Procedural Soundscape

Claire Goeckner-Wald & Amy Xiong

Contents

1 Executive Summary

We built a website with a procedural-generated soundscape inspired by 'coffeehouse jazz'. While we did not integrate the soundscape with live variables, we did integrate it with a synesthetic background image. It would not be terribly difficult to integrate the soundscape with live variables at this point, but we chose not to due to time constraints. The music consists of a single melody line, played by a saxophone, and an underlying bass line, played by a electric bass. We generate music measure by measure, choosing notes and rhythms randomly from a preset selection (for notes, the preset is a randomly-chosen scale, and for rhythms, a set of 8 rhythms, which then may be lengthened or shortened). To make it sound more musical, we limit the intervals between consecutive notes, and include rests, repeats, and runs. The music can be paused, played, or restarted with new parameters. We chose to use `tonejs-instruments` because it integrated open-source `WAV`-format files from real instruments with `Tone.js` [3]. We tested the final product in both the latest version of Mozilla Firefox (Mozilla Firefox Quantum 63.0) and Google Chrome (Google Chrome 70.0). Rather than using Google App Engine, or similar, we simply hosted the website for free on Claire's personal GitHub site, cgoecknerwald.github.io/procedural-soundscape. This greatly simplified the workflow, as we could easily push HTML/CS/JS to the GitHub repository, and it would then be immediately updated online with no extra work.

1.1 Description

The entire codebase (and the original `LaTeX` files for this very report) can be found on Claire's GitHub repository: github.com/cgoecknerwald/procedural-soundscape. The web application is live at cgoecknerwald.github.io/procedural-soundscape/.

- A web application
- A procedural soundscape
- A visual landscape

1.2 Technical Requirements

- Browser: Firefox Quantum 63.0 and above, or Google Chrome 70.0 and above.
- User input: Interaction with web browser via cursor.
- User output: Any functioning speakers will do.

Since Firefox and Chrome use unique browser engines – Gecko and Blink, respectively – we reasoned that this would be sufficient coverage. Brief trials with Safari, which uses WebKit, have been successful, as well. We did not test at all on Internet Explorer.

1.3 Technical Challenges

We had issues with `Tone.js`, namely with building realistic synths. Splitting our code into multiple files and including the recorded instrument files gave us CORS permission issues, for local testing. We ended up finding a workaround for Firefox, but not for Chrome, so we had to switch to Firefox for local testing.

1.4 Licensing

We chose to license our repository with the MIT License.

2 Technical Design

2.1 Sound Design Tool Discussion

We've decided to use `Tone.js` instead of `Audiosynth.js` by Keith William Horwood. We also chose `Tone.js` over `SuperCollider`, `Flocking.js`, `PureData`, because of web integration ease. `Flocking.js` also had web integration but a smaller online community and seemed less robust and less abstracted. `Audiosynth.js` did not have looping technologies. We briefly considered integrating our whole product with `Audiosynth.js` but it proved to be too difficult.

2.1.1 PureData

One of `PureData`'s strong points seems to be its easy and intuitive low-level sound synthesis, but we were more interested in higher-level procedural music generation than detailed sound creation. In general, `PureData` was far too low-level (and therefore tedious and slow) for our plans. We also weren't quite sure how we'd integrate into a web application.

> Pure Data is an open source visual programming environment that runs on anything from personal computers to embedded devices (ie Raspberry Pi) and smartphones (via libpd, DroidParty (Android), and PdParty (iOS). It is a major branch of the family of patcher programming languages known as Max (Max/FTS, ISPW Max, Max/MSP, etc), originally developed by Miller Puckette at IRCAM.
>
> Pd enables musicians, visual artists, performers, researchers, and developers to create software graphically without writing lines of code. Pd can be used to process and generate sound, video, 2D/3D graphics, and interface sensors, input devices, and MIDI. Pd can easily work over local and remote networks to integrate wearable technology, motor systems, lighting rigs, and other equipment. It is suitable for learning basic multimedia processing and visual programming methods as well as for realizing complex systems for large-scale projects.
>
> Algorithmic functions are represented in Pd by visual boxes called objects placed within a patching window called a canvas. Data flow between objects are achieved through visual connections called patch cords. Each object performs a specific task, which can vary in complexity from very low-level mathematical operations to complicated audio or video functions such as reverberation, FFT transformations, or video decoding. Objects include core Pd vanilla objects, external objects or externals (Pd objects compiled from C or C++), and abstractions (Pd patches loaded as objects). [10]

2.1.2 SuperCollider

`SuperCollider` seemed okay at first, but we could not see an easy way to integrate it with HTML/JS/CSS (knowing that we wanted to build a web application). And similar to `PureData`, `SuperCollider` seemed a bit too low-level for our plans. In particular, our project was unlikely to take advantage of the extensive sound synthesis tools that it provided.

