BIMM [LONDON]

Student ID:
1090817

[Music and Sound Production]

[8]
2023/2024

[LN/EMP604X/23]
[A1]

Word Count:
WORDS 2198

I've developed a web-based step-sequencer, hosted at https://dark-step-a013f.web.app/, using Tone.js, React.js, and Next.js. React.js automates updates/data management, and Next.js enables modular organization of the application for improved manageability.

My project has the following features:

- A module-based step-sequencer consists of a grid with 256 buttons, comprising 16 steps and 16 notes for each step. There are 8 blocks of 256 buttons for each instrument, and these can be enabled or disabled to create complex arrangements. ***(Figure.1)***

- The step-sequencer features four instruments, each with their own 8 blocks:

    - A bass guitar, offering one octave of short notes and one octave of longer notes.
    - A chord synth that plays various types of chords.
    - A bell-type synth with different oscillator wave options and ADSR controls. ***(Figure.2)***
    - A drum kit that can switch between 808 electronic drums and live drums.

- Additionally, there is a playable synth where dragging the mouse or your finger across the pad controls the note (x-axis) and the cutoff (y-axis). ***(Figure.5)***

- A granular sampler that allows for the upload of an audio file, enabling control over playback speed, grain size, granular overlap, randomness, detune, and pitch drift. ***(Figure.3/4)***

- The mixer panel includes volume controls for all instruments and a variable tempo slider.***(Figure.6)***

The inspiration for this project comes from the logic step sequencer, ***(Apple Inc., 2023)*** the Ample sound bass synthesizer which uses bass samples to create a virtual instrument ***(Ample Sound, no date)***, and the Jupiter 8 hardware synth that inspired and contextualized the synthesizer instruments.***(Roland Corporation. 1981)***

When I first tried to create the sequencer clock, I used the JavaScript setInterval() method which allows you to start a clock, and then change the current step to the next step at a timing interval. This did not work, further research showed that setInterval() relies on the JavaScript runtime environment's single-threaded event loop, so it is subject to non-deterministic execution times, especially under CPU load, leading to inconsistent and unreliable timing for triggering musical events. ***"JavaScript is cooperatively single-threaded, so interrupt callbacks execute atomically and do not preempt each other." (Mickens, Elson, and Howell, 2010)*** This led me to read about the WebAudio Api which allows you to perform said tasks more reliably ***(Tsuchiya, Freeman, and Lerner, 2016)*** and Tone.js which is a library of premade WebAudio Api objects. ***(Tone.js Documentation, 2024)*** Tone.js has a transport object which initiates a clock, you provide it with a bpm value and you can schedule events to repeat on set intervals. ***(Rajora, 2024)*** I set each 16th note to trigger the current step to change. This was reliable and allowed me to perform visual updates to the GUI on each step.

```
Tone.Transport.bpm.value = tempo;
Tone.Transport.timeSignature = 4;
Tone.Transport.scheduleRepeat((time) => {
  if (Tone.Transport.seconds === 0) {
    setActiveStep(0);
  } else {
    setActiveStep((prevStep) => (prevStep + 1) % steps);
  }
}, "16n");
```

I then created a function and a button to play/pause the transport.

```
const togglePlayback = () => {
  setPlaying(!playing);
  if (!playing) {
    Tone.Transport.start();
    sequencerCore.current.start(0);
  } else {
    Tone.Transport.stop();
    setActiveStep(0);
    setActiveSection(0);
  }
};
```

Initially, I tried to trigger notes directly within the transport scheduleRepeat function, yet this proved too intensive for scheduleRepeat to handle and only worked with one or two notes selected before auditory artifacts appeared.

I reviewed scheduling techniques in the Tone.js documentation, which suggested that pre-scheduling notes could optimize audio buffer performance. ***(Tone.js Documentation, 2024)*** However, early attempts introduced artifacts.
I utilized Tone.Part in Tone.js to efficiently schedule and adjust musical events using transport time notation like 0:0:1 or 0:1:2, synchronized with Tone.Transport. Tone.Part maintains a precise timeline of events, each with a specified time and callback, ensuring accurate execution during active sessions. ***(Tone.js, 2024)***

Tone.Part also supports looping, ideal for repetitive sequences. ***(Tone.js Documentation, 2024)*** I developed a function to adjust the Tone.Part length based on active "sections" — groups of 16th notes that can be toggled on or off, allowing complex rhythm configurations in my sequencer with up to 8 adjustable sections. This feature dynamically manipulates the sequence structure for intricate musical arrangements.

```
const initSequencer = () => {
    sequencerCore.current = new Tone.Part((time, value: any) => {
      switch (value.instrument) {
        case "polySynth":
          polySynthRef.current.triggerAttackRelease(value.note, "8n", time);
          break;
        case "chordSynth":
          chordSynthRef.current.triggerAttackRelease(value.note, "8n", time);
          break;
        case "bass":
          bassRef.current.triggerAttackRelease(value.note, "8n", time);
          break;
        case "drums":
          switch (kit) {
            case "808":
              eightOhEightDrums.current.triggerAttackRelease(
                value.note,
                "8n",
                time
              );
              break;
            case "Live":
              liveDrumsRef.current.triggerAttackRelease(value.note, "8n", time);
              break;
            default:
              console.log(kit, "Drum kit not found");
          }
          break;
        default:
          console.error("Instrument not found");
      }
    }, []);
    sequencerCore.current.loop = true;
    if (sections.length > 1) {
      sequencerCore.current.loopEnd = sections.length.toString() + "m";
    }
    if (elementsToCore.length > 0) {
      elementsToCore.forEach((element) => {
        sequencerCore.current.add(element);
      });
    }
  };
```

