Собрать звуковую спектрограмму на React и MobX

Привет!

Я Таня, фронтенд-разработчик в KTS и студент магистратуры МГТУ им. Баумана. На одном из недавних проектов я работала над интересной фичей — визуальным представлением аудиоданных, a.k.a. звуковой спектрограммой. Казалось бы, штука нехитрая: кто не видел график, прыгающий в такт с музыкой на разных частотах? Он есть в любом секвенсоре, на любом диджейском пульте и даже в динамическом островке последних айфонов.

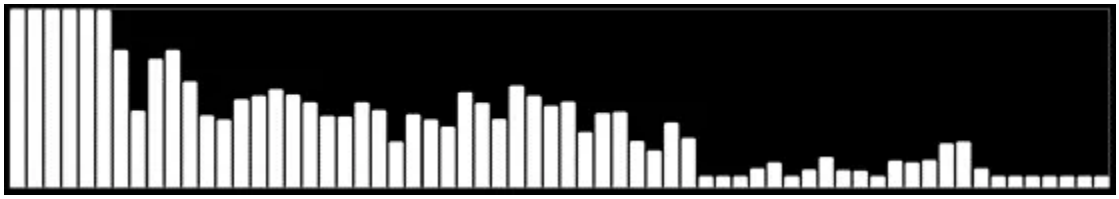
Однако задача оказалась нетривиальной, поскольку для целей проекта мне нужно было разработать звуковую спектрограмму на React и MobX в особом дизайне. Подробных разборов этой темы и готовых решений я не нашла, поэтому в процессе пришлось самостоятельно разобраться с кучей тонкостей и нюансов. Результат можете посмотреть по ссылке.

А в этой статье я расскажу, как сделать такую же звуковую спектрограмму, а также как изменять ее стиль и другие параметры.

Оглавление

Что такое звуковая спектрограмма

Звуковая спектрограмма — это визуальное представление аудиоданных, отображающее изменение интенсивности звука с течением времени на различных частотах. На такой диаграмме горизонтальная ось представляет время, а вертикальная — частоту.



Для обработки аудиоданных существует Web Audio API, который можно использовать непосредственно в браузере. С его помощью я буду получать звук в виде спектров. Этот API предоставляет инструмент AnalyserNode, позволяющий создавать различные звуковые графики. AnalyserNode — это интерфейс в Web Audio API, который позволяет извлекать в реальном времени данные о частоте и времени из аудиопотока, не искажая сам сигнал. Он используется для создания аудиовизуализаций по данным о форме волны или спектре частот.

Я покажу, как получать аудиоданные при записи с микрофоном, хранить их в виде частот и обрабатывать, а затем помещу реализацию в модели и React-компоненты.

Модели записи и отображения звука

Запись звука

В первую очередь необходимо разобраться с записью звука. Чтобы получить данные, когда мы говорим в микрофон, надо иметь доступ к потоку медиаданных (а конкретно к аудиопотоку). Получить доступ можно с помощью запроса getUserMedia:

```
const stream = await navigator.mediaDevices.getUserMedia({
    audio: true,
});
```

Из потока можно получить аудиоданные в виде частот. Как раз для этого воспользуемся AudioContext и при помощи функции createAnalyser создадим анализатор звука:

```
// звуковые параметры
const FFT_SIZE = 128;
const SAMPLE_RATE = 44100;
const TIME_CONSTANT = 0.2;
const MIN_DECIBELS = -90;
const MAX_DECIBELS = -10;

// создание контекста
const audioContext = new AudioContext({ sampleRate: SAMPLE_RATE });
// аудио источник
const mediaStreamAudioSourceNode =
audioContext.createMediaStreamSource(stream);
// анализатор (или передатчик)
const analyserNode = audioContext.createAnalyser();

// заполнение параметров анализатора
analyserNode.fftSize = FFT_SIZE;
analyserNode.smoothingTimeConstant = TIME_CONSTANT;
analyserNode.minDecibels = MIN_DECIBELS;
analyserNode.maxDecibels = MAX_DECIBELS;

// подключаем источник к передатчику
mediaStreamAudioSourceNode.connect(analyserNode);
```

В самом начале я задала аудиопараметры, описание которых можно посмотреть в таблице:

Smoothing Time Constant	**Насколько быстро система сглаживания реагирует на изменения входного сигнала и как долго сохраняет его влияние. Принимает значение от 0 до 1.**
Min Decibels	Минимальный уровень громкости в децибелах, который считается «отсутствием звука».

Max Decibels	Максимальный уровень громкости в децибелах, который может быть зафиксирован.
FFT Size	Размер преобразования Фурье. Указывает на размер выборки. Чем больше FFT size, тем более детальный спектр будет получен (будет больше делений). Как правило используются значения степени двойки.
Sample Rate	Частота дискретизации звука в герцах (Hz). Обычно стандартное значение — 44100 Hz или 48000 Hz.

Для того, чтобы хранить значения частот, создадим массив:

```
// массив частот
const pcmData = new Uint8Array(analyserNode.frequencyBinCount);
```

Поле frequencyBinCount хранит количество доступных частот, и это значение в 2 раза меньше, чем fftSize. С помощью этого поля можно создать массив частот нужной длины (pcmData).

Так как pcmData — это массив Uint8Array, то все его значения находятся в диапазоне от 0 до 255. Я заранее создала дополнительную функцию, чтобы сразу переводить эти числа в проценты (так как в React-компоненте спектрограммы высота каждой частоты будет указана как процент от максмальной длины контейнера, в котором они расположены):

```
export const getAudioPercent = (value: number) => {
     return value / 255 * 100;
}
```

Теперь, когда создан анализатор и массив частот, можно начинать обработку звука. Для того, чтобы записывать данные в pcmData, достаточно вызвать метод getByteFrequencyData. Данные будут обновляться через заданный интервал:

```
const SET_LOUDNESS_TIMEOUT_MS = 1000 / 10;

setInterval(() => {
    analyserNode.getByteFrequencyData(pcmData);
}, SET_LOUDNESS_TIMEOUT_MS);
```

Из массива частот несложно получить громкость:

```
const getCurrentLoudness = (pcmData: Uint8Array): number => {
```

```
    const loudness = pcmData.reduce((acc, amplitude) => {
        if (amplitude === -Infinity) {
            return acc;
        }

        return acc + amplitude * amplitude;
    }, 0.0);

    return getAudioPercent(Math.sqrt(loudness / pcmData.length));
}
```

Для того, чтобы остановить запись звука, нужно сделать следующее:

```
stream.getTracks().forEach((track) => track.stop());
```

Теперь напрашивается создание отдельной модели **AudioAnalyzerModel**, которая бы отвечала за обработку аудиоданных. В ней все изменяемые поля будут храниться при помощи модели FieldModel из библиотеки [mediaproject-stores](). К таким полям относятся:

- _pcmData (массив частот);
- _analyserNode (передатчик);
- currentLoudness (текущая громкость);
- maxLoudness (максимальная громкость, которая была).