> SuperCollider is a platform for audio synthesis and algorithmic composition, used by musicians, artists, and researchers working with sound. It is free and open source software available for Windows, macOS, and Linux. [8]

2.1.3 Flocking.js

`Flocking.js` was a strong contender for our choice of sound design technology. However, it was very difficult to find examples of projects using `Flocking.js`. The documentation and other support resources were also a little lacking.

> Flocking is a JavaScript audio synthesis framework designed for artists and musicians who are building creative and experimental Web-based sound projects. It runs in Firefox, Chrome, Safari, Edge, and Node.js on Mac OS X, Windows, Linux, iOS, and Android.
>
> Flocking is different. Its goal is to promote a uniquely community-minded approach to instrument design and composition. In Flocking, unit generators and synths are specified declaratively as JSON, making it easy to save, share, and manipulate your synthesis algorithms. Send your synths via Ajax, save them for later using HTML5 local data storage, or algorithmically produce new instruments on the fly.
>
> Because it's just JSON, every instrument you build using Flocking can be easily modified and extended by others without forcing them to fork or cut and paste your code. This declarative approach will also help make it easier to create new authoring, performance, metaprogramming, and social tools on top of Flocking.
>
> Flocking was inspired by the SuperCollider desktop synthesis environment. If you're familiar with SuperCollider, you'll feel at home with Flocking. [4]

2.1.4 Audiosynth.js

Partway through the project, we stumbled upon `Audiosynth.js`, and briefly considered switching to it. One of the main draws was that it had several instruments built into it already. However, we found `Audiosynth.js` to be a lot less fully-featured than `Tone.js`. In particular, `Audiosynth.js` did not have the sophisticated scheduling system that `Tone.js` did, meaning that we'd have to handle all the low-level details of music timing ourselves. We ultimately decided that this would require too much work, and opted to stick with `Tone.js`.

> Dynamic waveform audio synthesizer, written in Javascript. Generate musical notes dynamically and play them in your browser using the HTML5 Audio Element. No static files required. (Besides the source, of course!) [5]

2.1.5 Tone.js

In the end, we chose `Tone.js` out of all three Javascript sound design frameworks for its robust online community (and therefore support). We were able to find many examples of very cool musical projects others had done using `Tone.js`, and there was also quite extensive documentation for it. `Tone.js` also provided a very sophisticated scheduling system that made music generation much simpler.

> Tone.js is a Web Audio framework for creating interactive music in the browser. The architecture of Tone.js aims to be familiar to both musicians and audio programmers looking to create web-based audio applications. On the high-level, Tone offers common DAW (digital audio workstation) features like a global transport for scheduling events and prebuilt synths and effects. For signal-processing programmers (coming from languages like Max/MSP), Tone provides a wealth of high performance, low latency building blocks and DSP modules to build your own synthesizers, effects, and complex control signals. [13]

2.2 Technological Review

We looked at Karplus-Strong String Synthesis but determined it was only relevant if we used `Audiosynth.js` [11].

2.2.1 Tambien's Tone.js

For the reasons discussed in '2. Sound Design Tool Discussion' above, we ultimately chose `Tone.js` as our main sound design package.

2.2.2 Nbrosowsky's tonejs-instruments

The package `tonejs-instruments` was instrumental (haha) in our final project. It features soundclips for many instruments, such as piano, clarinet, guitar, violin, etc., all in `WAV`, `MP3`, and `OGG` formats. It allowed us to move away from a very "synth-y" sound to real soundclips from instruments. We ended up using a saxophone for our melody instrument and an electric bass for our bass line.

2.2.3 Wheelibin's synaesthesia

Synaesthesia is a web application that randomly produces music based off of an input string. Its method of generating music with multiple instrument lines (including drum kit), incorporating scales and chord progressions, was really useful examples for us to investigate. We modified his drum kit and synths. We also based our `assets/js/chords.js` and `assets/js/rhythms.js` off of `synaesthesia`.

2.2.4 Tambien's jazz.computer

We got mostly drums synths and pads from `tambien` (Yotam Mann, creator of `Tone.js`)'s `jazz.computer` (works best in Chrome or Safari). `Tambien`'s `jazz.computer` but they had a very strange music set-up with interpolation and mediators between synths. We attempted to replicate by playing mulitple synths simulataneously but the resulting 'piano' was very tinny and sounded like Scottish bagpipes.

2.3 Musical Review

2.3.1 Scheduling

The first challenge was in getting `Tone.js` to produce random music that would play forever. `Tone.js`'s has a nice scheduling system for producing music, with scheduling

components that allowed you to play a series of notes in a row (e.g. `Tone.Part`).