The above code shows setting up the Tone.part and instructing how many measures to run for. Tone.part allows you to pass through arguments in the values object. I set up each instrument as a switch case to manage its behavior within the part. ***(Kaity, 2023)*** This follows the command dispatcher pattern often used in programming. ***(Dupire & Fernandez, 2001)*** I can now add or remove notes on the part by providing an instrument, note and time.

```
sequencerCore.current.add({
        time: "0:2:1",
        note: "E3",
        instrument: "bass",
      };)
```

```
sequencerCore.current.remove({
        time: "0:2:1",
        note: "E3",
        instrument: "bass",
      };)
```

Initially, I attempted to remove notes from Tone.Part by constructing a new object and using the .remove method (the example above). However, this approach failed. The documentation stated this implementation should work. ***(Tone.js Documentation, 2024)*** As it did not work I further troubleshooted by inspecting the constructor within Tone.js (Tone.js is open source so I could search the codebase) and found that it compared objects using an equality operator. I discovered that JavaScript's object management system compares objects by memory address rather than content, meaning a note added to Tone.Part can only be removed by referencing the exact memory address it was assigned with.***"The proxy identity is observable with the JavaScript equality operators == and ===: When applied to two objects, ... distinct proxies return false even though the underlying target is the same." (Keil et al., 2015)*** This necessitates storing a copy of the "noteForSequencer" object before adding it to Tone.Part. For removal, I must retrieve and use this same object, effectively managing it by its memory address. This approach ensures precise identification and manipulation of the specific object in memory.

Here is an example of Tone.Part.Remove that works by referencing the exact same memory address.

```
const noteForSequencer = {time: "0:2:1", note: "E3", instrument: "bass"}
sequencerCore.current.add(noteForSequencer)
sequencerCore.current.remove(noteForSequencer)
```

For visual feedback and animation control, notes are additionally maintained in an array for each instrument. Below is the entire function with annotations for adding and removing notes:

```
function changeSelection(index, innerIndex) {
  // Retrieve the current instrument and the element data based on user selection
  const instrument = instruments[currentInstrument];
  const element = {
      time: stepToTimeMap[index.toString()],
      note: instrument.noteActions[innerIndex],
      instrument: instrument.name,
  };
  // Determine if the element already exists in the active elements of the visual
section
  const isThisDeleteOrAdd = activeElements[visualSection].findIndex(
      (el) => el.time === element.time && el.note === element.note && el.instrument
=== element.instrument
  );
  // If the element exists, handle deletion
  if (isThisDeleteOrAdd !== -1) {
      // Remove the element from the active visual section
      const newActiveElements = activeElements;
      newActiveElements[visualSection] = newActiveElements[visualSection].filter((_,
idx) => idx !== isThisDeleteOrAdd);
      setActiveElements(newActiveElements);
      // Adjust the element time with the current section index for core
synchronization
      const sectionCurrentlyEditing = sections.indexOf(visualSection);
      const adjustedElement = {
          time: sectionCurrentlyEditing.toString() + element.time,
          note: element.note,
          instrument: element.instrument,
      };
      // Remove the element from the core sequencer logic
      const originalElement = elementsToCore.find(
          (el) => el.time === adjustedElement.time && el.note === adjustedElement.note
&& el.instrument === adjustedElement.instrument
      );
      if (originalElement) {
          sequencerCore.current.remove(originalElement);
      }
      setElementsToCore([
          ...elementsToCore.filter((el) => el !== originalElement),
      ]);
```

```
      // Update the visual indicators specific to the instrument
      setInstruments((prevState) => {
          let newStyleWatch = prevState[currentInstrument].styleWatch.map((item, idx)
=>
              idx === visualSection ? item.filter((i) => i.index !== index ||
i.innerIndex !== innerIndex) : item
          );
          return {
              ...prevState,
              [currentInstrument]: {
                  ...prevState[currentInstrument],
                  styleWatch: newStyleWatch,
              },
          };
      });
  } else {
      // Handle addition of a new element
      const sectionCurrentlyEditing = sections.indexOf(visualSection);
      const newActiveElement = [...activeElements[visualSection], element];
      const newActiveElements = activeElements;
      newActiveElements[visualSection] = newActiveElement;
      setActiveElements(newActiveElements);
      // Adjust the element time with the current section index for core
synchronization
      const adjustedElement = {
          time: sectionCurrentlyEditing.toString() + element.time,
          note: element.note,
          instrument: element.instrument,
      };
      setElementsToCore([...elementsToCore, adjustedElement]);
      sequencerCore.current.add(adjustedElement);
      // Update the visual indicators specific to the instrument
      setInstruments((prevState) => ({
          ...prevState,
          [currentInstrument]: {
              ...prevState[currentInstrument],
              styleWatch: prevState[currentInstrument].styleWatch.map(
                  (item, idx) => idx === visualSection ? [...item, { index: index,
innerIndex: innerIndex }] : item
              ),
          },
      }));
  }
}
```

The same underlying logic as above was applied to develop a trash button and a copy button.