У объекта класса FieldModel есть поле value для хранения значения и метод changeValue для изменения значения. С помощью этой модели можно быстро создавать отслеживаемые параметры, не прописывая каждый раз computed, action и observable. Также для интервала потребуется таймер _setLoudnessIntervalId (NodeJS.Timeout). Получится следующий класс:

```
import { FieldModel } from '@ktsstudio/mediaproject-stores';

class AudioAnalyzerModel {
  // массив частот
  private readonly _pcmData = new FieldModel<Uint8Array |
null>(null);
  // передатчик
  private readonly _analyserNode: FieldModel<AnalyserNode>;

  // текущая громкость
  readonly currentLoudness = new FieldModel<number>(0);
  //максимальная громкость
  readonly maxLoudness = new FieldModel<number | null>(null);
```

```
  // интервал
  private _setLoudnessIntervalId: NodeJS.Timeout | null = null;
  ...
}
```

Заполнение параметров _analyserNode перенесем в конструктор (который будет принимать в качестве входного параметра аудио поток stream). Сначала зададим тип входных параметров:

```
// входные параметры для анализатора
type Props = {
    stream: MediaStream;
};
```

Функция конструктора будет выглядеть так:

```
class AudioAnalyzerModel {
  ...
     constructor({ stream }: Props) {
                // создание анализатора
                const audioContext = new AudioContext({ sampleRate:
SAMPLE_RATE });
                const mediaStreamAudioSourceNode =
audioContext.createMediaStreamSource(stream);
                const analyserNode = audioContext.createAnalyser();

                ...
          // заполнение парамеров анализатора
          ...

          this._analyserNode = new FieldModel(analyserNode);

mediaStreamAudioSourceNode.connect(this._analyserNode.value);

          this._pcmData.changeValue(new
Uint8Array(this._analyserNode.value.frequencyBinCount));
      }
  ...
}
```

Создадим функцию, которая будет срабатывать через интервалы времени. Как раз в ней будет обновляться массив частот, а также текущая и максимальная громкости:

```
class AudioAnalyzerModel {
  ...
```

```
    private _analysisFunc = () => {
        // проверка
        if (!this._analyserNode.value || !this._pcmData.value) {
          return;
        }

        // заполнение данными pcmData

this._analyserNode.value.getByteFrequencyData(this._pcmData.value);

        // получение текущей громкости

this.currentLoudness.changeValue(getCurrentLoudness(this._pcmData.val
ue));

        // расчет максимальной громкости
        if (!this.maxLoudness.value || this.maxLoudness.value <
this.currentLoudness.value) {
          this.maxLoudness.changeValue(this.currentLoudness.value);
        }
    };
    ...
}
```

В проекте будет выводиться текущая громкость.

Теперь добавим метод, который бы начал анализ, а также метод, останавливающий его:

```
class AudioAnalyzerModel {
  ...
    analyze(): void {
        this._setLoudnessIntervalId =
             setInterval(this._analysisFunc,
SET_LOUDNESS_TIMEOUT_MS);
    }

    stopAnalysis(): void {
        if (this._setLoudnessIntervalId) {
          clearInterval(this._setLoudnessIntervalId);
          this._setLoudnessIntervalId = null;
        }
    }
    ...
}
```

Итак, мы создали необходимые функции анализатора, которые позволяют обновлять массив частот и отслеживают громкость.

Сейчас мы перешли к этапу, когда нужно уметь начинать и останавливать запись звука, который передается на обработку. Для этого создадим модель **AudioRecorderModel**. В ней понадобится хранить следующие поля:

- isRecording - статус записи (тип: FieldModel<boolean>);
- _mediaStream - поле, хранящее поток (тип: MediaStream | null, по умолчанию равно null);
- _audioAnalyzer - анализатор (тип: AudioAnalyzerModel | null, по умолчанию равно null).

Выйдет следующая модель:

```
class AudioRecorderModel {
  // статус записи
  readonly isRecording = new FieldModel<boolean>(false);

  // аудио поток
  private _mediaStream: MediaStream | null = null;
  // анализатор
  private _audioAnalyzer: AudioAnalyzerModel | null = null;

  ...
}
```

Далее внутри этого класса создадим методы для старта записи (в которой предоставляется доступ к аудио потоку и начинается анализ частот) и остановки записи:

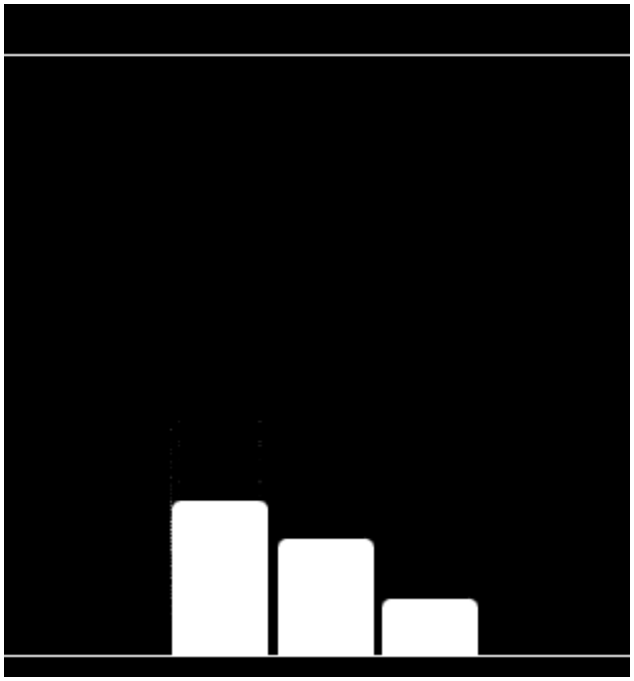
```
class AudioRecorderModel {
      ...
  // начало записи
      startRecording = async (): Promise<void> => {
          // проверка статуса записи
          if (this.isRecording.value) {
              return;
          }

          try {
              // получение потока
              const stream = await
navigator.mediaDevices.getUserMedia({
                  audio: true,
              });
```

```
            // создание анализатора и запись потока
            this._audioAnalyzer = new AudioAnalyzerModel({ stream
});
            this._mediaStream = stream;

            if (!this._mediaStream) {
                return;
            }

            // изменение статуса записи
            this.isRecording.changeValue(true);
            // начало анализа
            this._audioAnalyzer.analyze();
        } catch (e) {
            console.error('Error accessing microphone:', e);
        }
    };

    // остановка записи
    stopRecording = (): void => {
        // проверка статуса записи
        if (!this.isRecording.value) {
            return;
        }

        // остановка анализа
        this._audioAnalyzer?.stopAnalysis();
        this._audioAnalyzer = null;

        // изменение статуса
        this.isRecording.changeValue(false);

        // остановка записи звука
        this._mediaStream?.getTracks().forEach((track) =>
track.stop());
    }
    ...
}
```

Параметры стилей

Итак, теперь у нас есть возможность записывать и анализировать звук. Хотелось бы, чтобы можно было менять спектрограмму, т.е. настраивать разные стили.

Попробуем поменять число делений в спектрограмме. Как было сказано до этого, число спектров может быть только степенью двойки, однако в реальной жизни можно заметить, что иногда бывает, например, всего 3 деления (как при записи звука на видеовстречах):



Значит, деления можно получить, обработав каким-то образом частоты. Остановимся на том, чтобы вычислять среднее значение частот для равных диапазонов, количество которых соответствует нужному числу делений.

