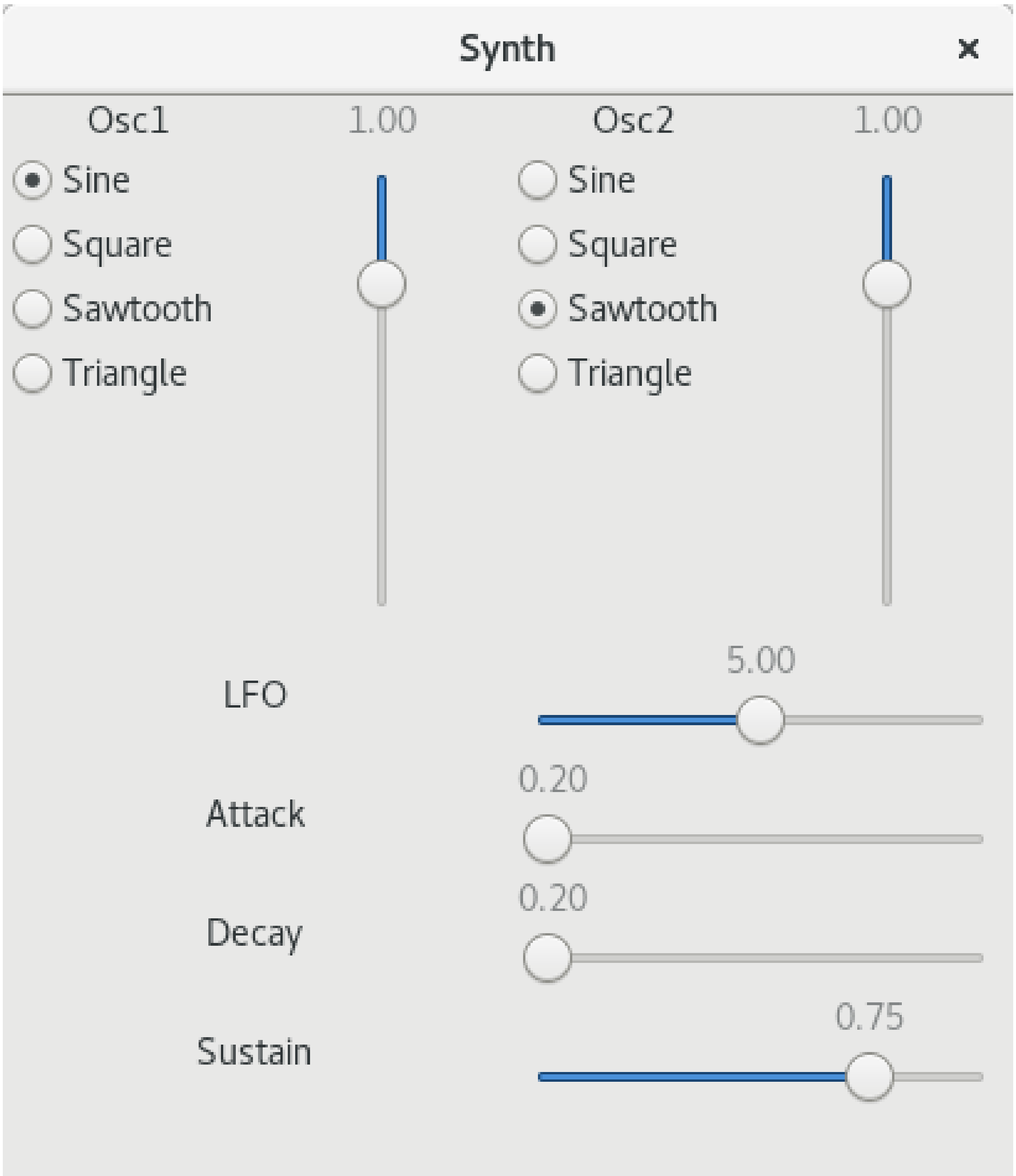
The midi-simple library was used to convert raw MIDI messages from JACK into a more sensible data structure for use within the program, this aids with pattern matching MIDI events.

3 Analysis

This program provides a musical instrument with configurable sound qualities which interfaces with JACK for MIDI input and audio output. It aims to provide a low latency response to MIDI input and a simple to use GUI.