I addressed a challenge in dynamically adjusting note timings based on active sequencer sections. For instance, with only sections 1 and 6 active, notes from section 1 are timed for the first measure and those from section 6 for the second. Activating another section, like section 2, shifts section 6 notes to the third measure. To manage this, I store notes without measure information (e.g., ":2:3") and developed a function to reset Tone.Part, recalibrate its length, and regenerate full time notations for notes when active sections change. This approach keeps note timing accurate and consistent.

```
//HELPER FUNCTION TO UPDATE SEQUENCER
const updateActiveElements = () => {
  setElementsToCore([]);
  sequencerCore.current.loopEnd = sections.length.toString() + "m";
  sequencerCore.current.clear();
  let tempElementsToCore: any = [];
  sections.forEach((section) => {
    if (activeElements[section].length > 0) {
      activeElements[section].forEach((element: any) => {
        const elementToPush = {
          ...element,
          time: sections.indexOf(section).toString() + element.time,
        };
        console.log("elementToPush", elementToPush);
        tempElementsToCore.push(elementToPush);
        sequencerCore.current.add(elementToPush);
      });
    }
  });
  setElementsToCore(tempElementsToCore);
  sequencerCore.current.stop(0);
  Tone.Transport.stop();
  setActiveStep(0);
  setActiveSection(0);
  Tone.Transport.start();
  sequencerCore.current.start(0);
};
//UPDATE SEQUENCER WHEN SECTIONS CHANGE
useEffect(() => {
  if (!isMounted) {
    setIsMounted(true);
  } else {
    updateActiveElements();
  }
}, [sections]);
```

I created a simple function to manage actions (select, remove and add) on sections when they are clicked.

```
onClick={() => {
    if (visualSection === index && sections.includes(index)) {
      if (visualSection === 0) {
        return;
      } else {
        let filteredSections = sections.filter(
          (section) => section !== index
        );
        setSections(filteredSections);
        setVisualSection(sections[filteredSections.length - 1]);
      }
    } else if (!sections.includes(index)) {
      const newSections = [...sections, index].sort(
        (a, b) => a - b
      );
      setSections(newSections);
      setVisualSection(index);
    } else if (
      sections.includes(index) &&
      visualSection !== index
    ) {
      setVisualSection(index);
    }
  }}
>
```

After resolving the note activation logic, I set up the instruments: a bass, 808 drum kit, and live drum kit, all using sampled sounds. I recorded the bass samples myself and found the drum samples in an old folder. Each sample was processed to normalize volume and improve tonal quality. This involved gain adjustment, Izotope RX (to de-noise the

recorded bass notes) dynamic range compression, equalization for frequency balance, and saturation to enhance warmth and harmonic detail. This ensured uniformity across all samples for consistent sound quality.

```
bassRef.current = new Tone.Sampler({
  urls: {
    C1: "E2.wav",
    D1: "F2.wav",
    ... more notes
    },
  baseUrl: "/audio/bass/",
  onload: () => {
    setIsBassLoading(false);
  },
});
```

In my implementation, I initialized a Tone.Sampler object, assigning audio samples to specific notes, albeit with unconventional note mapping due to an encountered issue. Tone.Sampler under the hood manages audio buffering and playback, which in a manual setup using the WebAudio Api would require explicit creation and management of AudioBuffer and AudioBufferSourceNode for each sample. ***(Matuszewski & Rottier, 2023)*** These elements handle the decoding and playback of audio data respectively, a complex process simplified by Tone.Sampler. ***(Tone.js Documentation, 2024)***

Recognizing the challenges of audio file loading, especially on mobile networks, I implemented a state monitoring system for each instrument. This setup utilizes the onload callback of Tone.Sampler to update the UI once all instrument's samples are fully loaded. This precaution ensures that playback does not commence before all audio files are properly loaded, thereby preventing potential playback errors. The same process was applied to both drum kits.

```
eightOhEightDrums.current = new Tone.Sampler({
  urls: {
    C1: "Kick Short.wav",
    ...More notes
  },
  baseUrl: "/audio/808-drums/",
  onload: () => {
    setIsEightOhEightDrumsLoading(false);
  },
});
liveDrumsRef.current = new Tone.Sampler({
  urls: {
    C1: "Kick 1.wav",
    ...More notes
  },
  baseUrl: "/audio/live-drums/",
  onload: () => {
    setIsLiveDrumsLoading(false);
  },
});
```

I utilized the Tone.js Tone.PolySynth constructor for the bell synth, which supports polyphony by creating multiple instances of a specified monophonic synth - Tone.Synth in this case. This approach significantly simplifies development, as Tone.js automates the management of multiple synthesizer instances and handles intricate configurations that would otherwise require manual implementation using the WebAudio Api directly.***(Tone.js Documentation, 2024)*** Without Tone.js, creating a monophonic synth would involve manually establishing oscillators, linking them through volume envelopes, and coding methods to alter ADSR and oscillator types. ***(Eriksson, 2013)*** To make it polyphonic would mean creating a method to duplicate the monophonic synth when extra notes are needed and handle cleanup, a process that Tone.js streamlines effectively.

