Execution

Introduction

The following chapter gives an outline of the process that was undertaken to build out the final application. A description of early prototype work is given to give context to the construction of the final working prototype version. This is followed by a description of the technical architecture of the system as well as an outline of the sometimes difficult setup process of getting the various architectural elements working together. A detailed discussion of the core functionality of the system is then given, followed by a description of functionality that is more secondary in nature.

Early prototype work

MelodyPainter

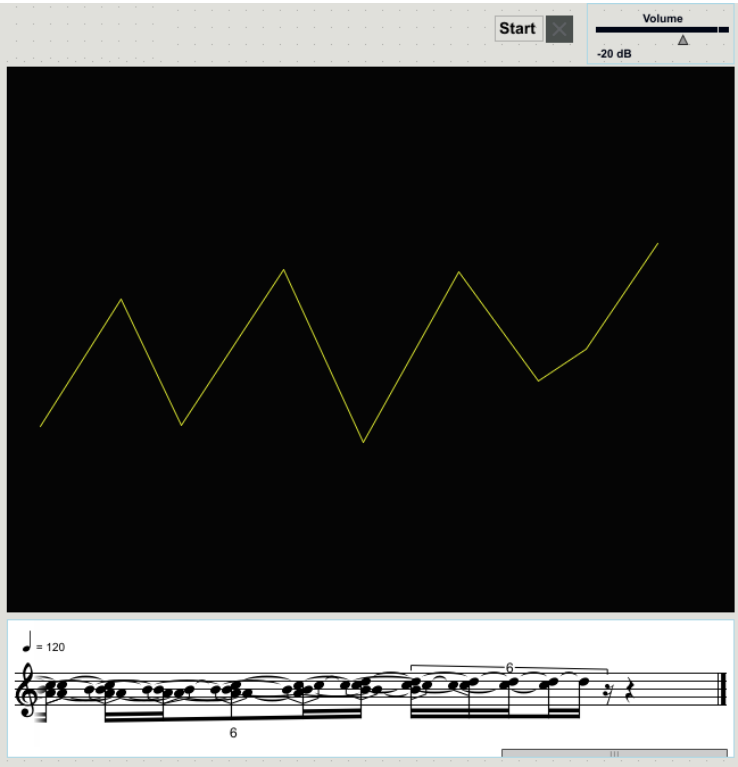


Figure 1: MelodySketch interface

MelodyPainter is an early protoype built out in Max MSP that allows users to draw freehand lines, which are converted into break point function data and used

to generate a melodic profiles using Bach for Max MSP (Agostini and Ghisi, 2015). Bach is a suite of composition tools that allow for a number of computer aided composition techniques (CAC) and provides similar functionality to IRCAM's Open Music system. These melodic profiles are then filtered to only includes notes from a pentatonic scale, to give reasonably pleasing aural results. Some notable flaws in the system include the following. It is limited to strictly western tonal music styles. It has no allowance for rhythm and plays only eight notes giving results a noticeably bland and predictable quality. The freeform nature of sketched input however was quite a pleasing means of inputting the control information.

SonicShaper

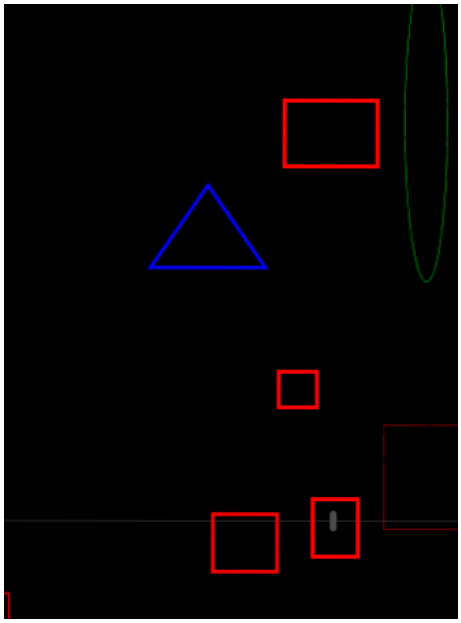


Figure 2: SonicShaper interface

A separate application was created in Processing which allowed users to draw shapes, using either mouse or ideally, pen input. A sound that is associated with each shape is then played back. As the sound of each shape plays back, it is lit up using animation, creating a strong connection between the shape and it's resulting sound. The application uses the "gesture variation follower" system (Caramiaux *et al.*, 2015), which while promising in principle, didn't have a high rate of accuracy in recognizing the shapes.