4 Design

4.1 User Interface



Osc1 1.00
● Sine
○ Square
○ Sawtooth
○ Triangle

Osc2 1.00
○ Sine
○ Square
● Sawtooth
○ Triangle

LFO 5.00

Attack 0.20

Decay 0.20

Sustain 0.75

4.2 Code Structure

The program is split into 5 modules; the Main module which contains the main function (Called on program execution); the JACK module, which has the necessary interface to communicate with JACK; the MIDI module, which provides the helper functions for MIDI event processing; the GUI module, which sets up the event network for the GUI; the Wave module, which defines the functions needed to create the waveforms fed into JACK's output and the Event module in which the main logic behind the event network is defined.

5 Implementation

5.1 foldWaves

The foldWaves function is an integral part of the program, as it takes the IntMap containing all the Frequencies, Times and Velocities of the current notes being held down, and folds them with the makeWave function to produce a single soundwave which is presented to JACK.

```haskell
foldWaves :: Fractional a =>
             Behavior (IntMap.IntMap (Double, Double, Word64)) ->
             Behavior ((Double, Double, Word64) -> Wave a) ->
             Behavior (Wave a)
foldWaves notes wave = (fmap (\f -> IntMap.foldr (\freq a -> f freq ^+^ a) zero) wave) <*> notes
```

The function takes the IntMap and a partially applied makeWave function already containing the GUI settings as parameters.
The first part of the function is a lambda being parsed as the first argument to the foldr function. The lambda takes a single note from the IntMap and applies it to the wave function, then it is added to the other notes applied via a pointwise addition from the Linear.Vector library.
This fold is embedded inside another lambda function which takes the wave function as an argument. It is fmapped with the wave function since it is in the Behavior functor, and the folding function is not.
Lastly the fold is applied to the IntMap containing the notes with ⟨*⟩ from Control.Applicative resulting in a single wave function.

5.2 notesB

notesB is a Behavior created from the accumulation of MIDI events. It combines the midiNotesE event, which filters just midi note events from midiE, and the runNotes function which either inserts or deletes notes from the IntMap depending on whether a key is pressed of released.

notesB:

```
    notesB          <- accumB notes $ fmap (\(m, t) -> MIDI.
        runNotes m t) midiNotesE
```

isNote:

```
isNote :: MidiMessage -> Bool
isNote (ChannelVoice NoteOn{}) = True
isNote (ChannelVoice NoteOff{}) = True
isNote _ = False
```

runNotes:

```
runNotes :: Integral a =>
            MidiMessage ->
            a ->
            IntMap.IntMap (Double, Double, a) ->
            IntMap.IntMap (Double, Double, a)
runNotes (ChannelVoice (NoteOn _ p v)) t notes =
    IntMap.insert (fromIntegral (getPitch p))
                  (getFrequency (getPitch p), scaleVelocity (
                      getVelocity v), t)
                  notes
runNotes (ChannelVoice (NoteOff _ p _)) _ notes =
    IntMap.delete (fromIntegral (getPitch p)) notes
runNotes _ _ _ = error "Internal␣Error:␣runNotes"
```

The isNote function takes a MidiMessage from Sound.MIDI and returns True if it is either a NoteOn or NoteOff.
The runNotes function takes a MidiMessage filtered by isNote, the velocity of the note and a tick value of the time the MIDI Event occurred and an IntMap in the state before the function was run.
It checks to see if the MidiMessage is either NoteOn or NoteOff, then either inserts the relevant data into the IntMap, or deletes a record pertaining to that note. notesB takes a Behavior accumulated by accumB from Reactive.Banana. accumB takes the empty IntMap as a starting value and the Event created by applied fmap to a lambda function containing runNotes and midiNotesE containing the MidiMessage, velocity and time.

6 Conclusion

6.1 Progess

6 main features were planned, out of those The Soundwave Generator, Jack Interface, Soundwave Effects and Graphical User Interface were implemented, whereas the Sqlite Database and MIDI file playing ability were omitted.

The Sqlite Database was omitted due to time restrictions. The developer didn't feel that the time spent implementing the Sqlite Database would be as valuable as time spent developing other main features of the program.

The Midi file playing ability was omitted as the developer felt that this feature went outside of the scope of the program, since the program will accept any MIDI input from JACK, not just that from a keyboard.