Tone.js allows me to manipulate the oscillator ADSR in any and all ways so to implement the aforementioned myself would be considered an anti-pattern in software development. There's no need to reinvent such fundamental components when they have already been efficiently implemented by existing libraries. ***(Ramirez Lahti, 2020)***

The oscillator and envelope settings for this synth are variable, and are controlled from the synths settings menu.

```
const initBellSynth = () => {

  polySynthOptionsRef.current = {
    oscillator: {
      type: polySynthOscillatorType,
      phase: 0,
      detune: 0,
    },
    envelope: {
      attack: polySynthADSR[0],
      decay: polySynthADSR[1],
      sustain: polySynthADSR[2],
      release: polySynthADSR[3],
    },
    volume: -18,
    portamento: 0.05,
  };

  polySynthRef.current = new Tone.PolySynth(
    Tone.Synth,
    polySynthOptionsRef.current
  );
;
```

Tone.js has effects built in that I used, such as delay and reverb. I used the implementation of Freeverb (an open source reverb algorithm) for the reverb. I would have enjoyed implementing at least one effect from scratch with more time. For example, Freeverb uses (Schroeder) filters that scatter phase reflections to smooth echo density, and comb filters create decaying echoes through a feedback loop with a low-pass filter to mimic environmental high-frequency absorption. If time had permitted, I could have explored making my own reverb, perhaps by creating a unique feedback delay network (FDN) that uses interconnected delay lines and feedback loops to produce a dense, diffuse reverberation, simulating various acoustic environments. ***(Välimäki et al., 2012)(Schlecht, 2020)***

```
polySynthVolumeRef.current = new Tone.Volume();
    polySynthPingPongDelayRef.current = new Tone.PingPongDelay("4n", 0.5);
    polySynthPingPongDelayRef.current.wet.value = 0.7;
    polySynthVolumeRef.current.volume.value = -18;
    polySynthFreeverbRef.current = new Tone.Freeverb({
      roomSize: 0.97,
      wet: 0.8,
    });
    polySynthRef.current.chain(
  polySynthPingPongDelayRef.current,
  polySynthFreeverbRef.current,
  polySynthVolumeRef.current,
  polySynthMasterVolume.current,
  masterVolume.current,
  Tone.Destination
);
```

Above shows how I initiate these effects, with values such as room size and wet. I then route the synth through these effects using Tone.chain which allows you to dictate the signal flow of the digital audio. The synth also gets routed through the volume values from the mixer. I attempted to add more effects to the mixer but doing it in the same way caused buffer overloads, this would take refactoring/rethinking most of the logic to fix.

All the other instruments get routed in the same way.

For the playable synth pad, instead of scheduling notes I create a monophonic synth and use the triggerAttack method instead of triggerAttackRelease. I use Tone.now as the time parameter and the note is dictated by where on the x axis is clicked. On mouse move/touch move the current note is accessed and changed, by using portamento this allows the notes to slide. On mouse/touch end actions use the triggerRelease method to stop the note. I also route this through a low pass cutoff filter that's frequency is determined by the y-axis. To make the synth more responsive and feel more interactive I also scale effects such as reverb, delay and chorus vertically the longer the mouse is held down. Below is how I handled mouse actions, touch actions (mobile) were handled differently but produce the same results.

```
const handleMouseDown = (e: any) => {
    e.preventDefault();
    if (playableDivRef.current !== null) {
      const filterFrequency = mapYPositionToFilterFrequency(
        e.nativeEvent.offsetY,
        playableDivRef.current.offsetHeight
      );
      note.current = calculateNote(
        e.nativeEvent.offsetX,
        playableDivRef.current.offsetWidth
      );
      lowpassFilter.current.frequency.linearRampToValueAtTime(
        filterFrequency,
        Tone.now() + 0.2
      );
    }
    if (Tone.now() >= 0) {
      playableSynthRef.current.triggerAttack(note.current, Tone.now());
    }
    currentStartTime.current = Tone.now();
    ... init effects
    intervalRef.current = setInterval(() => {
      ... scale effects
    }, 50);
    ... animation stuff
  };
  const handleMouseUp = () => {
    playableSynthRef.current.triggerRelease();
    if (intervalRef.current) {
      clearInterval(intervalRef.current);
      intervalRef.current = null;
```

```
  }
};
const handleMouseMove = (e:any) => {
  const filterFrequency = mapYPositionToFilterFrequency(
    e.nativeEvent.offsetY,
    playableDivRef.current.offsetHeight
  );
  lowpassFilter.current.frequency.linearRampToValueAtTime(
    filterFrequency,
    Tone.now() + 0.2
  );
  if (e.buttons !== 1) return;
  const newNote = calculateNote(
    e.nativeEvent.offsetX,
    playableDivRef.current.offsetWidth
  );
  ... animation stuff
};
```