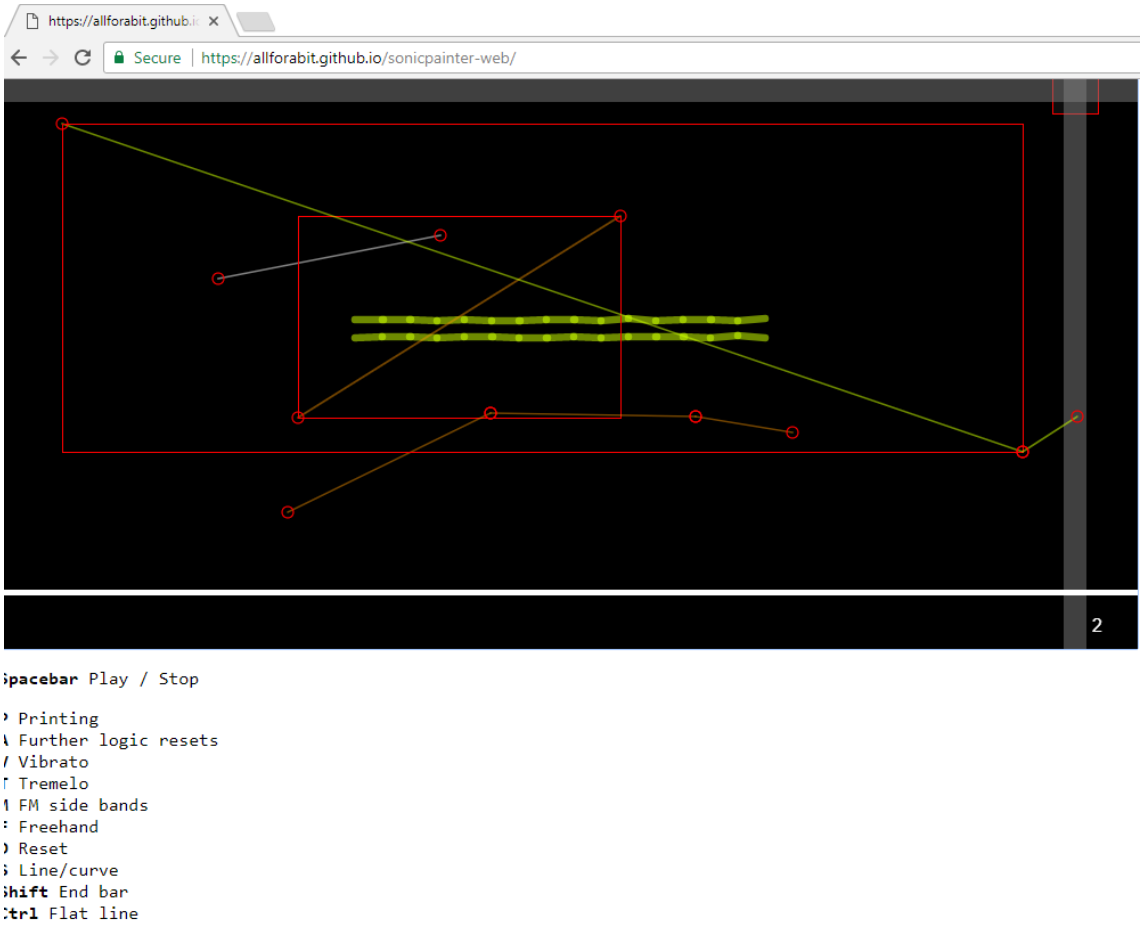


Figure 3: SonicPainter in a web browser

Web version of William Coleman's SonicPainter

A potential starting point that was considered was using the code from William's SonicPainter and porting it to the web browser (Coleman, 2015). This process proved to be quite straightforward. The processing code could be embedded in a webpage with minimal modification, using "processing.js", a web version of the Processing library that enables users to run processing sketches in the Web Browser (Fry and Reas, 2017). Some notable changes that had to be made were removing the OSC functionality as this is not technically possible to use in a browser. In addition, some other pieces of code were removed and altered slightly. As it's not possible to run Max MSP patches in the browser, the audio system was re-implemented using Tone.js (Mann, 2015). As SonicPainter uses simple FM synthesis, a very close approximation to the original version could be created. In the end, it was decided not to build on this codebase however due to the issues with functionality and usability detailed earlier. These would be difficult to resolve in an inherited codebase. The

process of porting the code did however give a more in depth incite into Coleman's implementation.