However, `Tone.js` can only repeat in regular intervals. It could not, for example, calculate the amount of time a section of music would take, then automatically repeat after that section was finished. This meant we had to make each randomly-generated section of music be of the same length, in order for them to fit together without breaks in the music. We thus decided to generate music one full measure at a time.

`Tone.js` also allows you to have multiple notes playing at the same time. As such, you must specify both the duration of the note and when it should be played, not just one or the other. We had to thus separately track the time elapsed as we randomly generated our notes, in order to be able to know when to schedule the next note.

2.3.2 Rhythm

For random rhythm patterns, we used a set of 8 rhythms patterns [7]:

whole	𝅝
half half	𝅗𝅥𝅗𝅥
dotted-half quarter	𝅗𝅥.♩
quarter dotted-half	♩𝅗𝅥.
half quarter quarter	𝅗𝅥♩♩
quarter half quarter	♩𝅗𝅥♩
quarter quarter half	♩♩𝅗𝅥
quarter quarter quarter quarter	♩♩♩♩

All rhythm constants are stored in `assets/js/rhythms.js`.

Note that each of these rhythms take up exactly one measure (in $\frac{4}{4}$ time). We then augmented this set by using the same patterns with shorter notes. We halved all the note lengths to get an equivalent set of rhythm patterns that take up half a measure, and then halved the lengths again to get another set that take up one quarter note.

We generate music one measure at a time, but within each measure, we generate the measure by connecting together these rhythm patterns. Since some of the patterns are shorter than a measure (the length of either one quarter note or one half note), we must be sure to generate the appropriate number of rhythm-segments, of the appropriate length, to exactly equal one measure of music.

For example, we might generate a measure of music by first generating four sixteenth-notes (a one-quarter-note rhythm pattern), then one half-note (a one-half-note rhythm pattern), then finally two eighth-notes (a one-quarter-note rhythm pattern again). We cannot, however, generate first four sixteenth-notes, then four quarter-notes, as that will last longer than one measure. We also cannot generate four sixteenth-notes, then one half-note, then stop, since that will last shorter than one measure.

`createMeasure` handles creating an individual measure. The function randomly decides what length of music to generate (1, 2, or 4 quarter notes of music) that falls within the time remaining in the measure, until it's filled up the entire measure. It then calls `createSectionNotes` (which calls `generateNotes`) with the specified length. These functions randomly pick one of the 8 possible rhythms (and they also handle pitch generation, runs, and repeats, described in later sections). The shorter rhythms (half-note-length or quarter-note-length) are obtained by dividing the note lengths according to whether it's generating 1, 2, or 4 quarter notes of music.

The bass line is much simpler. It has a set of four possible rhythms: 1. one note every beat, 2. one note on the first beat of the measure, 3. one note on the first and third beats, and 4. one note on the second and fourth beats. At the start of the music, one of these rhythms is randomly chosen and repeated every measure.

2.3.3 Pitch Generation

Pitches are also randomly generated. When the website is first loaded, it randomly picks a tonic note and a scale type to build a scale (e.g. C Major, A-flat Minor, D-sharp Dorian). Note pitches are chosen only from this scale.

The bass line plays only the tonic of the scale. The melody line (possibly) changes pitch for every new note. Since in real music, the difference in pitch from one note to the next is usually just a small interval, our program randomly chooses a new pitch that is within three notes (in the scale) from the previous pitch. For example, if the scale is D major, the melody might move from D to G to F-sharp to E to E again. The melody could move from D to the A below, but never to the A above.

To actually choose the next note, we start from a variable, `currPitchIndex`, which records the index into the array of available notes for the current note. We then generate an array of the indices of all the possible next notes, in the distribution specified by our constant array of intervals. An index is randomly chosen from this array of possibilities, the index is used to obtain the next pitch, and `currPitchIndex` is updated to this index, to be used for the next note.

It's also common in music for there to exist a "run" of notes. By a "run", we mean a series of ascending or descending notes that follow the scale. Some examples would be G-A-B-C in C major, A-flat-G-F in F minor, etc. Thus, every time a new rhythm-and-note-pattern is generated, there's a random chance for the melody to start a run of notes. While in a run, the interval from one note to the next is no longer randomized, and is instead always fixed at 1 (for ascending runs) or -1 (for descending runs). While in a run, every time a new rhythm-and-note-pattern is generated, there is also a random chance to stop the run.

We specified octave ranges for the pitch, as well. For the saxophone recordings, we found that only octaves 2 through 4 sounded good, and so all the melody pitches are restricted to these three octaves. If a run tries to go outside the octave bounds, we stop the run. Our original system for choosing the next note was to first randomly chose an interval, then calculate the pitch from `currPitchIndex` and that interval. One of the motivations behind switching to the current approach (first generate all possible pitches from all possible intervals, then randomly choose a pitch) is that it handled staying within the octave range better. With the previous approach, we would simply re-generate the note until it fell within the correct octave range. This, however, is inefficient, and actually has the theoretical possibility of looping forever.

