**Shearing Force Phase Ripper**

Specifications Document

1.

The *Shearing Force Phase Ripper* is a pitch-dependent, harmonic-matching phase-flipper. It is intended to create giant, shrieking, noise monsters that take up no more headroom than the original input audio. It is a distortion module, but is also intended to be played like a synthesizer, though it produces no audio of its own. It can be used in tandem with synthesizers, utilizing the same MIDI input sent to the audio source. It can also be used with monophonic input, and is able to track the pitch of the incoming audio, performing pitch calculations on its own. There are three main sections:

The MIDI Control Section contains:

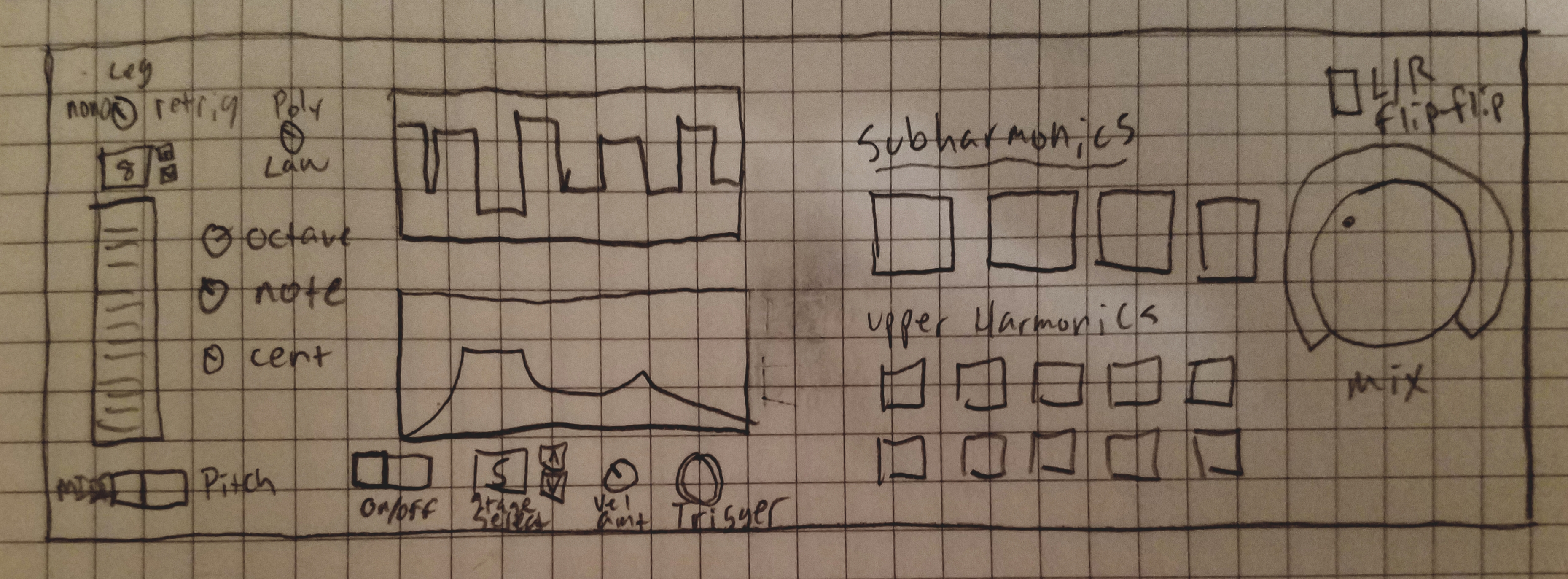
* + Input gain & VU Meter\*
  + Pitch bend wheel
  + Pitch bend range control
  + Polyphony control
  + Polyphony Law select
  + MIDI/pitch-tracking switch
  + Octave/note/cent controls
  + Glide speed control\*

The Display Section contains:

* + A graphical display that can switch between two (maybe 3) views:
    - Static image of what the “flips” currently look like (updates when harmonic is selected)
    - Dynamic image of the audio as it passes through, after flipping
    - Maybe we do a non-flipped view somehow too (integrate with mix knob?)
  + Graphical display where the user may edit an envelope generator
  + Stage-select (how many stages the envelope has)
  + Envelope on/off switch
  + Velocity to envelope amount knob
  + Envelope trigger button (so that the envelope may be used while in pitch-tracking mode)
  + Oscillator sync button\*

The Harmonics Control Section contains:

* + One row of slightly larger buttons for sub-harmonics (on/off/flip)
  + One or two rows of buttons for upper harmonics (on/off/flip)
  + Mix knob
  + L/R flip-flip button (inadvisable for mono signals, probably best utilized w/mix knob)
  + Output gain & VU meter\*



\*not pictured

Gameplan

2.