Setting up the architecture

Clojurescript and javascript npm modules

Despite the fact that clojurescript has existed for six years (Sierra, 2011), some areas of the development process are still difficult, particularly when building out a more complex real world applications. It should be noted that a good deal of work is being carried out to make this a smoother experience and, thusly, it is likely to become easier in the near future (Monteiro, 2017). It should also be noted that building applicatons using plain javascript is not a trivial process either and in will , in all likelihood, include a complex build process using a system like webpack or browserify.

A primary issue that had to be resolved to allow the application development to proceed was the incorporation of javascript npm modules. NPM is the package manager used by *node.js*. Node.js is a javascript platform originally designed for more server oriented applications, but, increasingly, also for rich client-side applications. The related NPM repository houses a large amount of javascript packages (currently 477,000) and is one of largest collections of code in the world . For a pure javascript application, it would be a matter of simply adding the desired libraries as NPM dependencies. However, with the use of ClojureScript, some extra steps needed to be carried out. In addition to adding these dependencies, a javascript file needed to be created that imported these into a `deps` object. This `deps` object could then be referred to in ClojureScript using the standard interop syntax `js/deps.myDependency` (Weller, 2017). At the time of development, an alpha feature that allowed npm dependencies to be declared as part of the project.clj file was experimented with but was not used due to some implementation difficulties. While the project setup was not as elegant or succinct as might be wished, it did provide a stable base to build on. Crucially the rich resource that is the NPM ecosystem could now be harnessed, to use such tools as Paper.js and React.js.

Paper.js and React (Paper.js bindings)

Paper.js runs in the context of a canvas element and thusly it is not possible to directly use React with it. This shortcoming has been addressed in projects such as *three.js react bindings* and *pixi.js react bindings* which allow the use of React's declarative programming style for 3d and 2d scene graph oriented systems that run in the html canvas element. These solutions both work by creating dummy empty DOM elements and hook into the *React.js* lifecycle events to do the real work of updating the scene graph. In many ways the scene graph structure of projects like these (and Paper.js) have a high resemblance to DOM structures and APIs, making React a good fit for them. A similar approach to the above mentioned libraries approach was taken to integrate Paper.js for use in SonicSketch. This worked reasonably well but required quite a bit of setup and ongoing development. During the course of the project build out, a more suitable solution emerged from the open source community. This used the next version of React (16), a version that has better support for render targets that are not the DOM. This has the distinct advantage of not requiring the creation of redundant DOM nodes. The library was far from comprehensive and, thusly, a custom version of the library was used that included some custom functionality required for SonicSketch.

Tone.js and React

In some ways audio output can be thought of in a similar way to the visual output of the app, merely as another type of I/O. Thusly, it can be treated in similar way by React and can use its declarative data oriented system to configure the particular settings and connections in the audio graph. React's lifecycle events can be used to instanciate the various audio generating and processing web audio nodes. This addresses a notable (by design) ommission in Tone.js which does not allow the state of the audio graph to be queried once it has been setup. It is the responsibility of the client code to keep track of and manage this. The advantage offered by introducing React into this part of the system is that it maintains the simple relationship between state and generated output. Conceptually the flow of change is:

1. The state updates
2. React components update their properties accordingly
3. React lifecycle events are triggered which take care of altering, adding and

removing web audio nodes (thus altering the audio being output)

The design of this part of the application is influenced by *React Music*, a system that uses React with *tuna.js*, a web audio library similar to tone.js (FormidableLabs, 2017).

Reagent and React paper.js bindings

The final piece of the jigsaw in the underlying technology stack is the integration of React with ClojureScript via the *Reagent* library. The core syntax of this system uses simple ClojureScript vectors similar to the following:

```
[:div
 "Hello " [:span {:style {:font-weight bold}}
 "world"]]
```

This would result in the following html output:

```
<div>Hello <span style="font-weight: bold">world</span></div>
```

As can be seen, the vectors begin with a keyword that corresponds to the HTML tag name. Additionally, instead of using HTML tag keywords, function calls can be made to generate html by using symbols that reference functions. This allows for code reuse and logic. It was unclear how the Paper.js bindings would work within this system due to the fact that it required a different version of React and uses non standard tag names for the elements that can be drawn on screen such as "Circle" and "Rectangle". This, however, turned out to be more straightforward than expected and the provided Paper.js primitives could be used by simply using the relevant keywords such as `:Circle` and `:Rectangle`. Complex scenegraphs could be constructed by using the following succint ClojureScript syntax to, for instance, describe the playback indicator:

```
[:Group {:position [position 0]
           :pivot [0 0]
           :opacity    0.75}
   ;; This is the main bar that runs from top to bottom
   [:Rectangle {:pivot [0 0]
                :size [1 height]
                :fill-color "#ffffff"}]
   ;; This is the triangle at the top
```

```
[:Path {:segments     [[-5 0] [5 0] [0 7] [-5 0]]
        :fill-color "#ffffff"}]]
```

The `position` and `height` are properties that are passed into the hiccup and trigger updates to the visual display when they change: in the case of position, when the playback position changes and in the case of the height, when the user resizes the browser window. The path element describes the triangle that is places at the top of the screen.