Создадим функцию, которая могла бы возвращать массив делений на основе полученной pcmData и желаемого количества делений levelCount:

```
processLevels(pcmData: Uint8Array, levelCount: number): number[] {
    const arrData: number[] = Array.from(pcmData);
    const normalizedValues = arrData.map((value) =>
getAudioPercent(value));

    // длина одного диапазона
    const bins = Math.floor(normalizedValues.length / levelCount);

    // индексы диапазонов
    const rangesIndices: number[][] = Array.from({ length: levelCount
}, (_, i) =>
      Array.from({ length: bins }, (_, j) => i * bins + j)
    );

    return rangesIndices.map((indices) => {
```

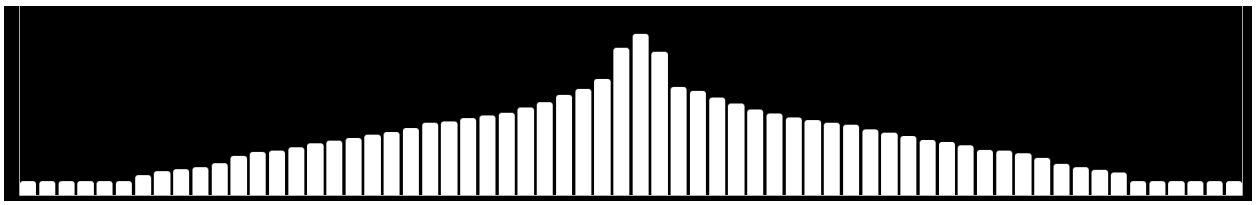
```
      const values: number[] = indices.map((index) =>
normalizedValues[index]);
      const sum = values.reduce((acc, val) => acc + val, 0);

      return values.length > 0 ? sum / values.length : 0;
    });
}
```

Создадим модель BasePositionModel, в которой находилась бы эта функция:

```
class BasePositionModel {
     processLevels(pcmData: Uint8Array, levelCount: number):
number[] {
           ...
     }
}
```

Также можно поменять расположение делений. Для красивого эффекта можно отсортировать самые высокие деления к центру:

```
const baseLevels = processLevels(pcmData, levelCount).sort((a, b) =>
a - b);
const leftLevels: number[] = [];
const rightLevels: number[] = [];

baseLevels.forEach((level, index) => {
      if (index % 2 === 0) {
        leftLevels.push(level);
      } else {
        rightLevels.push(level);
      }
});

const centeredLevels = [...leftLevels, ...rightLevels.reverse()];
```

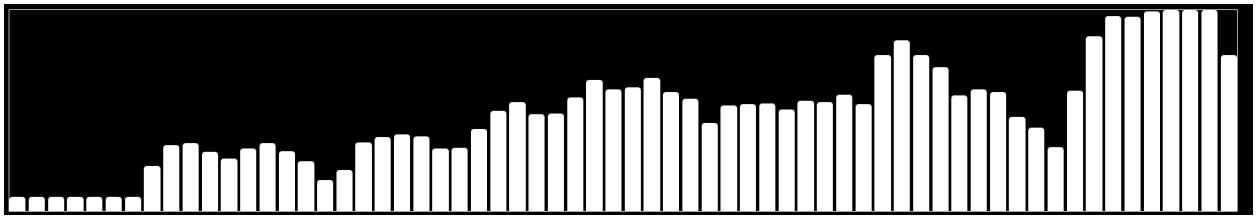


Для применения этого эффекта можно создать отдельную модель CenterPositionModel с наследованием от BasePositionModel и применением метода processLevels модели-родителя:

```
class CenterPositionModel extends BasePositionModel {
     processLevels(pcmData: Uint8Array, levelCount: number):
number[] {
          const baseLevels =
               super.processLevels(pcmData, levelCount).sort((a, b)
=> a - b);

          ...
     }
}
```

Или можно просто перевернуть массив:

```
processLevels(pcmData, levelCount).reverse();
```



Теперь звуковой график не совсем «спектрограмма», но зато красивее)

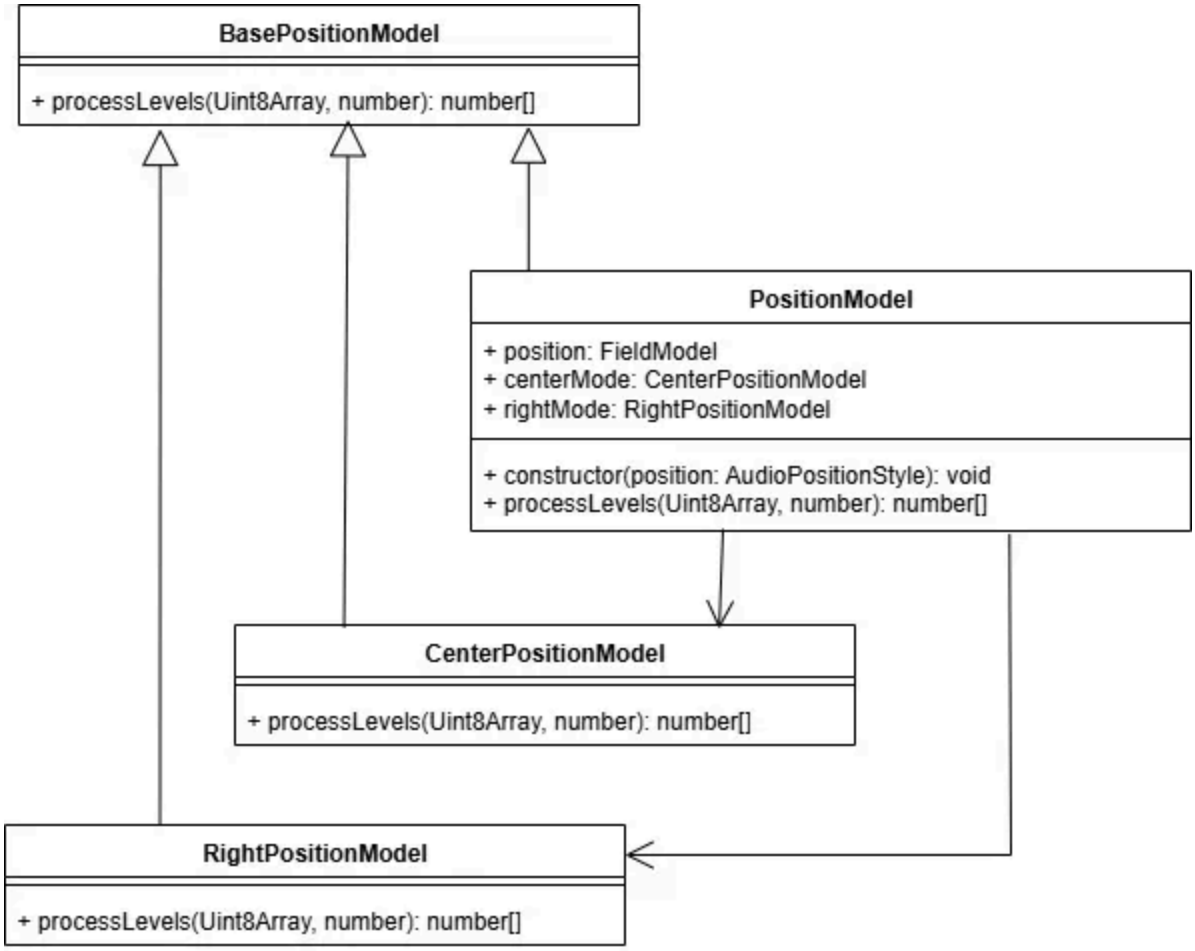
Для того, чтобы хранить настройки стилей, создадим отдельную модель **AudioVisualizerModel**. Заранее определим, что пользователь может сам изменять, чтобы в будущем эти настройки применять в React-компонентах:

1. **Прозрачность** — чем тише громкость, тем прозрачнее соответствующий “спектр” (или деление), и наоборот. За этот параметр будет отвечать поле opacityMode типа FieldModel<boolean>.
2. **Ось** — линия, по которой располагаются деления. За этот параметр будет отвечать поле line типа FieldModel<AudioLineStyle>. Создадим все возможные варианты, которые пользователь может выбрать:
   ```
   export enum AudioLineStyle {
     center = 'center',
     bottom = 'bottom',
     top = 'top',
   }
   ```
3. **Сортировка делений** — как деления будут располагаться на графике.
   ```
   export enum AudioPositionStyle {
     center = 'center',
     left = 'left',
     right = 'right',
   }
   ```

4. **Число делений** — сколько будет отображено делений. Дадим пользователю выбирать несколько вариантов: 3, 64 и 128. Для этого понадобятся следующие типы и конфиги:

```
export enum AudioLevelKeys {
  three = 'three',
  short = 'short',
  big = 'big',
}

export const AVAILABLE_AUDIO_LEVELS: Record<AudioLevelKeys,
number> = {
  [AudioLevelKeys.three]: 3,
  [AudioLevelKeys.short]: 64,
  [AudioLevelKeys.big]: 128,
};
```

Введем в модель **AudioVisualizerModel** поле levels, которое бы соответствовало делениям (массив number[]). Это поле будет сортироваться и менять количество делений внутри (что соответствует описанным выше параметрам 3 и 4). Из кода выше видно, что для этих задач можно сделать отдельную модель: PositionModel с функцией processLevels. Используя наследование, сделаем следующую структуру:



В модели PositionModel также есть метод processLevels, в которой в зависимости от выбранного пользователем вида эффекта будет применяться соответствующая модель (BasePositionModel, CenterPositionModel, RightPositionModel):

```
class PositionModel extends BasePositionModel {
      // выбранный пользователем эффект
  readonly position: FieldModel<AudioPositionStyle>;
  // модель для центрирования делений
  readonly centerMode = new CenterPositionModel();
  // модель, которая располагает все деления справа
  readonly rightMode = new RightPositionModel();

  ...

  processLevels(pcmData: Uint8Array, levelCount: number): number[] {
    const baseLevels = super.processLevels(pcmData, levelCount);

    switch (this.position.value) {
      case AudioPositionStyle.center:
```

```
        return this.centerMode.processLevels(pcmData, levelCount);
      case AudioPositionStyle.right:
        return this.rightMode.processLevels(pcmData, levelCount);
      case AudioPositionStyle.left:
      default:
        return baseLevels;
    }
  }
}
```

Тогда в модели **AudioVisualizerModel** можно добавить поле positionModel типа **PositionModel** и вызывать его метод processLevels при получении pcmData. Окончательный тип параметров выглядит так:

```
type AudioVisualizerProps = {
  levelCount?: number;
  position?: AudioPositionStyle;
  line?: AudioLineStyle;
  opacityMode?: boolean;
};
```

В конечном итоге получится следующая модель, где в конструкторе задаются параметры по умолчанию, если они не были переданы:

```
class AudioVisualizerModel {
  // массив делений
  readonly levels: FieldModel<number[]>;
  // параметр расположения делений
  readonly positionModel: PositionModel;
  // параметр оси
  readonly line: FieldModel<AudioLineStyle>;
  // параметр мода прозрачности
  readonly opacityMode: FieldModel<boolean>;

     // параметры по умолчанию
  constructor({
    levelCount = AVAILABLE_AUDIO_LEVELS[AudioLevelKeys.short],
    position = AudioPositionStyle.left,
    line = AudioLineStyle.bottom,
    opacityMode = false,
  }: AudioVisualizerProps) {
    this.levels = new FieldModel(Array<number>(levelCount).fill(0));
    this.positionModel = new PositionModel(position);
    this.line = new FieldModel(line);
```

```
    this.opacityMode = new FieldModel(opacityMode);
  }

  ...
}
```

Осталось добавить функции, которые бы обновляли массив делений, где как раз будет использован метод processLevels (применяемый, чтобы получить массив делений желаемого количества и в желаемом порядке):

```
class AudioVisualizerModel {
      ...
      updateLevels = (pcmData: Uint8Array): void => {
          const normalizedLevels =
               this.positionModel.processLevels(pcmData,
this.levels.value.length);

          this.levels.changeValue([...normalizedLevels]);
      };

      // перезапись массива делений с новым количеством
      updateCount = (newCount: number): void => {
          this.levels.changeValue(Array<number>(newCount).fill(0));
      };
      ...
}
```

Именно деления будут выводиться на графике. Тогда необходимо, чтобы визуализатор мог получать частоты для обработки. Очевидно, нужно добавить новое поле **_audioVisualizer** в **AudioRecorderModel**.

Сделаем в анализаторе функцию, которая бы вызывалась при обновлении pcmData. Для этого понадобится дополнительный тип и обновление принимаемых значений в конструкторе:

```
// тип перехватчика
type DataHandler = (data: Uint8Array) => void;

type Props = {
  stream: MediaStream;
  handler: DataHandler;
};
```

Добавим новое поле в класс:

```
class AudioAnalyzerModel {
  ...
      private _dataHandler: DataHandler | null = null;
      ...
}
```

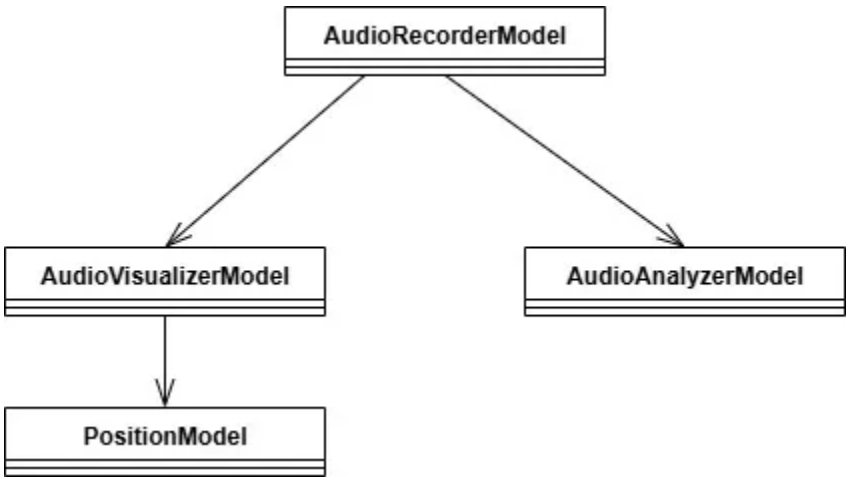
Тогда в конструкторе понадобится сохранить передаваемую из параметров функцию:

```
class AudioAnalyzerModel {
  ...
      constructor({ stream, handler }: Props) {
          this._dataHandler = handler;
          ...
      }
      ...
}
```

Создание анализатора станет таким:

```
class AudioRecorderModel {
      startRecording = async (): Promise<void> => {
            ...
            this._audioAnalyzer = new AudioAnalyzerModel({
                  stream,
              handler: this._audioVisualizer.updateLevels,
            });
            ...
      }
}
```

В конечном итоге получится следующая иерархия моделей:



Теперь необходимо создать компонент звукового графика со всеми заданными стилями и настройками.