The granular sampler operates by first providing a file upload interface that stores an uploaded audio file into an audio buffer. The Tone.GrainPlayer internally manages the dissection of audio samples into small segments, or "grains," and their subsequent playback. ***(Tone.js Documentation, 2024)***

The granular synthesis process handled by Tone.GrainPlayer involves:

- **Overlap:** Determines the degree to which grains overlap each other, affecting the density and texture of the sound.
- **Detune:** Alters the pitch of individual grains, measured in cents, to create subtle pitch variations.
- **Random:** Introduces randomness in the starting point of each grain within the sample, contributing to a more complex and evolving sound texture.
- **Drift:** Modifies the grain playback speed over time to achieve a shifting, dynamic sound.
- **Playback Speed:** Controls the rate at which the grains are played back, impacting the perceived speed and pitch of the sound.
- **Grain Size:** Sets the duration of each grain, influencing the smoothness or choppiness of the resultant audio texture.

I enable the user to adjust these values with the UI.

```
setIsLoading(true);
const blobUrl = URL.createObjectURL(samplerSample);
granularSynthRef.current = new Tone.GrainPlayer({
      url: blobUrl,
      grainSize: 0.2,
      overlap: 0.1,
      playbackRate: 1,
      reverse: true,
      detune: 0,
      random: 0,
      drift: 0,
```

```
        onload: () => {
            console.log("Sample loaded");
        },
} as any);
setIsLoading(false);
```

I created a Tone.Analyser in "waveform" mode with 1024 samples to capture and visualize real-time audio amplitude changes by connecting it to the master output and drawing the waveform onto a HTML canvas.

```
analyserRef.current = new Tone.Analyser("waveform", 1024);
masterVolume.current.connect(analyserRef.current);
    const canvas = canvasRef.current;
    const canvasContext = canvas.getContext("2d");
    const WIDTH = canvas.width;
    const HEIGHT = canvas.height;
    const draw = () => {
      animationFrameId = requestAnimationFrame(draw);
      const dataArray = polySynthAnalyserRef.current.getValue();
      canvasContext.clearRect(0, 0, WIDTH, HEIGHT);
      canvasContext.lineWidth = 2;
      canvasContext.strokeStyle = "#EABE6C";
      canvasContext.beginPath();
      const sliceWidth = (WIDTH * 1) / dataArray.length;
      let x = 0;
      for (let i = 0; i < dataArray.length; i++) {
        const v = dataArray[i] / 3 + 0.5;
        const y = v * HEIGHT;

        if (i === 0) {
          canvasContext.moveTo(x, y);
        } else {
          canvasContext.lineTo(x, y);
        }

        x += sliceWidth;
      }
      canvasContext.lineTo(WIDTH, HEIGHT / 2);
      canvasContext.stroke();
    };
    draw();
  };
```

A few examples of how I update musical values:

```
useEffect(() => {
  if (masterVolume.current) {
    masterVolume.current.volume.value = masterVolumeUi;
  }
}, [masterVolumeUi]);
```

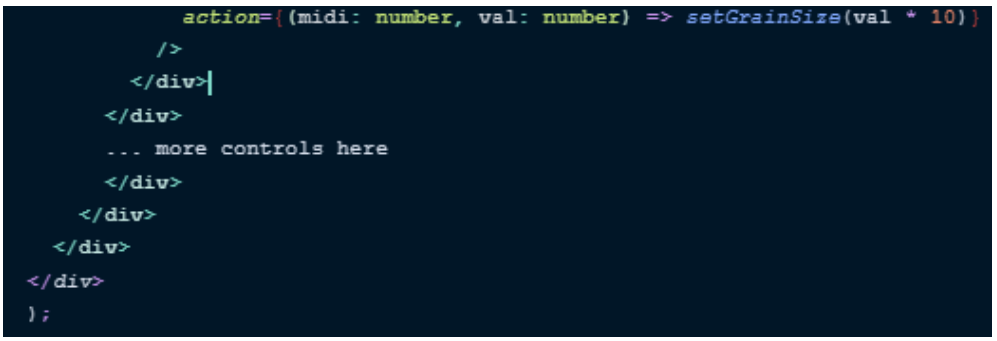
```
useEffect(() => {
  if (masterVolume.current) {
    playableLeadMasterVolume.current.volume.value = playableLeadMasterVolumeUi;
  }
}, [playableLeadMasterVolumeUi]);
```

```
useEffect(() => {
    if (
      granularSynthRef.current &&
      "playbackRate" in granularSynthRef.current
    ) {
      granularSynthRef.current.playbackRate = Math.max(
        playbackRateState,
        0.001
      );
    }
  }, [playbackRateState]);
```