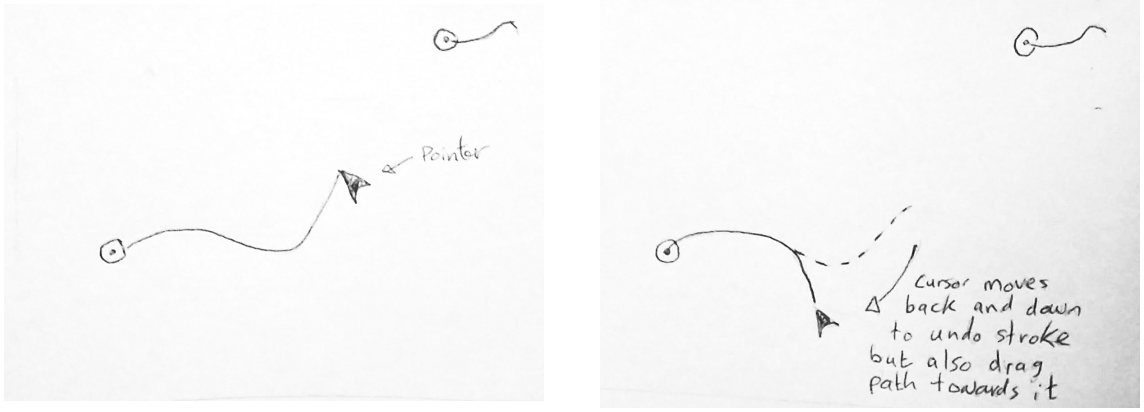
Core functionality - notes

At the core of the application is the creation of timeline events which unfold in a looped fashion. These events are created based on the input of the user with a mouse or mouse like input device. On the production of a valid input gesture, the screen is updated immediately with a visual display of this content. The details of this gesture is stored in memory and the event that will eventually create the sound is registered with Tone.js. Much of the events that occur in the system are captured in a main `View` component which houses the central html canvas element. To aid in organising the large amount of functionality associated with the component, higher order components are used to separate this out into logical groupings. A higher order component is a component that wraps a normal component to add functionality to it and accepts the same properties as the component it wraps (**???**). In this case the most logical grouping is by tool and so there are higher order components setup for each of the tools: draw, vibrato, delete, move, resize and probability.

Adding notes

The sketch tool is the default tool that is activated when the user opens the application. It enables the user to add notes by drawing them onto the screen. The event is captured in the main canvas view and is initiated when the user left clicks the mouse to trigger the following function:

```
(defn pointer-down [{:keys [temp-obj active-preset]} evt]
  (let [pointer-point (.. evt -point)
        ;; Group circle and path are temporary shapes
        group         (js/paper.Group. (clj->js {:position
                        [(.. pointer-point -x) (.. pointer-point -y)]
```



(a) User starts note by dragging left to right

(b) Dragging back clears note

Figure 4: Adding notes in SonicSketch

```
                                          :pivot [0 0]}))
      circle        (js/paper.Shape.Circle. (clj->js {:fillColor "#ffffff"
                                                      :radius    5}))
      path          (js/paper.Path. (clj->js {:strokeColor   "#ffffff"
                                              :strokeWidth   2
                                              :fullySelected true
                                              :segments      [[0 0]]}))]
  (.. group (addChildren #js [circle path]))
  (reset! temp-obj {:path   path
                    :circle circle
                    :group  group
                    :loc pointer-point})))
```

This function receives a hashmap with a reference to a ClojureScript atom to store the temporary visualisation of the newly created note. This function uses javascript interop to directly instanciate paper.js objects and add them to a shared group.