Отображение спектрограммы

Базовые компоненты

Чтобы нарисовать аудиографик, необходимо показать все деления, которые хранятся в визуализаторе. Создадим компонент **TrackUnit** для каждого деления:

```
import { observer } from 'mobx-react';
import * as React from 'react';

import { useSpectrogramPageStore } from
'pages/SpectrogramPage/model';

import s from './TrackUnit.module.scss';

// минимальная высота каждого деления
const MIN_HEIGHT = '14px';

type Props = {
  height: number;
};

const TrackUnit: React.FC<Props> = ({ height }) => {

  // получение делений из созданного заранее контекста, где лежат
модели со значениями
  const {
    audioRecorderModel: {
      visualizer: { levels },
    },
  } = useSpectrogramPageStore();

  return (
    <div
      className={s.root}
      style={{
        width: `calc(100% / ${levels.value.length})`,
      }}
    >
```

```
      <div
        className={s.root__content}
        style={{
             height: `calc((100% - ${MIN_HEIGHT}) * ${height} / 100
+ ${MIN_HEIGHT})`
           }}
      />
    </div>
  );
};

export default observer(TrackUnit);
```

Этот компонент принимает в пропсах высоту деления height. В нем контролируется как ширина (зависящая от количества всех делений) каждой полосы, так и ее длина:

- **width = 100% / levels.value.length**
- **height = max(length, 14px)**

Теперь создадим компонент графика — **AudioTrack**. В нем выводятся все деления:

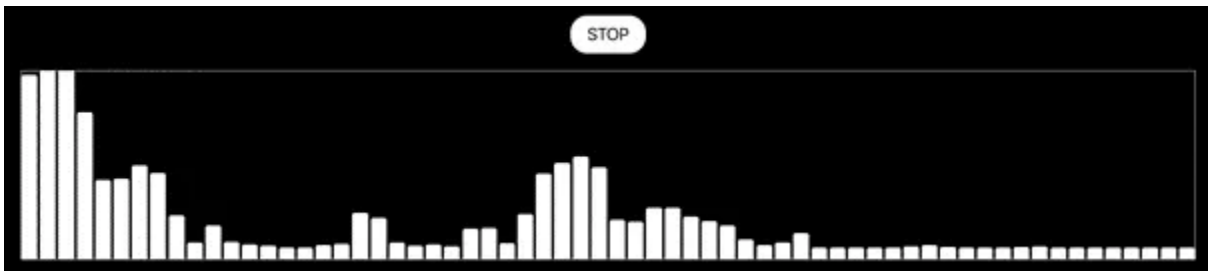
```
import { observer } from 'mobx-react';
import * as React from 'react';

import { useSpectrogramPageStore } from
'pages/SpectrogramPage/model';

import s from './AudioTrack.module.scss';
import { TrackUnit } from './TrackUnit';

const AudioTrack: React.FC = () => {

  // получение данных из контекста
  const {
    audioRecorderModel: {
      visualizer: { levels },
    },
  } = useSpectrogramPageStore();

  return (
    <div className={s.root}>
        {levels.value.map((level, index) => (
          <TrackUnit key={index} length={level} />
        ))}
```

```
      </div>
  );
};

export default observer(AudioTrack);
```

Затем добавим компонент кнопки для начала и завершения записи:

```
import { observer } from 'mobx-react';
import * as React from 'react';

import { useSpectrogramPageStore } from
'pages/SpectrogramPage/model';

import s from './RecorderButton.module.scss';

const RecorderButton: React.FC = () => {

  // получение данных из модели
  const {
    audioRecorderModel: { startRecording, stopRecording, isRecording
},
  } = useSpectrogramPageStore();

  return (
    <button className={s.root} onClick={isRecording ? stopRecording :
startRecording}>
      {isRecording ? <>STOP</> : <>START</>}
    </button>
  );
};

export default observer(RecorderButton);
```

Теперь можно посмотреть и сам график:



Изменение аудио параметров и стилей

Чтобы контролировать параметры моделей, я заранее создала компоненты слайдера и чекбоксов. Изменяемые параметры анализатора можно контролировать при помощи слайдера, причем диапазон децибел также можно менять на одной полосе. Для того, чтобы это реализовать, была использована библиотека react-slider.

Теперь необходимо добавить в модели методы, с помощью которых пользователь мог бы менять параметры, а также применить их к самой спектрограмме.

Аудио параметры

Чтобы изменять аудиопараметры, достаточно обновить поля объекта _analyserNode в AudioAnalyzerModel. Добавим следующие методы, с помощью которых можно менять smoothingTimeConstant, minDecibels, maxDecibels и FFT Size:

```
class AudioAnalyzerModel {
  ...
      // обновление константы сглаживания
      setTimeSmoothing = (value: number) => {
        this._analyserNode.value.smoothingTimeConstant = value;
      };

      // обновление диапазона децибел
      setDecibels = (min: number, max: number) => {
            // значения не должны быть равны
            if (min === max) {
       return;
     }

        this._analyserNode.value.minDecibels = min;
        this._analyserNode.value.maxDecibels = max;
      };

      // обновление размера преобразования Фурье в зависимости от
количества делений
      setFFTSize = (levelCount: number) => {
        // по умолчанию значение FFT_SIZE = 128
        let newSize = FFT_SIZE;

        // все доступные значения количества полос, кроме 3, равны
степеням двойки, поэтому можно поменять FFT Size пропорционально
        if (levelCount >
AVAILABLE_AUDIO_LEVELS[AudioLevelKeys.three]) {
          newSize = levelCount * 2;
```

```
        }

        this._analyserNode.value.fftSize = newSize;
        this._pcmData.changeValue(new
Uint8Array(this._analyserNode.value.frequencyBinCount));
      };
           ...
}
```

Сложность возникает только при контроле FFT Size, так как необходимо обновить длину массива pcmData, используя поле frequencyBinCount.

Так как модель **AudioAnalyzerModel** может анализировать звук только при наличии stream и создается только при записи, необходимо сделать отдельно хранимые параметры в модели **AudioRecorderModel**. Для этого нужно создать конфиги, которые определяют тип каждой аудионастройки и ее значения по умолчанию:

```
export type AudioAnalyzerProps = {
  [AudioAnalyzerEnum.timeSmoothing]: number;
  [AudioAnalyzerEnum.decibels]: number[];
};

export const AUDIO_ANALYZER_DEFAULT: AudioAnalyzerProps = {
  [AudioAnalyzerEnum.timeSmoothing]: 0.2,
  [AudioAnalyzerEnum.decibels]: [-90, -10],
};
```

В модели **AudioRecorderModel** добавим новое поле audioParams, которое хранит значения, изменяемые пользователем, а также методы setTimeSmoothing и setDecibels, которые обновляют пользовательские значения и записывают их в анализатор:

```
class AudioRecorderModel {
      ...
      readonly audioParams = new
FieldModel<AudioAnalyzerProps>(AUDIO_ANALYZER_DEFAULT);
      ...

      // обновление константы сглаживания
      setTimeSmoothing = (value: number): void => {
        this.audioParams.changeValue({
          ...this.audioParams.value,
          timeSmoothing: value,
        });
        this._audioAnalyzer?.setTimeSmoothing(value);
      };
```

```
      // обновление диапазона децибел
      setDecibels = (value: number[]): void => {
        this.audioParams.changeValue({
          ...this.audioParams.value,
          decibels: value,
        });
        this._audioAnalyzer?.setDecibels(Math.min(...value),
Math.max(...value));
      };
      ...
}
```

Также добавим в конструктор модели **AudioAnalyzerModel** начальные параметры и число делений, так как пользователь может их настроить перед записью:

```
type Props = {
  stream: MediaStream;
  handler: DataHandler;
  params: AudioAnalyzerProps;
  levelCount: number;
};

class AudioAnalyzerModel {
  ...
      // конструктор AudioAnalyzerModel
      constructor({ stream, handler, params, levelCount }: Props) {
        ...
        analyserNode.smoothingTimeConstant = params.timeSmoothing;
        analyserNode.minDecibels = params.decibels[0];
        analyserNode.maxDecibels = params.decibels[1];
        ...
      }
      ...
}
```