6.2 Review

Over the course of this project several new technologies have been learnt such as Functional Reactive Programming and GTK. The developer has also greatly expanded their knowledge about applicative functors in Haskell.
If more time was available the developer would implement features such as a sustain envelope, preset database and a reverb effect. The developer would also implement the ability for mapping GUI Controls to MIDI Controls. The project has been a success.

7 Appendix A: Functional Specification

Functionality	Included	Priority	Notes
Sine Wave Function	Yes	Must Have	
Square Wave Function	Yes	Should Have	
Sawtooth Wave Function	Yes	Should Have	
Triangle Wave Function	Yes	Should Have	
MIDI Input	Yes	Should Have	Implemented using JACK
Audio Output	Yes	Must Have	Implemented using JACK
Low Latency MIDI Response	Yes	Should Have	
Low Frequency Oscillator	Yes	Should Have	
Attack Decay Sustain Envelope	Yes	Could Have	
Sustain Envelope	No	Could Have	Ommited due to time restraints
Graphical User Interface	Yes	Must Have	Implemented using GTK
Radio Button Oscillator Selectors	Yes	Should Have	
Oscillator Amplitude Scale	Yes	Should Have	
LFO Frequency Scale	Yes	Should Have	
ADR Selector Scale	Yes	Could Have	
Sustain Time Scale	No	Could Have	Sustain not implemented
Preset Database	No	Should Have	Omitted due to time restraints

8 Appendix B: Test Plan

Test	Result	Pass/Fail
Press key on midi input	Sound output	Pass
Vary velocity of key press	Sound amplitude changes	Pass
Change note being pressed	Frequency of sound changes	Pass
Press multiple notes at once	Multiple frequencies of sound	Pass
Change oscillators	Timbre of sound changes	Pass
Change oscillator amplitude	Volume of sonud rises	Pass
Change LFO frequency	Timbre of sound changed	Pass
Change attack	Note takes longer/shorter to reach full volume	Pass
Change decay	Note takes loner/shorter to reach sustain volume	Pass
Change sustain	Sustain volume is quieter / louder	Pass
Close GUI	GUI closes but program does not	Partial Pass

9 Appendix C: Full Code

Main.hs

```haskell
module Main where

import qualified Synth.Event as Event
import qualified Synth.JACK as JACK

import Control.Concurrent
import Control.Monad.Trans.Class (lift)
import Reactive.Banana
import Reactive.Banana.Frameworks
import Sound.JACK
import Data.IORef
import Data.Word

import qualified Data.IntMap.Strict as IntMap

import qualified GI.Gtk as Gtk


-- Setup GUI, Event Network and JACK
main :: IO ()
main = do
    _ <- Gtk.init Nothing

    handleExceptions $ do
        client  <- newClientDefault "Haskell-Synth"
        input   <- newPort client "input"
        output  <- newPort client "output"
        rate    <- lift $ getSampleRate client

        (midiEvent, runMidiEvent) <- lift $ newAddHandler
        wave <- lift $ newIORef (pure 0)

        ticks <- lift $ newIORef 0

        let notes = IntMap.empty :: IntMap.IntMap (Double, Double, Word64)

        lift $ actuate =<< compile (Event.network midiEvent notes wave rate)
        withProcess client
            (JACK.process input output ticks wave runMidiEvent) $ do
                _ <- lift (forkOS Gtk.main)
                activate client
                lift waitForBreak
                deactivate client
```

Synth/JACK.hs

```haskell
module Synth.JACK ( process ) where
```

```

import qualified Synth.Wave as Wave

import Control.Monad
import Control.Monad.Trans.Class (lift)
import Control.Monad.Exception.Synchronous (ExceptionalT)
import Reactive.Banana.Frameworks
import Sound.JACK
import Sound.JACK.Audio (Sample, getBufferArray)
import Sound.JACK.Exception
import Sound.MIDI
import Data.Array.Storable
import Data.Either
import Data.IORef
import Data.Word

import qualified Sound.JACK.Audio as JAudio
import qualified Sound.JACK.MIDI as JMIDI

-- Jack process function, handles reading and writing data from JACK
process :: ThrowsErrno e =>
           JMIDI.Port Input ->
           JAudio.Port Output ->
           IORef Word64 ->
           IORef (Wave.Wave Sample) ->
           Handler (MidiMessage, Word64) ->
           NFrames -> ExceptionalT e IO ()
process input output ticks sound runEvent nf@(NFrames n) = do
    oarr <- lift $ getBufferArray output nf
    tstart <- lift $ readIORef ticks

    midiEvs <- fmap (decodeMidi1 . JMIDI.rawEventBuffer)
            <$> JMIDI.readRawEventsFromPort input nf

    lift $ mapM_ runEvent (map (\x -> (x, tstart)) (rights midiEvs))

    forM_ (nframesIndices nf) $ \i@(NFrames t) -> do
        wave <- lift $ readIORef sound
        let s = Wave.runWave wave (tstart + fromIntegral t)
        lift $ writeArray oarr i s
    lift $ modifyIORef' ticks (+ fromIntegral n)
```