A few examples of how I create the User Interface:

```
<div className="flex w-full justify-center relative space-x-1">
      <div
        ref={playableDivRef}
        className="w-full ... more styles"
        onMouseDown={handleMouseDown}
        ... more actions
      >
        {ripples.map(({ id, x, y }: { id: any; x: number; y: number }) => (
          <div
            key={id}
            className="fade bg-quarternary w-8 h-8 rounded-full absolute "
            style={{ left: x - 15, top: y - 14 }}
          ></div>
        ))}
        {trails.map(({ id, x, y }: { id: any; x: number; y: number }) => (
          <div
            key={id}
            className="burst bg-quarternary w-2 h-2 rounded-full absolute "
            style={{ left: x , top: y }}
          ></div>
        ))}
      </div>
    </div>
```

```
return (
  <div className='w-full h-full flex items-center justify-center'>
  <div className="flex ... more styles ">
    <div className="flex flex-wrap w-full ... more styles">
      {visualizationText.map((spanElement: any, index: number) => (
        <Fragment key={index}>{spanElement}</Fragment>
      ))}
    </div>
    <div className="flex flex-wrap h-1/2 w-full">
      <div className="p-3">
        <div className=" w-8 md:w-16 cursor-pointer">
          <Knob
            diameter="100%"
            color="#F6F1D1"
            pointerColor="#EABE6C"
```



Throughout this project, challenges such as buffer overloads emerged frequently. particularly when I attempted to enhance the mixer beyond basic volume controls and during initial note triggering via the transport in the WebAudio Api. These issues underscore the need for efficient methods in programming. Future updates would focus on optimizing for mobile platforms, extending mixer functionalities, and adding key modulation capabilities. To add mixer effects, such as EQ etc, a refactoring of the audio effects initialization sequence is required to prevent overloads. Additionally, enabling musical key changes would necessitate a redesign of how notes are stored and manipulated, increasing the complexity of the already complex add and remove functions.

Bibliography:

1. *MDN contributors, 2023. Web Audio API. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API [Accessed 16 April 2024].*

2. *Mickens, J., Elson, J. and Howell, J., 2010. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In: NSDI. Available at: https://www.usenix.org/legacy/events/nsdi10/tech/full_papers/mickens-mugshot.pdf [Accessed 20 April 2024].*

3. *Tsuchiya, T., Freeman, J. and Lerner, L.W., 2016. Data-Driven Live Coding with DataToMusic API. In: Proceedings of the Web Audio Conference, Georgia Institute of Technology, Center For Music Technology, pp.1-8. Available at: http://webaudioconf.com/_data/papers/pdf/2016/2016_55.pdf [Accessed 20 April 2024].*

4. *Tone.js Documentation, 2024. Tone.js API Documentation. [online] Available at: https://tonejs.github.io/docs/14.7.77/index.html [Accessed 20 April 2024].*

5. *Rajora, Y., 2024. Composing Music in JavaScript using Tone.js. C# Corner. Available at: https://c-sharpcorner.com/article/composing-music-in-javascript-using-tone-js/ [Accessed 27 February 2024].*

6. *Tone.js, 2019. Events. GitHub. Available at: https://github.com/Tonejs/Tone.js/wiki/Events [Accessed 20 April 2024].*

7. *Keil, M., Guria, S.N., Schlegel, A., Geffken, M. & Thiemann, P., 2015. Transparent Object Proxies for JavaScript. Technical Report. University of Freiburg, Germany. [Accessed 20 April 2024].*

8. *Kaity, P., 2023. The Ultimate Guide to Understanding Switch Cases in JavaScript. Medium. Available at: https://medium.com/@pradipkaity/the-ultimate-guide-to-understanding-switch-cases-in-javascript-b49a2b3927da [Accessed 14 April 2024].*

9. *Dupire, B. & Fernandez, E.B., 2001. The Command Dispatcher Pattern. In: 8th Conference on Pattern Languages of Programs. Available at: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=b6fdf774e0b86a0586855405f109b8bda227cc1c [Accessed 20 April 2024].*

10. *Matuszewski, B. & Rottier, O., 2023. The Web Audio API as a Standardized Interface Beyond Web Browsers. Journal of the Audio Engineering Society, 71(11), pp.790-801. Available at: https://hal.science/hal-04352384 [Accessed 20 April 2024].*