2.3.4 Repeats

Music contains a lot of repetition—in whole measures or phrases, in single notes, in melodic lines and rhythms. To increase the musicality of our music generator, we added some very simple repetition.

Every time a new rhythm pattern is generated, there is a fixed chance that that rhythm pattern might be exactly repeated. If there isn't enough room in the measure to repeat the pattern (i.e. if repeating the pattern would extend past the end of the measure), then

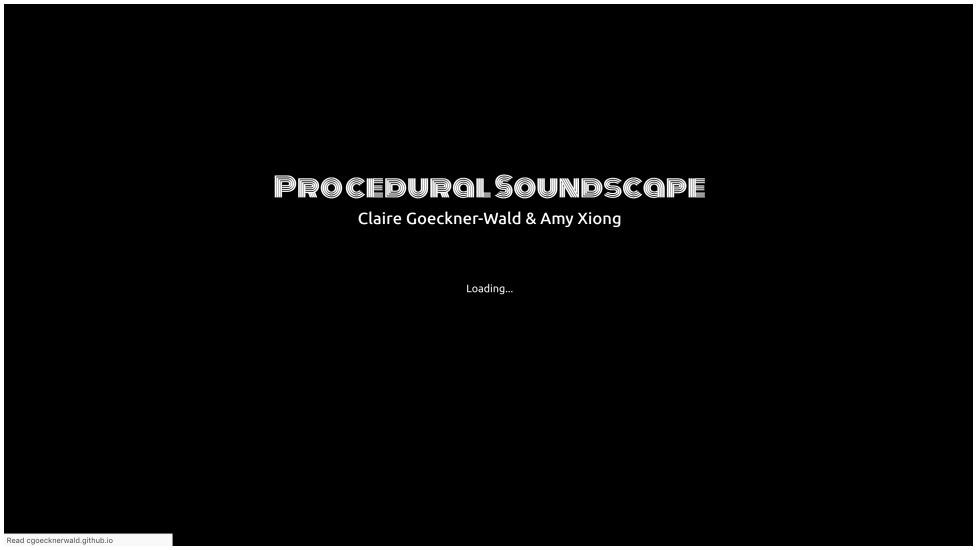

Figure 1: The loading screen as seen on Firefox Quantum 63.0. During this screen, fonts, and `WAV` soundfiles are loading.

the pattern won't be repeated. In addition, every individual measure also has a fixed chance of repeating.

2.3.5 Randomness

All the randomness is produced by using `Math.random()`.

A series of constants control much of the randomness. These constants govern the chance the next note (while not in a run) will be higher than the previous note (set to 0.5), the chance to start a run (0.5), the chance to end the run (0.7), the chance the run will be an ascending run (0.5), the chance to repeat the previous rhythm-and-note pattern (0.3), the chance to repeat an entire measure (0.2), and the chance to have a rest instead of a note (0.3). Since these are constants, it's very easy for us to quickly change them, which then alters the character of the melody.

For the rest of the randomness, all the options have equal chance. These include choosing one of the 8 rhythms for the melody, choosing the bass line rhythm, choosing which scale type and tonic to use, and choosing whether to generate 1, 2, or 4 quarter notes of music.

The one exception is the intervals between notes in the melody (while not in a run). For this, our code randomly chooses an integer out of a constant array, and this integer determines the magnitude of the interval between the current note and the next note. Each element in the array has an equal chance of being chosen, but some numbers are duplicated, to give a distribution of intervals that we thought would sound best. The array: `[0, 0, 1, 1, 1, 2, 2, 3]`.

2.4 User Interface

We wanted to make our user interface fun without being confusing, or being so difficult to implement that it distracts from the work of sound design. We settled on a semi-transparent paneling design that would allow us to compartmentalize the different functions while showing off a collection of highly-textured backgrounds.