As the user continues to move the cursor further points are added to the path created in the `pointer-down` function. Some constraints however are placed on the creation of the path and only points that are past the last previous from left to right are added. If the users backtracks it lead to a deletion of points, providing an on-the-fly undo like behaviour.

```
(defn pointer-move [{:keys [temp-obj active-preset]} evt]
  (when-let [{:keys [path group] :as temp-obj} @temp-obj]
    (let [pointer-point (.. evt -point)
          rel-pos       (.. group (globalToLocal pointer-point))]
      ;; Only add positive points relative to first
```

```
    ;; Remove points greater than pointer-points
    (when-let [last-seg (.. path getLastSegment)]
      (let [first-seg   (.. path getFirstSegment)
            first-point (-> first-seg .-point)
            last-point  (-> last-seg .-point)
            pointer-x   (.-x rel-pos)
            amp-env     (-> active-preset :envelope)
            stage-width (.. evt -tool -view -viewSize -width)
            max-width   (if (= (-> amp-env :sustain) 0)
                          (let [time (+ (-> amp-env :attack)
                                        (-> amp-env :decay)
                                        (-> amp-env :release))]
                            (-> time
                                ;; Seconds to beats
                                (* (/ js/Tone.Transport.bpm.value 60))
                                (time->euclidian stage-width)))
                          nil)]
        (when (or
               (nil? max-width)
               (< pointer-x (+ (.-x first-point) max-width)))
          (-> path (.add rel-pos))
          (let [greater-segs (filter
                              #(> (-> % .-point .-x) pointer-x)
                              (.-segments path))]
            ;; Remove greater points
            (doseq [seg greater-segs]
              (.removeSegment path (.-index seg))))))))
```

Completion of a note occurs when the user releases the button and to trigger the `pointer-up` function:

```
(defn pointer-up [{:keys [temp-obj active-preset stage-size]} evt]
  (let [{:keys [path circle group loc] :as temp-obj} @temp-obj]
    (.simplify path 10)
    ;; Send the actual note
    (dispatch [:note-add (-> (path->note path loc stage-size)
                             (assoc ,,, :preset active-preset)
                             ;; Use the color from the active preset
                             (assoc ,,, :color (:color active-preset)))] )
    ;; Remove temporary objects
    (.remove path)
    (.remove group)
    (.remove circle))
```

```
  ;; Unset temp obj
  (reset! temp-obj nil))
```

This function simplifies the path by calling the paper.js `simplify` method on the path object and dramatically reduces the amount of data captured while preserving the basic characteristic of the user's stroke. Most importantly it calls the re-frame `dispatch` function to add the note to the `app-db`. A `path->note` function is used to convert the stroke from the domain of euclidean space in the visual space of the canvas to the domain of time-pitch space for use with the audio synthesis system. The `path->note` function can be seen below:

```
(defn path->note [path first-point stage-size]
  "Main entry point to this namespace"
  (let [path-width (.. path -bounds -width)
        width      (:width stage-size)
        height     (:height stage-size)]
    {:freq        (domain/euclidean->freq (.. first-point -y) height)
     :onset       (domain/euclidean->time (.. first-point -x) width)
     :duration    (domain/euclidean->time path-width width)
     :velocity    0.5
     :enabled     true
     :probability 1.0
     :color       @(col/as-css (get colors (rand-int 100)))
     :height      (.. path -bounds -height)
     :width       (.. path -bounds -width)
     :envelopes   {:frequency {:raw     (paper-path->vec path [width height])
                               :sampled (paper-path->sample path stage-size)}
                   :vibrato   (reduce (fn [a b]
                                        (assoc a b [b 0]))
                                      (sorted-map) (range 11))}}))
```

The domain of time-pitch is used to store the notes in memory and makes it possible to maintain a relative relationship between the screen size and the drawn notes.

The dispatched note event is handled by a re-frame `reg-event-db` handler which describes the alteration that is required to be made to the `app-db`. It also uses a series of interceptors, to perform validation of the `app-db` and to remove some of repeated code from the event handler functions. Interceptors are similar conceptually to middleware and is the place where all of the side-effects arising from an event are actioned. Moving application side-effects here ensures that they are isolated and the majority of the program can be kept as pure functions. As can be seen the handler

function is very simple:

```
(reg-event-db
 :note-add
 note-interceptors
 (fn [notes [note-info]]
   (let [id    (allocate-next-id notes)
         note  (assoc note-info :id id)]
     (if (>= (:duration note) 0.001)
       (assoc notes id note)
       notes))))
```

This does a simple check to make sure that note has a minimum duration and if so alters the notes vector to include the new vector which will instruct re-frame to update the `app-db` with this new state.