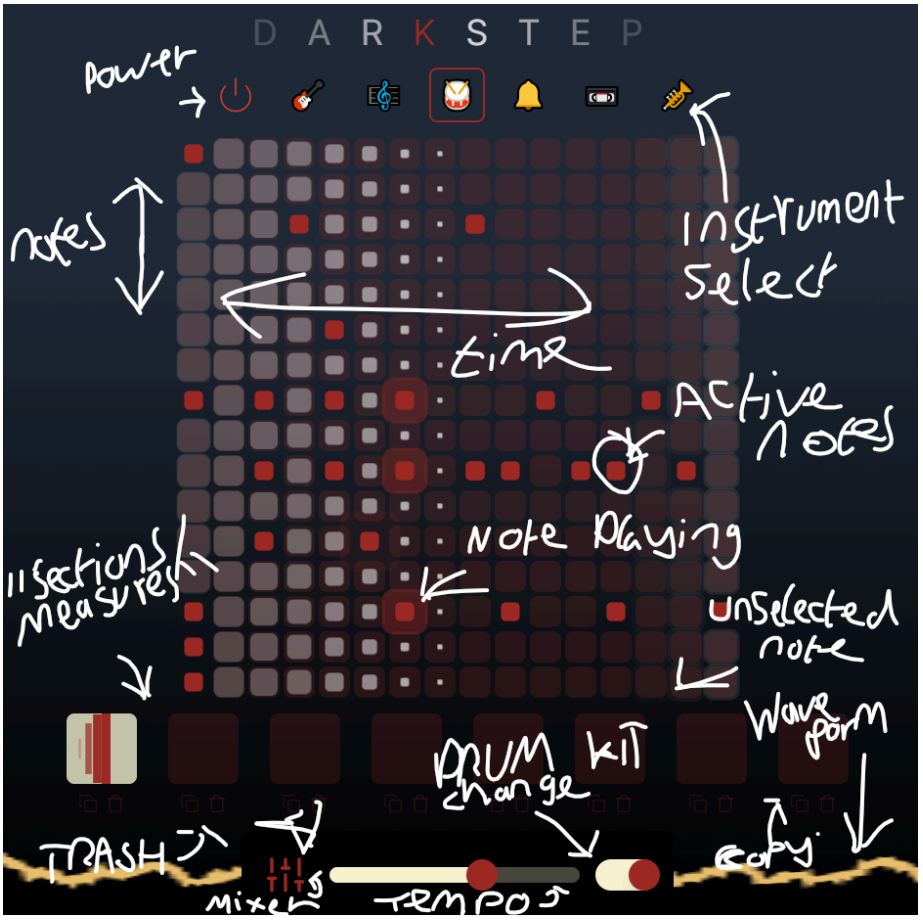
11. *Eriksson, O., 2013. Implementing virtual analog synthesizers with the Web Audio API: An evaluation of the Web Audio API. Bachelor's thesis, Linnaeus University, Faculty of Technology, Department of Computer Science. Available at: https://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-27099 [Accessed 20 April 2024].*

12. *Ramirez Lahti, J., 2020. Reversing Entropy in a Software Development Project: Technical Debt and AntiPatterns. MSc Thesis, University of Helsinki, Faculty of Science, Department of Computer Science. Available at: https://helda.helsinki.fi/server/api/core/bitstreams/f1e19f83-32e7-4618-8615-95002677b44d/content [Accessed 20 April 2024].*

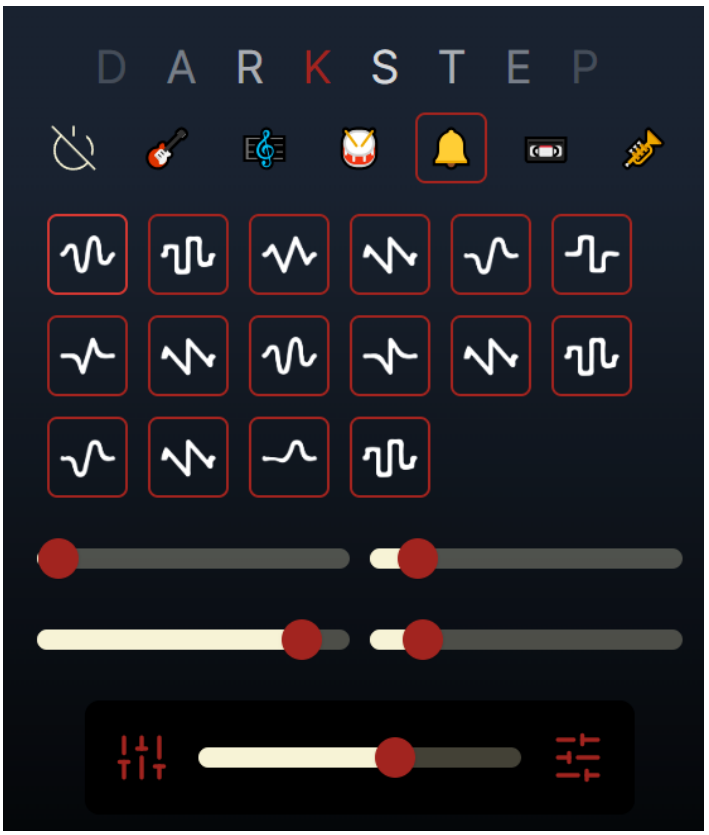
13. *Välimäki, V., Parker, J.D., Savioja, L., Smith, J.O. & Abel, J.S., 2012. Fifty Years of Artificial Reverberation. IEEE Transactions on Audio, Speech, and Language Processing, 20(5), pp.1421-1448. Available at: https://doi.org/10.1109/TASL.2012.2189567 [Accessed 20 April 2024].*

14. *Schlecht, S., 2020. FDNTB: The Feedback Delay Network Toolbox. In Proceedings of the International Conference on Digital Audio Effects, pp.211-218. DAFx. Available at: https://dafx2020.mdw.ac.at/proceedings/papers/DAFx2020_paper_53.pdf [Accessed 20 April 2024].*
15. *Apple Inc. (2023) Logic Pro X [Software]. Available at: https://www.apple.com/logic-pro/ (Accessed: 22 April 2024).*
16. *Ample Sound. (no date) Ample Bass P Lite [Software]. Available at: https://www.amplesound.net/en/pro-pd.asp?id=19 (Accessed: 22 April 2024).*
17. *Roland Corporation. (1981) Jupiter-8 Synthesizer [Hardware]. (Accessed: 22 April 2024).*