Определим возможные диапазоны значений для каждого параметра:

```
// тип настройки для каждого параметра
export type AudioAnalyzerPropSettings = {
  key: AudioAnalyzerEnum;
  min: number;
  max: number;
  title: React.ReactNode;
};
```

```
// диапазоны значений и конфиги
export const DEFAULT_ANALYZER_SETTINGS: Record<AudioAnalyzerEnum,
AudioAnalyzerPropSettings> = {
  [AudioAnalyzerEnum.timeSmoothing]: {
    min: 0,
    max: 100,
    title: 'Константа сглаживания',
    key: AudioAnalyzerEnum.timeSmoothing,
  },
  [AudioAnalyzerEnum.decibels]: {
    min: -100,
    max: 0,
    title: 'Децибелы',
    key: AudioAnalyzerEnum.decibels,
  },
};
```

Теперь привяжем эти методы и переменные к слайдеру, чтобы менять диапазон децибел:

```
<Slider
  value={audioParams.value.decibels}
  onChange={setDecibels}
  {...DEFAULT_ANALYZER_SETTINGS.decibels}
/>
```

Получится вот такой компонент:

Компонент Slider использует ReactSlider, который позволяет в качестве значения передавать как одно число, так и массив чисел.

Стили визуализатора

Теперь применим настройки визуализатора. Перед тем, как использовать их в React-компонентах, необходимо сделать небольшое изменение в AudioRecorderModel.

Как было написано выше, для количества делений нужно контролировать как поля визуализатора, так и аудиопараметры (в частности FFT Size), поэтому нужно дополнительно добавить метод в модель AudioRecorderModel, который бы обращался и к анализатору, и к визуализатору:

```
class AudioRecorderModel {
  ...
      // обновление числа делений
      setLevelCount = (count: number): void => {
        this._audioAnalyzer?.setFFTSize(count);
        this._audioVisualizer.updateCount(count);
      };
      ...
}
```

Теперь применим к спектрограмме все настройки, которые были заданы в визуализаторе:

1. Прозрачность

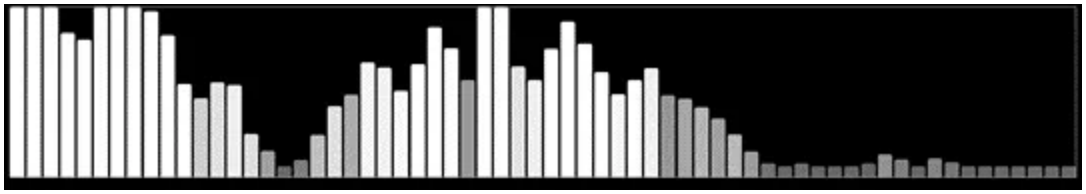
Чтобы сделать деления прозрачными в зависимости от громкости, в компоненте TrackUnit необходимо контролировать свойство opacity через пропс style:

```
// минимально возможная прозрачность
const OPACITY_LOW_LIMIT = 0.4;

const opacityValue = React.useMemo(() => {
  // если включена прозрачность, то ставим прозрачность как процент
длины, учитывая минимальный порог 40% прозрачности
  if (opacityMode.value) {
    return Math.min(length / 100 + OPACITY_LOW_LIMIT, 1);
  }

  return 1;
}, [opacityMode.value, length]);
```

В компоненте это поле будет применяться таким образом:

```
<div
  className={s.root}
  style={{
    width: `calc(100% / ${levels.value.length})`,
    opacity: opacityValue,
  }}
>
```

Теперь можно увидеть, что у каждого деления свой уровень прозрачности:



2. Ось

Сделаем такие конфиги:

```
// возможные варианты выбора для оси
export const AUDIO_LINE_STYLE: Record<AudioLineStyle, string> = {
  [AudioLineStyle.center]: 'center',
  [AudioLineStyle.top]: 'start',
  [AudioLineStyle.bottom]: 'end',
};

// функция, определяющая borderRadius для каждой полосы в зависимости
от выбранной оси
export const audioLineBorderRadius = (style: AudioLineStyle,
borderValue: string) => {
  switch (style) {
    case AudioLineStyle.center:
      return borderValue;
    case AudioLineStyle.top:
      return `0 0 ${borderValue} ${borderValue}`;
    case AudioLineStyle.bottom:
    default:
      return `${borderValue} ${borderValue} 0 0`;
  }
};
```

Эти значения необходимо применить в компоненте **TrackUnit**:

```
<div
  className={s.root}
  style={{
    justifyContent: AUDIO_LINE_STYLE[line.value],
    width: `calc(100% / ${levels.value.length})`,
    opacity: opacityValue,
  }}
>
  <div
    className={s.root__length}
    style={{
```

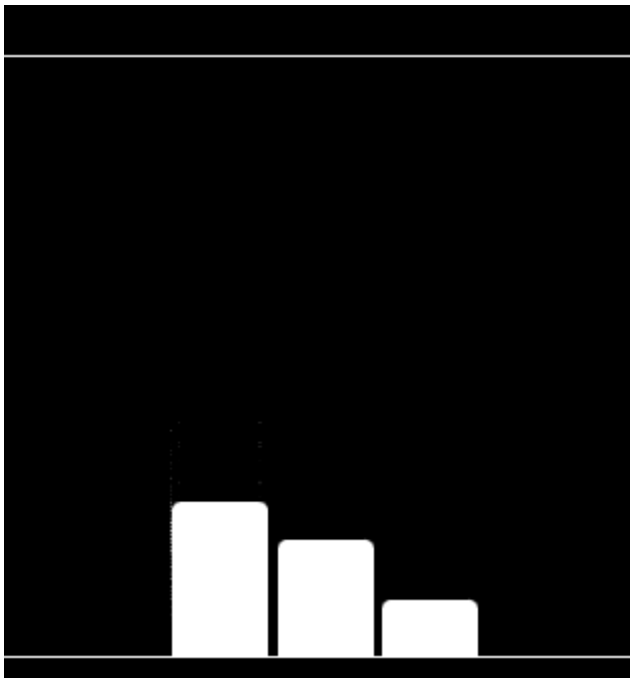
```
      height: `max(calc(${length}%), 14px)`,
      borderRadius: audioLineBorderRadius(line.value, '3px'),
    }}
  />
</div>
```

Пример использования значения оси по центру:

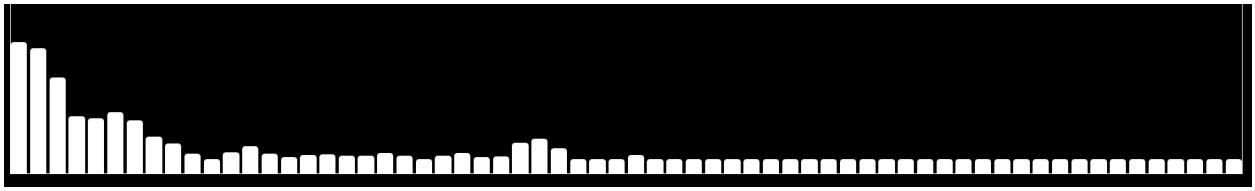
Остальные параметры уже применяются к спектрограмме, так как для них было необходимо менять массив делений. Приведу пример компонента, который позволяет настраивать число делений:

В итоге варианты разных делений будут выглядеть так:

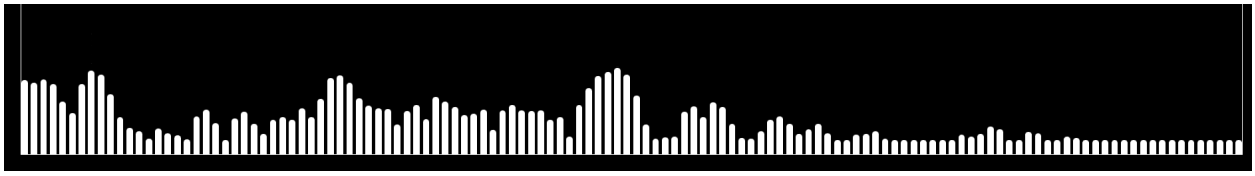
для 3 делений:



для 64 делений:

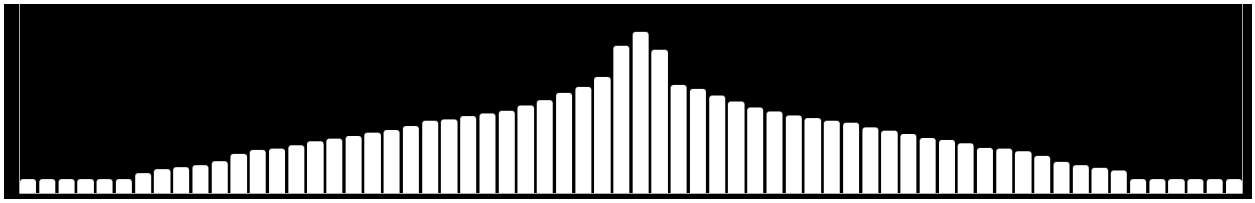


для 128 делений:

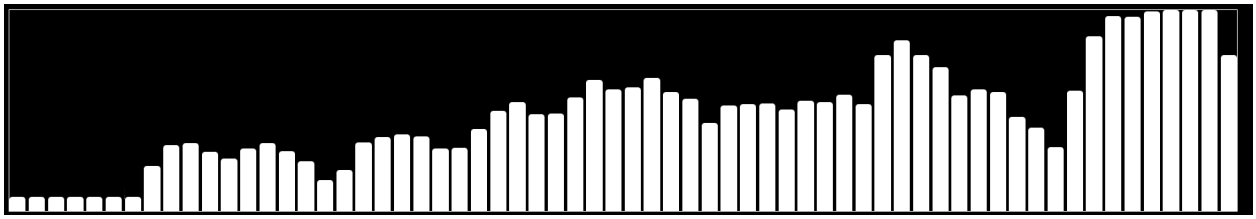


Также при изменении положения делений можно расположить их:

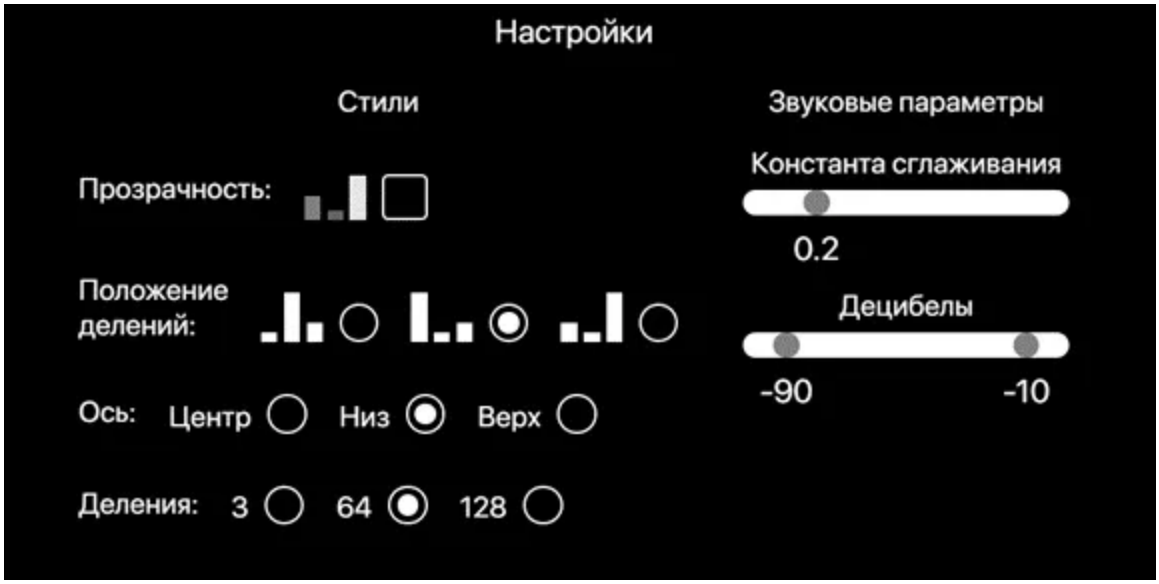
в центре:



с правой стороны:



Все параметры будут выглядеть так:



Код

Здесь можно посмотреть код всех моделей.

Модель AudioAnalyzerModel

```
import { FieldModel } from '@ktsstudio/mediaproject-stores';

import {
  AudioAnalyzerProps,
  AudioLevelKeys,
  AVAILABLE_AUDIO_LEVELS,
} from 'entities/audio';
import { getCurrentLoudness } from 'utils/getCurrentLoudness';

const FFT_SIZE = 128;
const SAMPLE_RATE = 44100;
const SET_LOUDNESS_TIMEOUT_MS = 1000 / 10;

type DataHandler = (data: Uint8Array) => void;

type Props = {
  stream: MediaStream;
  handler: DataHandler;
  params: AudioAnalyzerProps;
  levelCount: number;
};
```

```
type CreatingProps = Omit<Props, "handler">;

export default class AudioAnalyzerModel {
  private readonly _pcmData = new FieldModel<Uint8Array |
null>(null);
  private readonly _analyserNode: FieldModel<AnalyserNode>;
  private _dataHandler: DataHandler | null = null;

  readonly maxLoudness = new FieldModel<number | null>(null);
  readonly currentLoudness = new FieldModel<number>(0);

  private _setLoudnessIntervalId: NodeJS.Timeout | null = null;

  constructor({ stream, handler, params, levelCount }: Props) {
    this._dataHandler = handler;

    const { node, sourceNode } = this._createAnalyzer({stream,
params, levelCount});

    this._analyserNode = new FieldModel(node);
    sourceNode.connect(this._analyserNode.value);

    this._pcmData.changeValue(new
Uint8Array(this._analyserNode.value.frequencyBinCount));
  }

  private _getUpdatedFFTSize = (levelCount: number): number => {
    let fftSize = FFT_SIZE;

    if (levelCount > AVAILABLE_AUDIO_LEVELS[AudioLevelKeys.three]) {
      fftSize = levelCount * 2;
    }

    return fftSize;
  }

  private _createAnalyzer = ({ stream, params, levelCount }:
CreatingProps): {
    node: AnalyserNode;
    sourceNode: MediaStreamAudioSourceNode;
  } => {
    const audioContext = new AudioContext({ sampleRate: SAMPLE_RATE
});
```

```
    const mediaStreamAudioSourceNode =
audioContext.createMediaStreamSource(stream);
    const analyserNode = audioContext.createAnalyser();

    analyserNode.fftSize = this._getUpdatedFFTSize(levelCount);
    analyserNode.smoothingTimeConstant = params.timeSmoothing;
    analyserNode.minDecibels = params.decibels[0];
    analyserNode.maxDecibels = params.decibels[1];

    return {
      node: analyserNode,
      sourceNode: mediaStreamAudioSourceNode,
    };
  };

  private _analysisFunc = () => {
    // проверка
    if (!this._analyserNode.value || !this._pcmData.value) {
      return;
    }

    // заполнение данными pcmData

this._analyserNode.value.getByteFrequencyData(this._pcmData.value);

    if (this._dataHandler) {
      this._dataHandler(this._pcmData.value);
    }

    // получение текущей громкости

this.currentLoudness.changeValue(getCurrentLoudness(this._pcmData.val
ue));

    // расчет максимальной громкости
    if (!this.maxLoudness.value || this.maxLoudness.value <
this.currentLoudness.value) {
      this.maxLoudness.changeValue(this.currentLoudness.value);
    }
  };

  analyze() {
    this._setLoudnessIntervalId = setInterval(this._analysisFunc,
SET_LOUDNESS_TIMEOUT_MS);
  }
```

```
  // обновление константы сглаживания
  setTimeSmoothing = (value: number) => {
    this._analyserNode.value.smoothingTimeConstant = value;
  };

  // обновление диапазона децибел
  setDecibels = (min: number, max: number) => {
       // значения не должны быть равны
           if (min === max) {
      return;
    }
    this._analyserNode.value.minDecibels = min;
    this._analyserNode.value.maxDecibels = max;
  };

  // обновление размера преобразования Фурье
  setFFTSize = (levelCount: number) => {
    this._analyserNode.value.fftSize =
this._getUpdatedFFTSize(levelCount);
    this._pcmData.changeValue(new
Uint8Array(this._analyserNode.value.frequencyBinCount));
  };

  stopAnalysis() {
    if (this._setLoudnessIntervalId) {
      clearInterval(this._setLoudnessIntervalId);
    }

    if (this._dataHandler) {
      this._dataHandler = null;
    }
  }
}
```