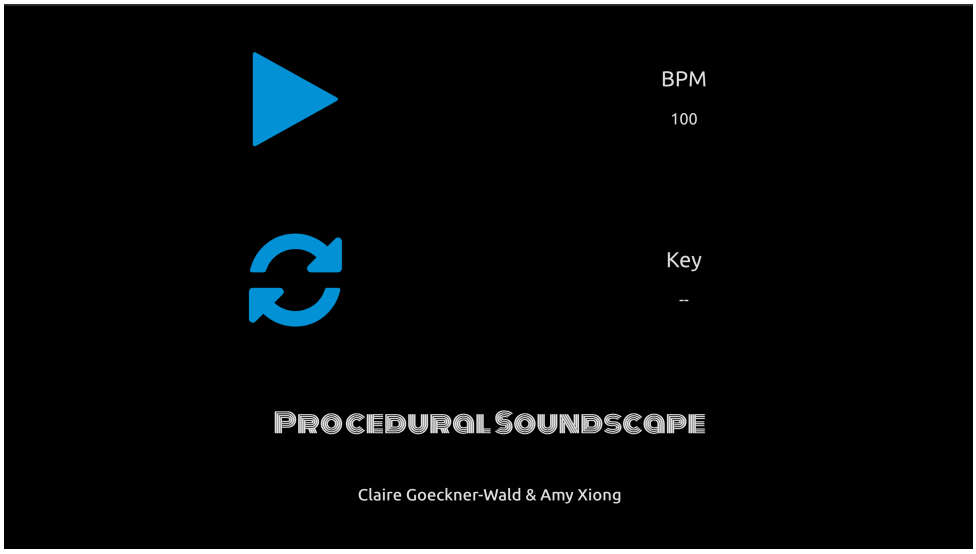

Figure 2: The pre-interaction screen as seen on Firefox Quantum 63.0. This screen occurs after loading is finished, but before the user has interacted with the interface. To begin, users can interact with the play/pause button and the reload/refresh button.

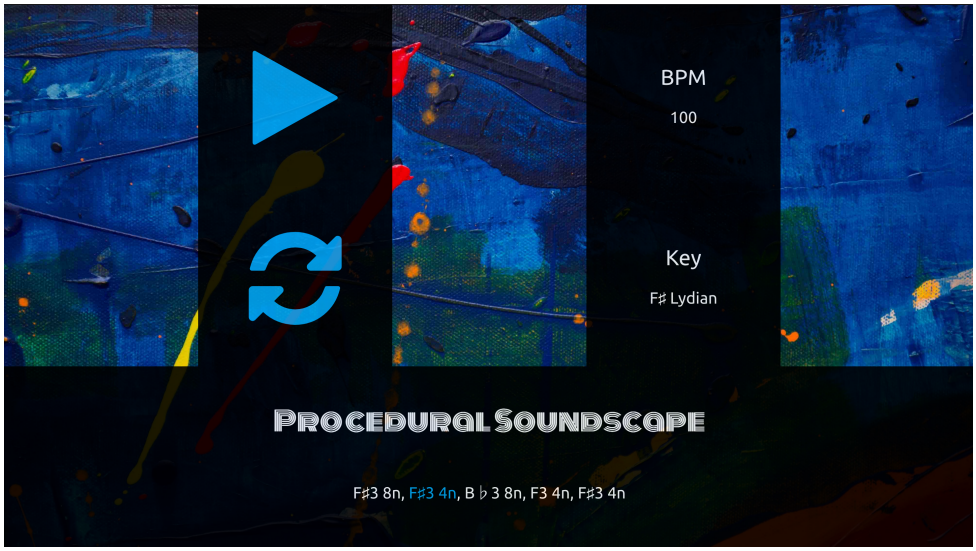

Figure 3: The post-interaction user interface as seen on Firefox Quantum 63.0. Users can interact with the play/pause button and the reload/refresh button.

2.4.1 Actions

There are two actions a user make take: toggle play-pause button or toggle refresh button. Toggling the play-pause button functions exactly as expected - if the music is playing, it stops the music. If the music is not playing, it restarts the music exactly where it left off (or, at the beginning, if toggled for the first time). Toggling the refresh button updates the background, BPM, and key. It necessarily restarts the music, as well.

The first time that either the play-pause button or refresh button is toggled, the program initializes a background, BPM, and key. These three qualities are *unlinked*, meaning they change independently of each other. We considered using a `constants.json` file to build dependencies between the backgrounds and the type of music. However, given the large variety of backgrounds we wished to use, we found that this functionality would be cumbersome without adding significant value to the project.

The program also features a count-down because we generate notes a full measure in advance (so that we may display them on screen). Without this extrapolative generation, we may encounter an issue with `Tone.js` where the first note is not scheduled in advance of being played, causing the first note to be 'dropped,' or not played at all.

2.4.2 Displays

There are 3 different kinds of dynamic information displays to the user: the notes, the BPM, and the key.

The notes are scheduled a full measure in advance, which allows us to display them on screen. The function `emphasizeNote()` uses the index of the currently sounded note to updated the displayed HTML. In this way, the user can visually track the notes (or rests!) as they are sent live to `Tone.js`.

2.4.3 Backgrounds

The project features approximately 50 unique backgrounds (the number fluctuates when we delete one we don't like). The backgrounds were selected for their flat, textured appeal from Unsplash [14]. Unsplash was used for its selection of "Over 550,000 free (do-whatever-you-want) high-resolution photos brought to you by the world's most generous community of photographers," with a very generous license.