*Phase 1: Up & Running*

* Consider: VST2 backwards-compatibility, compatibility w/different bit- and sample-rates
* Investigate GUI MVC setup in sample projects
* Get a blank panel that will pass audio
* Input gain slider / input VU meter
* Output gain slider / output VU meter
* Ensure input/output sliders/meters are coupled properly

*Phase 2: MIDI Handling*

* Add all MIDI section controls, event handling infrastructure as needed
* Get mono legato MIDI working
* Create small temp display showing the octave/letter/accidental/Hz of current note (no audio output). There should be enough room for multiple rows
* Get octave/note/cent working
* Get glide working
* Get Pitchbend wheel/range working
* Get legato/retrig/poly working
* Get polyphony modification working (think about poly law handling)

*Phase 3: Shred It*

* Design implementation—consider zero-crossings? Investigate ‘Factory’ design style
* Add all Harmonics buttons
* Implement harmonic flips
* Mix knob, L/R flip-flip
* DC Offset filter/autogain?
* Osc. sync button
* Testing around how many and which harmonics buttons we want to use
* Design and implement Flips Display

*Phase 4: Pitch Detect*

* Research and design pitch detect circuits
* Pitch/MIDI switch
* Implement pitch detect circuit
* Glide knob to control speed—test what ranges/scales make sense

*Phase 5: Envelope*

* Add all Envelope section controls
* Design touch panel
* Get stage select/editor working
* On/off switch working
* Velocity —> env working
* Trigger working

*Phase 6: Finalization*

* Profile and optimize
  + - FACTORY: can be set up to construct as many harmonicFlips as can be needed based upon polyphony/harmonic button press, and can be called to pass those out to a note object, to avoid constructing/destructing them over and over again?
* Finalize and adjust GUI: Consult an expert in HCI or panel/graphics design?
* Finalize customer documentation
* Get some beta testers
  + Implement feedback/bug fixes
* DRM?
* Sell/market

3.

**Class: NoteBank**

Fields:

notes: Array holding all active FlipNotes (linkedlist better for polyphony? overall probably no)

polyphony: int saying how many notes are allowed

law: enumerated type? (polyStop - new notes can’t be added, polyDrop - oldest notes are

dropped to make room for new ones, monoLegato, monoRetrig)

**Class: FlipNote**

Fields:

flips: Array holding HarmonicFlippers

note: MIDI note

Methods:

*Constructor*:

* + - * Takes a MIDI note event, storing it
      * Creates a HarmonicFlipper[], the size of which is the max number of flippers possible (or the number of flippers currently active, depending if we want to dynamically resize this array)
      * Fills array with flippers initialized to the frequency of this note (calculated from MIDI note info & device’s pitch controls), and their appropriate harmonicLevel based upon position in the array
        + How do we interface with the pitch controls? *Global variables??* or do we store the info locally and update it whenever there’s a controller change?

*flip()*:

* + - * Takes a float (the current sample)
      * Iterates through flips[], calling flipper.flip(sample)
      * returns the sample

*noteChange()*:

* + - * Called whenever there is an update to the MIDI note information, or there is a change to the pitch controls of the device
      * Iterates through the flips, updating the fundamental

*getFlippers()*:

* + - * Returns shallow copy of flips array

**Class: HarmonicFlipper**

Fields:

fundamental: Hz of fundamental freq this note is on

harmonicLevel: number indicating which harmonic/subharmonic this is, should be FINAL

timer: float that holds how many samples have passed

polarity: switches between 1/-1

inversion: 1/-1 depending on whether this harmonic is inverted

active: boolean declaring whether this harmonic is active

Methods:

*Constructor*:

* + - * Takes initial values for fundamental and harmonicLevel
      * initialize polarity to 1, timer to 0

//This may cause an issue with interpolation error—may be fixed

*timerIncrement()*:

* + - * + called every sample
        + increment timer by 1
        + Check if timer is > timerVal()

polarity \*= -1

timer -= timerVal()

*flip()*

* + - * takes a sample value
      * if active, multiply by polarity and inversion
      * return the sample value

*timerVal()*

* + - * returns SMPL\_RTE / (fundamental \* harmonicLevel) ///Is this even right????

*activate()*

* + - * active = true
      * Alternatively to this and the next three methods, there could be one activate() method that flips between valid states with conditionals, depending on which would be more convenient for structure

*deactivate()*

* + - * active = false

*invert()*

* + - * inversion = -1

*uninvert()*

* + - * inversion = 1

JOSH:

Latency reporting/multithreading (<https://sdk.steinberg.net/viewtopic.php?t=209>)

Hashmapping functional decomp

JON:

Figure out actual latency implementation

Pitch tracking implementation