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 tricky setup process of getting the live reload system working (as described in the previous chapter). Some detailed discussion of the the core functionality of the system is then given. The core of the system primarily consists of timeline events, which visually manifest themselves as strokes on the canvas and aurally as fm syntesized frequency modulated sounds. As will be discussed an important aspect of any well constructed software system is a good degree of seperation of concerns, a characteristic espoused to and evident in the resulting code. To this end, the code that brings the central functionality to life can conceptually divided into a data or entity layer, a business logic or use case layer and an output layer which in this case consists of the visual output into a html canvas element and the audio output through the web audio api. In this sense the architecture conforms to the principles of the Clean Architecture as presented by Bob Martin (???).

Early prototype work

Melodypainter

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. 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 [0/1]

A separate application was created in Processing which allowed users to draw shapes, using either mouse or ideally, pen input and have a sound that is associated with each shape 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_adaptive_2015], which while promising in principle, didn't have a high rate of accuracy in recognizing the shapes.

TODO Web version of William Coleman's SonicPainter [0/1]

A potential starting point that was considered was using the code from William's SonicPainter and porting it to the web platform. This process proved to be quite straightforward. The processing code could more or less be embedded in a webpage as is using "processing.js", a web version of the Processing library that enables users to run processing sketches in the Web Browser. 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 had to be commented out and tweaked. As it's not possible to run Max MSP patches in the browser, the audio system was re-implemented using Tone.js. 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 as there were some issues with functionality and useability that would be difficult to resolve in an inherited codebase. A fundamental issue was that transitions between certain states would cause crashes or unpredicatable behaviour. An example of this is when a user attempts to use the vibrato tool while in the process of creating a note. Instead of either finishing the note and starting the vibrato tool or disallowing the behaviour, the program would crash. This is a common problem in software development and is evidence even in commercial products. To alleviate such issues in the new codebase a more disciplined approach would be taken to managing transitions between states. The process of porting the code did however give a more in depth incite into Coleman's implementation incuding the pitfalls mentioned above. In addition, the basic visual look and conceptual functionality would form the basis of the workings of SonicSketch.

Actual implementation

Setting up the architecture

1. Clojurescript and javascript npm modules

   Despite the fact that clojurescript has existed for six years(???), some areas of the development process are still difficult, particularly when building out a more complex real world application. It should be noted that a good deal of work is being carried out to make this a smoother experience and thusly these pains are likely to become less of an issue in the near future (???ref). It should also be noted that building applicatons using

plain javascript is not a trivial process and in all likelyhood will include a build process using a system like webpack or browserify. A primary issue that had to be resolved to allow the application to be built out was the incorporation of javascript npm modules. NPM is the module system used by node.js originally for more server oriented technologies but increasingly for rich clientside applications. For a purely javascript application, it would be a matter of simply adding the desired libraries as dependencies. However, with the use of clojurescript some extra steps needed to be carried out. In addition to adding the dependencies, a javascript file was 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'. 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 in due to difficulties getting it to work. While the project setup was not as elegant or succint as might be wished, it did provide a stable base to build on and a means to harness the rich resource that is the NPM ecosystem and use such tools as Paper.js and React.js.

2. Paper.js and react.js (paper.js bindings)

As has been outlined in the previous chapter, a declarative coding style would be employed to enable a live coding workflow and to avoid a building a codebase that is increasingly difficult to understand. In other words the code should as much as possible describe the "what" of the functionality rather than the "how". These qualities emerge quite naturally when using the *React.js* architecture. Paper.js however runs in the context of a canvas element and thusly it is not possible to directly use *React.js* 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 declaritive programming style for 3d and 2d scenegraph 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 the real work of updating the scenegraph. In many ways the scene graph structure of projects like these and indeed Paper.js exhibit 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 and worked reasonably well but required quite a bit of setup. During the development of the project, a more suitable solution emerged from the open source community at an opportune time. This used the next version of *React.js* which has better support for render targets that are not the DOM and 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.

3. Tone.js and react.js

4. Reagent and react.js paper.js bindings

Core functionality - timeline events (or notes)

1. Introduction
    - Describe the core functionality
    - Describe core entities
2. Add timeline event
    - Business logic
    - UI
    - Audio
3. Add vibrato
    - Business logic
    - UI
    - Audio
4. Remove note
5. Move note
6. Change sound (preset system)
7. Probability

Secondary functionality

1. Introduction
2. Transport controls
3. Animation (current play position & notes)
4. Undo and redo
5. Fullscreen
6. Outer UI
7. Save and load file

Performance issues

Conclusion

- Summarise the resulting artifact