Synth/MIDI.hs

```
module Synth.MIDI ( isNote
                  , runNotes ) where

import Sound.MIDI

import qualified Data.IntMap.Strict as IntMap

-- Return True if MIDI Event is a keypress
```

```haskell
isNote :: MidiMessage -> Bool
isNote (ChannelVoice NoteOn{}) = True
isNote (ChannelVoice NoteOff{}) = True
isNote _ = False

-- Convert from a MIDI pitch to a frequency
getFrequency :: Integral a => a -> Double
getFrequency n = 2 ** (((fromIntegral n) - 69) / 12) * 440

-- Convert velocity from Integral to Amplitude
scaleVelocity :: Integral a => a -> Double
scaleVelocity v = fromIntegral v / 127

-- Take MIDI notes and insert/delete them into an IntMap
runNotes :: Integral a =>
            MidiMessage ->
            a ->
            IntMap.IntMap (Double, Double, a) ->
            IntMap.IntMap (Double, Double, a)
runNotes (ChannelVoice (NoteOn _ p v)) t notes =
    IntMap.insert (fromIntegral (getPitch p))
                  (getFrequency (getPitch p), scaleVelocity (getVelocity v), t)
                  notes
runNotes (ChannelVoice (NoteOff _ p _)) _ notes =
    IntMap.delete (fromIntegral (getPitch p)) notes
runNotes _ _ _ = error "Internal Error: runNotes"
```

Synth/Wave.hs

```haskell
{-# LANGUAGE MultiWayIf #-}

module Synth.Wave ( Wave (..)
                  , foldWaves
                  , makeWave
                  , sine
                  , square
                  , sawtooth
                  , triangle ) where

import Data.Word
import Linear.Vector
import Reactive.Banana

import qualified Data.IntMap.Strict as IntMap

-- Define a type for Waves
newtype Wave a = Wave { runWave :: Word64 -> a }

-- Required instances for new type
instance Functor Wave where
    fmap f (Wave k) = Wave $ f . k

instance Applicative Wave where
```

```haskell
    pure x = Wave (const x)
    f <*> x = Wave $ \t -> runWave f t (runWave x t)

instance Additive Wave where
    zero = pure 0

-- Convert Ticks to Seconds
sampled :: Fractional a => Int -> Word64 -> (Double -> a) -> Wave a
sampled r t f = Wave $ \s -> f (fromIntegral (s - t)  / fromIntegral r)

-- Wave Functions
sine :: Fractional a => Int -> Word64 -> Double -> Wave a
sine r t f = sampled r t $ \x -> realToFrac (sin (2 * pi * f * x))

square :: Fractional a => Int -> Word64 -> Double -> Wave a
square r t = fmap signum . (sine r t)

sawtooth :: Fractional a => Int -> Word64 -> Double -> Wave a
sawtooth r t f = sampled r t $ \x -> realToFrac (2 * ((x * f) - fromIntegral (floor (0.5 + (x * f)) :: Int)))

triangle :: Fractional a => Int -> Word64 -> Double -> Wave a
triangle r t = fmap (subtract 1 . abs . (* 2)) . (sawtooth r t)

-- Fold multiple waves into a single wave
foldWaves :: Fractional a =>
             Behavior (IntMap.IntMap (Double, Double, Word64)) ->
             Behavior ((Double, Double, Word64) -> Wave a) ->
             Behavior (Wave a)
foldWaves notes wave = (fmap (\f -> IntMap.foldr (\freq a -> f freq ^+^ a) zero) wave) <*> notes

-- Attack Decay Sustain envelope
ads :: Fractional a => Int -> Word64 -> Double -> Double -> a -> Wave a
ads r t a d s = sampled r t $ \x -> if | x <= a -> realToFrac (x / a)
                                       | a < x && x <= (d + a) ->
                                        let x' = realToFrac (x - a)
                                         in (((s - 1) * x') / realToFrac d) + 1
                                       | otherwise -> s

-- Vector Multiplication
(^*^) :: Fractional a => Wave a -> Wave a -> Wave a
(^*^) = liftA2 (*)

-- Combine Wave functions to produce desired sound
```

```haskell
makeWave :: Fractional a =>
            Int ->
            Double ->
            a ->
            a ->
            Double ->
            Double ->
            a ->
            (Int -> Word64 -> Double -> Wave a) ->
            (Int -> Word64 -> Double -> Wave a) ->
            (Double, Double, Word64) -> Wave a
makeWave r lfo oscAmp1 oscAmp2 a d s osc1 osc2 (freq, v, t) =
    (realToFrac v) *^ ((ads r t a d s) ^*^
    (0.25 *^ sine r t lfo) ^*^
    (0.5 *^ (((0.1 * oscAmp1) *^ osc1 r t freq) ^+^ ((0.1 * oscAmp2) *^ osc2 r t freq))))
```