Appendices:

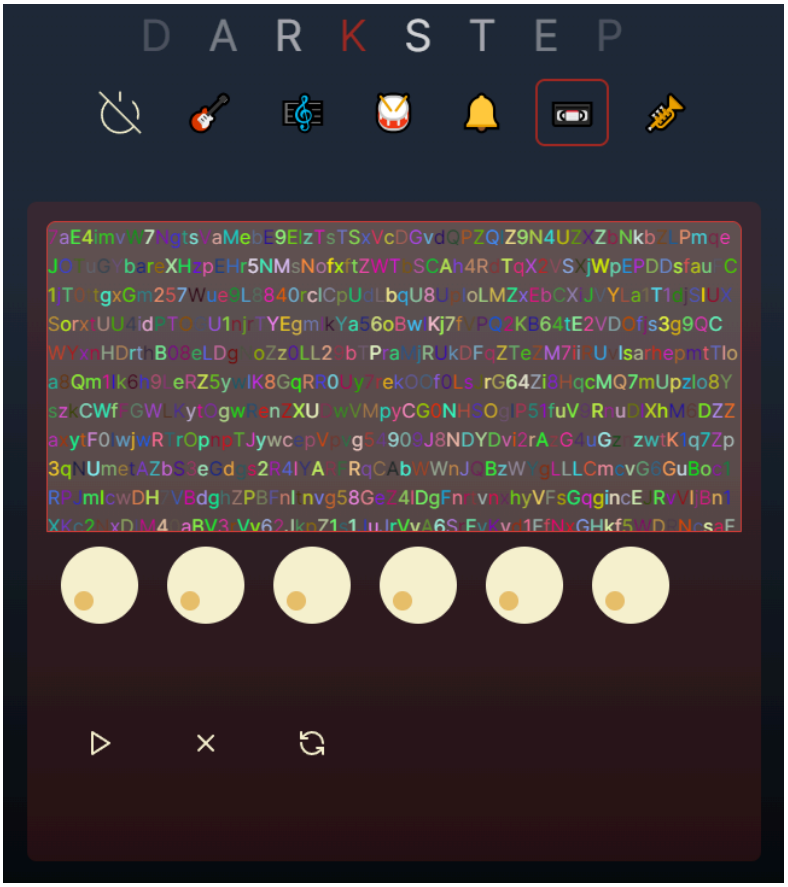


(Figure 1, Step Sequencer Labeled)

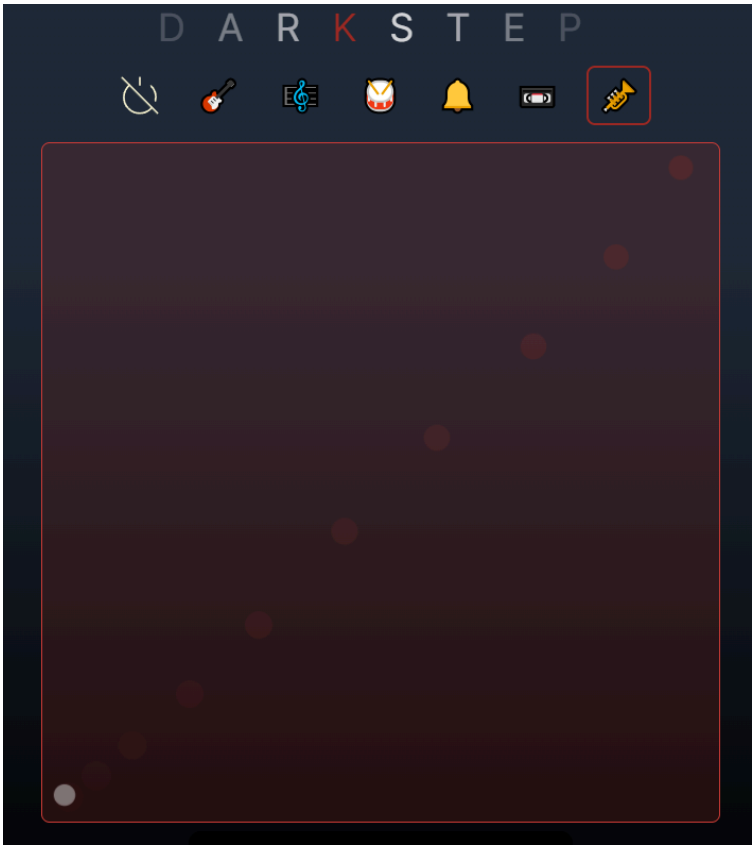


(Figure 2, Bell Synth Oscillator & ADSR)

(Figure 3, Granular Sampler File Upload)



(Figure 4, Granular Sampler Interface)



(Figure 5, Playable Synth pad)

(Figure 6, Volume Mixer)