The structure of the note hashmap is defined using *clojure.spec*, a core Clojure/Clojurescript library to perform data validation. The note specs are defined as follows:

```
(s/def ::id int?)
(s/def ::freq float?)
(s/def ::onset float?)
(s/def ::duration float?)
(s/def ::velocity float?)
(s/def ::color string?)
(s/def ::note (s/keys :req-un [::id ::freq ::onset ::duration]
                      :opt-un [::velocity]))
```

Although not specified here, notes also have an `envelopes` key that stores frequency and vibrato envelopes:

```
{:frequency {:raw [{:point [-0.01 99.7]
                    :handle-in [0 100]
                    :handle-out [0 99.8]}  ...
                   {:point [2.8 98.02]
                    :handle-in [-0.10 100.4]
                    :handle-out [0 100]}]
             :sampled [-0.36991368680641185
                       ;; ...
                       -2.172026174596564]}
 :vibrato {0 [0 0], ... 10 [10 0]}}
```

The update in state puts re-frame's subscription system into action and any views that are subscribed to the changed application state are now re-rendered. The `graphics-notes` view for instance is subscribed to `:notes`, `:graphics-stage-width`, `:graphics-stage-height` and `:mode`:

```
(defn graphics-notes []
  (let [notes          (subscribe [:notes])
        width          (subscribe [:graphics-stage-width])
        height         (subscribe [:graphics-stage-height])
        ....]
    (into [:Group]
          (map (fn [note]
                 ^{:key (:id note)} [graphics-note* ...]) @notes))))
```

When a note is added this render function will run which will call the `[graphics-note]` component for each of the notes, and update the visual display of the screen to show the new note. A similar process happens in the audio system except, in this case, new web audio nodes are created, timeline events are queued up at the correct time and audio envelopes are setup that trace the curve of the drawn lines.

```
(defn fm-synth [{:keys [out] :as props} & children]
  "FM synth"
  ;; The new synth is instanciated using js interop
  (let [synth (js/Tone.FMSynth. (clj->js (dissoc props :out)))]
    (reagent/create-class
     {:component-did-mount
      (fn []
        ;; The synth is connected to it's output (which is passed
        ;; in as a property to the component)
        (.. synth (connect out)))
      :reagent-render
      (fn [props & children]
        ;; The render function renders a dummy span dom element and
        ;; renders it's children and passing it's synth as the out
        ;; for these components.
        (into [:span]
              (map (fn [child]
                     (assoc-in child [1 :out] synth))
                   children)))
      :component-will-unmount
      (fn []
```

```
    ;; Here we dispose of the synth
    ;; This will happen when the parent note is removed or when
    ;; a live code reload happens
    (.. synth dispose))}))))
```

The above shows the `fm-synth` which sits at the heart of the audio generating part of the system. Potential parent components of this would be audio effects or the master bus. It's child components are comprised of note events and envelopes that drive frequency changes over the course of note playback. The composition of events, envelopes, synths, effects and channels is shown in truncated form:

```
;; Parent component is the project and
;; has settings such as tempo
[project {:project :settings}
 [master-bus {}
  ;; Master volume is set here
  [volume {:volume :settings}
   ;; Adds a simple reverb effect
   [reverb-effect {}
    [
     ;; First note
     ;; Each note has a vibrato effect
     [vibrato-effect {}
      ;; Envelope to control vibrato depth
      [timeline-evt evt
       [envelope {:param "depth"
                  :env   vib-env}]]
      ;; The fm synth that generates the signal
      [fm-synth (get-in evt [:preset])
       ;; Timeline event that takes care of queuing
       ;; it's child components
       [timeline-evt evt
        ;; In this case a note
        [note {:note :settings}]
        ;; And a frequency envelope
        [envelope {:param "frequency"
                   :state state
                   :env   freq-env}]]]]
     ;; Second note
     [vibrato-effect {}
      ;; ...
     ]
```

```
    ;; ...
    ]]]]]
```

Editing notes

Figure 5: SonicSketch tools panel

Once created, a number of editing operations can be performed on notes. Apart from the **delete** action all of these operations involve first selecting the note on the **mouse down** event, carrying out the editing of the note and completing the action with the **mouse up** event. The particular tool can be activated by clicking the icon the right-hand side of the screen (see figure 5). Clicking one of these dispatches the `:change-mode` event and changes the `app-db` to the specified mode as well as setting the application to state `:pending`. Flowing from this change are a number of changes:

1. The relevant paper.js tool is activated which will route further mouse events to the appropriate dispatch handlers. For instance, when the `:resize` mode is selected, **mouse move** events will be handled by `:resize-tool-pointer-move`.
2. It updates the visual look of the cursor to remind the user which of mode they are in.
3. A visual indicator is shown around the icon of the selected mode

The overall effect of this is an activation of a number of different modes, each of which will now be discussed.