Synth/GUI.hs

```haskell
{-# LANGUAGE OverloadedLabels #-}
{-# LANGUAGE OverloadedStrings #-}

module Synth.GUI ( GUIEvs (..)
                 , Oscillator (..)
                 , guiNetwork ) where

import Reactive.Banana
import Reactive.Banana.Frameworks
import Reactive.Banana.GI.Gtk

import qualified GI.Gtk as Gtk

-- Data structure which stores GUI events
data GUIEvs = GUIEvs
    { osc1E :: Event Oscillator
    , osc2E :: Event Oscillator
    , oscAmp1E :: Event Double
    , oscAmp2E :: Event Double
    , lfoE :: Event Double
    , attackE :: Event Double
    , decayE :: Event Double
    , sustainE :: Event Double
    }

-- Data structure which stores Oscillator wave types
data Oscillator = Sine | Square | Triangle | Saw deriving (Show, Eq)

-- Helper function which combines a list of Events into a single Event
leftUnion :: [Event a] -> Event a
leftUnion xs = foldl (unionWith (\a _ -> a)) never xs

-- Converts 4 RadioButton bool values into a single Oscillator
```

```
selectOsc4 :: Gtk.RadioButton ->
              Gtk.RadioButton ->
              Gtk.RadioButton ->
              Gtk.RadioButton -> MomentIO (Event Oscillator)
selectOsc4 rSine rSquare rSaw rTriangle = do
    rSineE <- (Sine <$) . filterE (==True) <$> (attrE rSine #active)
    rSquareE <- (Square <$) . filterE (==True) <$> (attrE rSquare #active)
    rSawE <- (Saw <$) . filterE (==True) <$> (attrE rSaw #active)
    rTriangleE <- (Triangle <$) . filterE (==True) <$> (attrE rTriangle #active)
    pure $ leftUnion [rSineE, rSquareE, rSawE, rTriangleE]

-- Sets up the GUI Events
guiNetwork :: MomentIO GUIEvs
guiNetwork = do
    b               <- Gtk.builderNew
    _               <- Gtk.builderAddFromFile b "gui.glade"

    w               <- castB b "window" Gtk.Window
    destroyE        <- signalE0 w #destroy

    reactimate $ Gtk.mainQuit <$ destroyE

    oscSine1        <- castB b "oscSine1" Gtk.RadioButton
    oscSquare1      <- castB b "oscSquare1" Gtk.RadioButton
    oscSaw1         <- castB b "oscSaw1" Gtk.RadioButton
    oscTriangle1    <- castB b "oscTriangle1" Gtk.RadioButton

    osc1Ev          <- selectOsc4 oscSine1 oscSquare1 oscSaw1 oscTriangle1

    oscSine2        <- castB b "oscSine2" Gtk.RadioButton
    oscSquare2      <- castB b "oscSquare2" Gtk.RadioButton
    oscSaw2         <- castB b "oscSaw2" Gtk.RadioButton
    oscTriangle2    <- castB b "oscTriangle2" Gtk.RadioButton

    osc2Ev          <- selectOsc4 oscSine2 oscSquare2 oscSaw2 oscTriangle2

    oscAmp1         <- castB b "oscAmpAdj1" Gtk.Adjustment
    oscAmp1Ev       <- attrE oscAmp1 #value

    oscAmp2         <- castB b "oscAmpAdj2" Gtk.Adjustment
    oscAmp2Ev       <- attrE oscAmp2 #value

    lfo             <- castB b "lfoAdj" Gtk.Adjustment
    lfoEv           <- attrE lfo #value

    attack          <- castB b "attackAdj" Gtk.Adjustment
    attackEv        <- attrE attack #value

    decay           <- castB b "decayAdj" Gtk.Adjustment
    decayEv         <- attrE decay #value

    sustain         <- castB b "sustainAdj" Gtk.Adjustment
    sustainEv       <- attrE sustain #value

    #showAll w

    pure $ GUIEvs osc1Ev osc2Ev oscAmp1Ev oscAmp2Ev lfoEv attackEv decayEv sustainEv
```