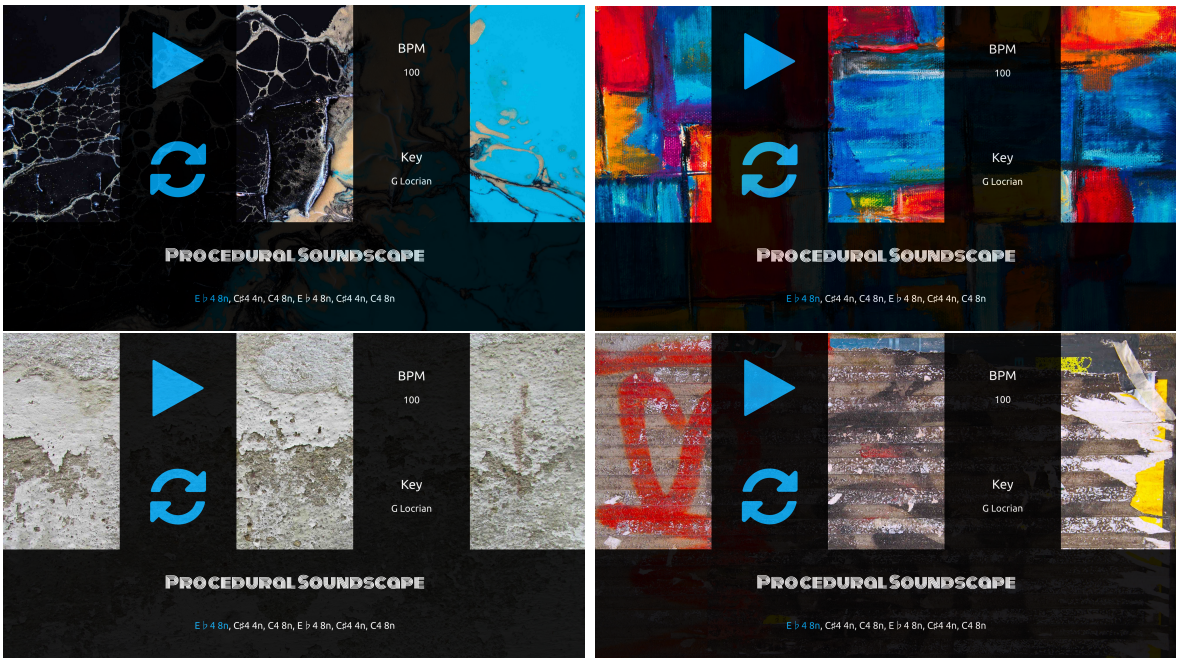

Figure 4: Various backgrounds.

2.5 File Directory Structure

```
procedural-soundscape
├── assets
│   ├── backgrounds
│   │   ├── bg0.jpg
│   │   ├── bg1.jpg
│   │   ├── ...
│   │   └── bgN.jpg
│   ├── css
│   │   └── main.css
│   ├── js
│   │   ├── chords.js
│   │   ├── constants.json
│   │   ├── instruments.js
│   │   ├── LICENSE-Brosowsky.md
│   │   ├── LICENSE-Mann.md
│   │   ├── LICENSE-Wheeler.md
│   │   ├── main.js
│   │   ├── music.js
│   │   ├── rhythms.js
│   │   ├── Tone.js
│   │   └── Tonejs-Instruments.js
│   └── samples
│       ├── bass-electric
│       ├── bassoon
│       ├── cello
│       ├── clarinet
│       ├── contrabass
│       ├── flute
│       ├── french-horn
│       ├── guitar-acoustic
│       ├── guitar-electric
│       ├── guitar-nylon
│       ├── harmonium
│       ├── harp
│       ├── organ
│       ├── piano
│       ├── saxophone
│       ├── trombone
│       ├── trumpet
│       ├── tuba
│       ├── violin
│       └── xylophone
├── latex
│   ├── report.tex
│   ├── report.pdf
│   └── references.bib
└── index.html
```

```
├── README.md
└── LICENSE.md
```

2.6 Codebase

The entire codebase (and the original `LaTeX` files for this very report) can be found on Claire's GitHub repository: github.com/cgoecknerwald/procedural-soundscape. The web application is live at cgoecknerwald.github.io/procedural-soundscape/.

2.7 Workflow

Rather than using Google App Engine, or similar, we simply hosted the website for free on Claire's personal GitHub site, cgoecknerwald.github.io/procedural-soundscape. This greatly simplified the workflow, as we could easily push HTML/CS/JS to the GitHub repository, and it would then be immediately updated online with no extra work.