In the case of deleting notes, the event is simply raised in the **click** event of the note, which takes care of dispatching the `:note-delete` event. This is handled by a very simple handler, `(dissoc notes note-id)` that instructs re-frame to remove the note in question from the `app-db`. Similar to the process that occurs when a note is added, the visual display is now updated to no longer show the note. In addition, the signal generator, effects, envelopes and events associated with the note are removed from the audio component tree. React's lifecycle events take care of cleaning up any synths and effects by calling their `dispose` methods.

Moving notes and resizing notes work very similarly and both follow the most basic pattern described above of selection, manipulation, and completion. The note selection event is raised from the note's **mouse down** event handler, dispatching the `:note-select` event with the **note-id**. This updates the `app-db` to set the `:active-note-id` to the received id, sets the app state to `:active`, sets the note state to be `:active` and resets previously active notes to state, `:normal`. This state update changes the visual display of the active note to be highlighted by slightly lightening the colour of the note's surrounding glow. More importantly, the note now becomes the active item on which further user interactions are performed on. In the case of the **move** tool the stored onset and pitch properties of the note are altered to reflect the position of the user's mouse. This, in turn, updates the visual display of the note and causes the audio system to re-queue the note events to the new onset and pitch values. The **resize** tool, on the other hand, changes the size of the starting node of the note and alters the velocity accordingly. The altered size is calculated from the coordinates of the **mouse down** event and is clamped to a maximum size that corresponds to the maximum velocity of the note.

The slightly more esoteric **probability** tool works in a very similar fashion to the **resize** tools and again creates adjustments that work relative to the initial **mouse down** coordinates. The visual effect that this creates, however, is a dulling of the saturation of the note. Its effect is to add an element of randomness and depending on how saturated the note is, it will cause the note to randomly skip. The code for this is below:

```
(let [enabled (-> (Math.random)
                  (<= ,, probability))]
  ;; Only trigger the note if enabled is true.
  ;; If probability is 1.0 will always be true.
  (if enabled
```

```
    (do
      (rf/dispatch [:note-enable id])
      (some-> out
              (.triggerAttackRelease ,,, freq (prep-time dur) t velocity)))
    (rf/dispatch [:note-disable id])))
```

If the probability is fully saturated (corresponding to `1.0`, the default value), the note will play on every loop. If, however, it is below this it will skip in a random but probabilistic fashion, adding a small amount of stochasticism to playback.

The vibrato tool is the most complex in its implementation and involves a small popup UI element that allows the user to draw in a vibrato envelope which is visually reflected in the note lines as an overlaid sinewave. After selection occurs, in the usual way, **mouse move** events dispatch to the `:vibrato-continue` handler which updates a vibrato modulation parameter in the note, a single float value that represents the current real-time vibrato value. This modulation parameter is passed back to the view through properties which triggers a `:component-did-update` function call. It is here that a dispatch is made to the `:vibrato-envelope-point` to create the envelope point. The reason for the back and forth between the view and the event handlers is to allow the event handlers to manage the state changes but deal with the specifics of the geometry of the vibrato overlay in the view.

A visual overlay is shown to the user when a vibrato action is started, that shows a 10 point envelope, whose points are all set to `0.0` by default. The user can then drag varying heights at various points on the horizontal axis and create the time-varying vibrato envelope. This envelope is visualised (and remains visible after the vibrato operation has completed) with a sinewave that tracks the curve of the frequency envelope and varies its amplitude depending on the strength of the vibrato at that particular point. The code to achieve this is below:

```
(->> (range 0.0 1.0 0.01)
     (map #(vector % (get-pos-of-curve-at-time curve %)))
     (map  (fn [[time p]]
             (let [norm          (.. curve (getNormalAt time))
                   loc           (.. curve (getLocationAtTime time))
                   relative-left (- (.. loc -point -x) left)
                   idx           (->
                                   ;; Normalize
                                   (/ relative-left width)
                                   ;; Mult by amp-env count
```

```
                        (* (count amp-env))
                        ;; Add amp-env starting point
                        (+ (-> amp-env (first) (first)))
                        ;; Round it off
                        (Math/round))
      amp             (-> (get amp-env idx)
                          (second))]
;; Amp is here
(sine-eq-along-vec time p norm (* amp 100) l)))))
```

TODO Secondary functionality

Aside from the core functionality of note creation and editing, a number of other use cases were covered in the implementation. This includes transport controls, to allow the user to start and stop playback; some simple animation to show which notes are being played and to tighten the link between the visuals and the audio; undo and redo functionality; a fullscreen toggle; save and load functionality. Some of these extra features were made more straightforward than would normally be the case due to the architecture of placing the state in a single place, the re-frame **app db**.