Synth/Event.hs

```
{-# LANGUAGE RecordWildCards #-}

module Synth.Event ( network ) where

import Data.Word
import Data.IORef
import Reactive.Banana
import Reactive.Banana.Frameworks
import Sound.MIDI

import qualified Data.IntMap.Strict as IntMap

import qualified Synth.MIDI as MIDI
import qualified Synth.Wave as Wave
import qualified Synth.GUI as GUI

-- Convert from the GUI.Oscillator type to a Wave function
oscToWave :: Fractional a =>
             GUI.Oscillator -> (Int -> Word64 -> Double -> Wave.Wave a)
oscToWave GUI.Sine = Wave.sine
oscToWave GUI.Square = Wave.square
oscToWave GUI.Saw = Wave.sawtooth
oscToWave GUI.Triangle = Wave.triangle

-- Combine a list of events into a single event for reactimate
anyE :: [Event a] -> Event ()
anyE = foldr (unionWith (\_ _ -> ()) . (() <$)) never

network :: Fractional a =>
           AddHandler (MidiMessage, Word64) ->
           IntMap.IntMap (Double, Double, Word64) ->
           IORef (Wave.Wave a) ->
           Int -> MomentIO ()
network midiH notes ref rate = do

    GUI.GUIEvs{..}  <- GUI.guiNetwork

    midiE           <- fromAddHandler midiH
    let midiNotesE = filterE (\(x, _) -> MIDI.isNote x) midiE

    -- Accumilate all Midi note events into a single Behavior
    notesB          <- accumB notes $ fmap (\(m, t) -> MIDI.runNotes m t) midiNotesE

    -- Create Behaviours from GUI Events
    lfoV            <- stepper 5 lfoE
    oscAmp1V        <- stepper 1 oscAmp1E
    oscAmp2V        <- stepper 1 oscAmp2E
    attackV         <- stepper 0.2 attackE
    decayV          <- stepper 0.2 decayE
    sustainV        <- stepper 0.75 sustainE
    osc1V           <- stepper GUI.Sine osc1E
    osc2V           <- stepper GUI.Sine osc2E


    -- Fold all notes into a wave for JACk
    let wave :: Fractional a =>
                Behavior ((Double, Double, Word64) -> Wave.Wave a)
        wave = Wave.makeWave <$> (pure rate)
                             <*> (realToFrac <$> lfoV)
                             <*> (realToFrac <$> oscAmp1V)
                             <*> (realToFrac <$> oscAmp2V)
                             <*> (realToFrac <$> attackV)
                             <*> (realToFrac <$> decayV)
                             <*> (realToFrac <$> sustainV)
                             <*> (oscToWave <$> osc1V)
                             <*> (oscToWave <$> osc2V)

        foldWavesV = Wave.foldWaves notesB wave

    -- Re-Fold Waves when a relevant Event occurs
    reactimate $ fmap (writeIORef ref) (foldWavesV <@ anyE [ 0 <$ osc1E
                                                           , 0 <$ osc2E
                                                           , 0 <$ midiE
                                                           , lfoE
                                                           , attackE
                                                           , decayE
                                                           , sustainE
```

```
                                                                    ,
                                                                        oscAmp1E
                                                                    ,
                                                                        oscAmp2E
                                                                    ])
```