- Commit history: one can view the entirety of commits to this project over time.
- Issue tracking: We kept track of bugs and wishful enhancements with the issue tracker. At this time, we have 5 open issues and 8 closed issues.
- Project management: We at times used two Kanban-inspired workflow management boards. One board represented primarily the user interface work, and the other represented primarily the sound design work. In general, however, we relied mostly on verbal in-person communication and issue tracking for workflow.

2.8 Implementation Issues

We looked into playing and modifying audiofiles (such as `MP3`s or `WAV`s) but the integration with Tone.js seemed too complicated [12]. This threw us off track for several weeks as we searched for good synths to use (we found none). Claire had switched to DuckDuckGo during the project, which has a much less 'intelligent' search algorithm, which would not return the repo `tonejs-instruments` from a search for "tone js instruments". Google, however, did, so eventually it was found. In `tonejs-instruments`, they use publicly available `WAV`s from about a dozen instruments, including saxophone and piano (but no drums) [3]. We relied heavily on this repo due to its realistic sound qualities. Originally, we believe that this issue references why we can't use `WAV`-format files in Tone.js. However, that issue seems to be outdated, incorrect, or misread.

We discussed moving instruments synths into `assets/js/instruments.js`, out of `assets/js/main.js` Moving roots/scales/chord progressions/rhythms/min & max on octaves in `assets/js/chords.js`. As the music portion of the site grew more and more complicated, the file grew messier and messier, and it became hard to add to and still understand what was happening. Our `assets/js/main.js` (UI) and `assets/js/music.js` (music generation) files were actually originally one single file, since the UI and the music had to be synced together. This split was one of the first big modularization changes we made. We also took the time to break up `assets/js/music.js` into separate functions, to improve readability and comprehension. This modularization had its own problems, in that the functions had to share a lot of data between each other and with the UI. This data sharing problem was eventually solved

somewhat satisfactorily through a combination of function static variables, refactoring so fewer functions needed the same data, and exported, effectively-global variables.

We had a variety of other implementation challenges, some of which were touched in the Musical Review section. The bulk of the challenges came from scheduling all the events at the correct time. We wanted to schedule measures one measure ahead of time, for example, to resolve a bug where sometimes the first note of the measure would be dropped. This meant that we needed to include a countdown for the first measure after the user pressed the play button, but more importantly, it required careful work to synchronize the UI. We have a display of the notes that are currently playing, with the current note highlighted. We had to think carefully about where in the code to trigger the updates of the UI, given that what is being displayed is not what was most recently generated. Instead, what is displayed on the UI is what was generated last measure, and what was most recently generated is the next upcoming measure.

2.9 Licensing

As all open-source contributors should be, we were careful to correctly attribute credit (and liability!) to repositories and sources we borrowed from. We relied on one repository, by Github user `wheelibin`, that did not have a license. As Github-hosted *Choose A License*, hosted and run by GitHub, was very helpful in guiding us (and `wheelibin`) through the licensing process. Failure to license an open-source project can lead to unsavory legal troubles.

> When you make a creative work (which includes code), the work is under exclusive copyright by default. Unless you include a license that specifies otherwise, nobody else can copy, distribute, or modify your work without being at risk of take-downs, shake-downs, or litigation. Once the work has other contributors (each a copyright holder), "nobody" starts including you.
> ...
> If you find software that doesn't have a license, that generally means you have no permission from the creators of the software to use, modify, or share the software. Although a code host such as GitHub may allow you to view and fork the code, this does not imply that you are permitted to use, modify, or share the software for any purpose. [6]

Thankfully, `wheelibin` was responsive to our request via GitHub's issue tracker to license his repository. He ended up using the MIT license by adding it to his `README.md` [15]. Once he had done so, we were free to use and modify his work, so long as we included the original license along with our distribution.

3 Analysis & Conclusion

3.1 Original Goals

- Notably, we did not integrate with any live input (but I suspect it would take less than a full day to integrate with the camera, grab some value from the camera (eg: mean pixel color), linearly scope it to a 'temperature'-type value, and apply said temperature as a variable to any of our musical-type variables or functions.)
- We did not quite achieve the coffeehouse jazz aesthetic, but I believe we could, given more time to experiment.

3.2 Regrets

- Immediately find `tonejs-instruments`. We spent a lot of time looking for and trying to create good synthetic instruments. That time could've easily been saved if we had found `tonejs-instruments` from the beginning.
- Construct a clear, more well-defined end goal. We had many ideas of what we wanted to thought we might want to include in the final product, but didn't precisely define which of these we for sure wanted to include. As a result, we weren't always synced up on what our immediate goals were for the project, which sometimes resulted in wasted effort.
- Plan out music generation more carefully from the beginning. We sort of just started trying things out to see if they would work, then building on top of it. As our idea for the project changed, this sometimes required reconsidering the entire structure of how music generation system.

In general, we felt that we had taken appropriate steps throughout.