Both the transport and the fullscreen mechanism consisted of a simple FSM whose transitions are caused by user actions. The transport state machine responds to both the **play toggle** button click and **spacebar** keypresses to toggle its state between playing and stopped and based on this, triggering playback in tone.js. The fullscreen FSM transitions occur not only when the user clicks the fullscreen button but also when the browser fullscreen status changes. This keeps the UI in a correct and consistent state at all times, regardless of how the state is reached.

Undo/redo and save/load functionality was relatively straightforward to implement for the reasons mentioned above (centrally located state) as well as the ease of serialising ClojureScript data. Adding undo/redo was as easy as adding an additional dependency and adding it as an interceptor to any events that needed to be undoable such as adding or deleting notes. The particular parts of the app state that needed to be restored were also configured and only included the `notes` collection and the `tempo`. The same two elements were saved and restored by the save/restore functionality which worked by serialising this data as a string and restoring with a single function call to *ClojureScript*'s `reader/read-string`.

Some simple animation was added to playback to increase heighten the connection

between visuals and audio. To achieve this, the `:note` subscription is itself subscribed to `:playback-beat` a reactive value that is kept up to date via the use of events that fire on every animation frame. The subscription checks if the playback head is within the notes range and if so sets its `:playing` property to `true` and updates `:playback-time` to reflect the position of the playback head in the note. Similar to any other changes in state, the note views react to this and updates visual content accordingly to create the animation.

```
(rf/reg-sub
 :note
 :<- [:notes-raw]
 :<- [:playback-beat]
 (fn [[notes playback-time] [_ id]]
   (let [{:keys [onset duration] :as note} (get notes id)
         end-time                          (+ onset duration)]
     (let [updated-note (if (and
                             (> playback-time onset)
                             (< playback-time end-time))
                          (-> note
                              (assoc ,,, :playing true)
                              (assoc ,,, :playback-time (- playback-time onset)))
                          (if (true? (-> note :playing))
                            (assoc note :playing false)
                            note))]
       updated-note))))
```

Conclusion

This chapter outlined the development process that took place to bring SonicSketch to life. The early prototype work was detailed, in addition to the web port of SonicPainter, all work that contributed conceptually to the final artifact. The setup of the technical architecture that brings together tone.js, React, ClojureScript, Reagent and Paper.js was outlined. An in depth description of the primary functionality that enables users to draw and edit notes on the screen was given, followed by a more brief look at secondary functionality such as undo, saving and note animation.

Agostini, A. and Ghisi, D. (2015) 'A Max Library for Musical Notation and Computer-Aided Composition', *Computer Music Journal*, 39(2), pp. 11–27. doi: 10.1162/COMJ_a_00296.

Caramiaux, B. *et al.* (2015) 'Adaptive gesture recognition with variation estimation for interactive systems', *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 4(4), p. 18. Available at: http://dl.acm.org/citation.cfm?id=2643204 (Accessed: 21 April 2017).

Coleman, W. (2015) *sonicPainter: Modifications to the Computer Music Sequencer Inspired by Legacy Composition Systems and Visual Art (MMT thesis).*

FormidableLabs (2017) *React-music: Make beats with React!* Formidable. Available at: https://github.com/FormidableLabs/react-music.

Fry, B. and Reas, C. (2017) *Processing.Js.* Available at: http://processingjs.org/ (Accessed: 23 August 2017).

Mann, Y. (2015) 'Interactive music with tone. js', in *Proceedings of the 1st annual Web Audio Conference.* Available at: http://wac.ircam.fr/pdf/wac15_submission_40.pdf (Accessed: 28 July 2017).

Monteiro, A. N. (2017) *ClojureScript - ClojureScript is not an Island: Integrating Node Modules.* Available at: https://clojurescript.org/news/2017-07-12-clojurescript-is-not-an-island-integrating-node-modules (Accessed: 23 August 2017).

Sierra, S. (2011) *Clojure :: Introducing ClojureScript.* Available at: http://clojure.com/blog/2011/07/22/introducing-clojurescript.html (Accessed: 23 August 2017).

Weller, T. (2017) *Clojurescript/Reagent : Importing React components from NPM, Tomer Weller / Blob.* Available at: http://blob.tomerweller.com/reagent-import-react-components-from-npm (Accessed: 23 August 2017).