3.3 Continuation

- Flesh out the soundscape to make it more musical.
- Add more instrument lines (e.g. a harmony instrument).
- Implement phrasing as best as possible.
- More sophisticated repeat mechanisms (for instance, repeating a section, but transposing it up or down a few notes from the original).
- Integrate chords and chord progressions.
- Allow more parameters to vary from refresh to refresh (for example, changing the random chances of repeats, runs, etc.).
- Preload background images to avoid white flashing.
- Add more pleasant, fading background image transition.

- Fix the saxophone from `tonejs-instruments`. Currently, it gets a wispy / airy / reedy tone in the upper ranges, such that you can hear the reed from the instrument. This is probably caused by a high-pitched soundfile that is interpolated to the intermediate pitch range. Ideally, we would remove the soundfile that contains this wispiness, and `tonejs-instruments` would successfully extrapolate pitches without it.
- Add a music notation system using standard bar notation. This can be done with moderately complex CSS and JS implementation, as demonstrated here on Codepen. [9]
- Spend more time trying to get `Tone.js`'s `MetalSynth` function to work appropriately. At the beginning, we seemed to have a perfectly fine synths for `Kick`, `HiHat`, `OpenHat`, and `DampedOpenHat` functionality. However, it either broke due to some yet-unidentified internal conflict (such as a clash between two `Tone.js` implementations), or we simply had much better taste in synths by the time we began to use it in our music. Either way, we were not satisfied enough to use the synth.

4 Contact

Claire Goeckner-Wald (claire@caltech.edu) or Amy Xiong (axiong@caltech.edu).

5 Appendix

5.1 Original Proposal

The original proposal, submitted Monday, 22 October 2018:

> We aim to build a procedurally-generated soundscape inspired by 'coffeehouse jazz'. The ideal end-product is jazz-esque music that can respond in live time to environmental variables. We selected jazz as our target because we believe it will be easier to procedurally generate due to the improvisational and diverse nature of jazz. We will initially follow this Procedural Music Generation tutorial. To begin, we select a 'palette' of frequencies and sounds that function well together. Then, we will add rhythm and beat to generate our base product. After this is accomplished, we may add additional complexities such as palette-changes, instrument modifications, etc. We may additionally attempt to link these complexities to environmental variables for an interactive soundscape.
>
> We will use SuperCollider for the sound synthesis and algorithmic composition. We will begin with MdaPiano for a piano synthesizer. We will also add drums, saxophone, bass, and other instruments, as appropriate. Ideally we will have large suite of instrument synthesizers. We have considered creating a web application for this project, a la Soft Murmur and similar websites, which allow the client to modify the mixture directly. If we chose to do this, we might use Google App Engine.

5.2 MIT License

The MIT License, in its most frequent form:

> Copyright (c) ⟨year⟩ ⟨copyright holders⟩
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. [2]

5.3 The GNU General Public License v3.0

The GNU GPL v3, in its most frequent form (notably, the full text of the license is online):

> ⟨one line to give the program's name and a brief idea of what it does.⟩ Copyright (C) ⟨year⟩ ⟨name of author⟩
>
> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see ⟨https://www.gnu.org/licenses/⟩. [1]

References

[1] GNU General Public License.

[2] MIT License.

[3] Nicholaus P. Brosowsky. tonejs-instruments. `https://github.com/nbrosowsky/tonejs-instruments`, 2018.

[4] Colin Clark. Flocking. https://github.com/colinbdclark/Flocking, 2018.

[5] Keith Horwood. audiosynth. https://github.com/keithwhor/audiosynth, 2014.

[6] Choose A License. No license. https://choosealicense.com/no-permission/, 2018.

[7] Allen Mathews. The 8 most common rhythms (and how to simplify tricky rhythms). https://classicalguitarshed.com/8-common-rhythms/, 2017.

[8] James McCartney. Supercollider,. https://supercollider.github.io/, 2018.

[9] Lavi Perchik. Css musical notes. https://codepen.io/laviperchik/pen/mIACq.

[10] Miller Puckette. Pure Data. https://github.com/pure-data/pure-data/, 2018.

[11] Alexander Strong and Kevin Karplus. Digital synthesis of plucked string and drum timbres. *Computer Music Journal*, 8:43–55, 1983.

[12] Srgy Surkv. Add to tone.js real instruments for music applications. https://github.com/Tonejs/Tone.js/issues/290, 2018.

[13] Tonejs. Tone.js. https://github.com/Tonejs/Tone.js, 2018.

[14] Unsplash.com. Unsplash. https://unsplash.com/, 2018.

[15] Jon Wheeler. synaesthesia. https://github.com/wheelibin/synaesthesia/commit/5d27951bef5078c72f9c73a65983374879c9c920, 2018.