Модель AudioRecorderModel

```
import { FieldModel } from '@ktsstudio/mediaproject-stores';
import { action, computed, makeObservable, observable } from 'mobx';

import { AUDIO_ANALYZER_DEFAULT, AudioAnalyzerProps } from
'entities/audio';

import { AudioAnalyzerModel } from '../AudioAnalyzerModel';
import { AudioVisualizerModel } from '../AudioVisualizerModel';
```

```
type PrivateFields =
  | '_audioAnalyzer'
  | '_mediaStream'
  | '_startRecording'
  | '_stopRecording';

export default class AudioRecorderModel {
  private _audioAnalyzer: AudioAnalyzerModel | null = null;
  private _audioVisualizer: AudioVisualizerModel = new
AudioVisualizerModel({});
  private _mediaStream: MediaStream | null = null;

  private _recordingStartTime = new FieldModel<number | null>(null);
  private _recordingCounter = new FieldModel<number>(0);
  private _recordingIntervalId: NodeJS.Timeout | null = null;

  readonly audioParams = new
FieldModel<AudioAnalyzerProps>(AUDIO_ANALYZER_DEFAULT);

  readonly isRecordingStatus = new FieldModel<boolean>(false);

  constructor() {
    makeObservable<AudioRecorderModel, PrivateFields>(this, {
      _audioAnalyzer: observable.ref,
      _mediaStream: observable.ref,

      isRecording: computed,
      timeRecording: computed,

      visualizer: computed,

      _startRecording: action.bound,
      _stopRecording: action.bound,
      startRecording: action,
      stopRecording: action,
      setLevelCount: action,
    });
  }

  get isRecording(): boolean {
    return this.isRecordingStatus.value;
  }

  get timeRecording(): string {
```

```
  const totalSeconds = this._recordingCounter.value;

  const hours = Math.floor(totalSeconds / 3600);
  const minutes = Math.floor((totalSeconds % 3600) / 60);
  const seconds = totalSeconds % 60;

  const pad = (n: number) => n.toString().padStart(2, '0');

  return `${pad(hours)}:${pad(minutes)}:${pad(seconds)}`;
}

get currentLoudness(): number {
  return this._audioAnalyzer?.currentLoudness.value ?? 0;
}

get maxLoudness(): number {
  return this._audioAnalyzer?.maxLoudness.value ?? 0;
}

get visualizer(): AudioVisualizerModel {
  return this._audioVisualizer;
}

startRecording = (): void => {
  if (this.isRecording) {
    return;
  }

  void this._startRecording();
};

stopRecording = (): void => {
  if (this.isRecording) {
    this._stopRecording();
  }
};

setTimeSmoothing = (value: number): void => {
  this.audioParams.changeValue({
    ...this.audioParams.value,
    timeSmoothing: value,
  });
  this._audioAnalyzer?.setTimeSmoothing(value);
};
```

```
  setDecibels = (value: number[]): void => {
    this.audioParams.changeValue({
      ...this.audioParams.value,
      decibels: value,
    });
    this._audioAnalyzer?.setDecibels(Math.min(...value),
Math.max(...value));
  };

  setLevelCount = (count: number): void => {
    this._audioAnalyzer?.setFFTSize(count);
    this._audioVisualizer.updateCount(count);
  };

  private _startTimer() {
    this._recordingCounter.changeValue(0);
    this._recordingStartTime.changeValue(Date.now());
    this._recordingIntervalId = setInterval(() => {
      this._increaseCounter();
    }, 1000);
  }

  private _increaseCounter() {
    const start = this._recordingStartTime.value;

    if (start !== null) {
      this._recordingCounter.changeValue(Math.floor((Date.now() -
start) / 1000));
    }
  }

  private _stopTimer() {
    if (this._recordingIntervalId) {
      clearInterval(this._recordingIntervalId);
      this._recordingIntervalId = null;
    }
  }

  private _startRecording = async (): Promise<void> => {
    try {
      const stream = await navigator.mediaDevices.getUserMedia({
        audio: true,
      });

      this._audioAnalyzer = new AudioAnalyzerModel({
```

```
        stream,
        params: this.audioParams.value,
        handler: this._audioVisualizer.updateLevels,
        levelCount: this._audioVisualizer.levels.value.length,
      });

      this._mediaStream = stream;

      if (!this._mediaStream) {
        return;
      }

      this.isRecordingStatus.changeValue(true);
      this._audioAnalyzer.analyze();
      this._startTimer();
    } catch (e) {
      console.error('Error accessing microphone:', e);
    }
  };

  private _stopRecording() {
    this._audioAnalyzer?.stopAnalysis();
    this._audioVisualizer.reset();
    this._stopTimer();

    this.isRecordingStatus.changeValue(false);

    this._mediaStream?.getTracks().forEach((track) => track.stop());
    this._reset();
  }

  private _reset = () => {
    this._audioAnalyzer = null;
    this._audioVisualizer.reset();
  };

  destroy() {
    this.stopRecording();
  }
}
```

Модель PositionModel

```
import { FieldModel } from '@ktsstudio/mediaproject-stores';
```

```
import {
  AudioPositionStyle,
  AudioLevelKeys,
  AVAILABLE_AUDIO_LEVELS,
  getAudioPercent
} from 'entities/audio';

class BasePositionModel {
  processLevels(pcmData: Uint8Array, levelCount: number): number[] {
    const arrData: number[] = Array.from(pcmData);
    const normalizedValues = arrData.map((value) =>
getAudioPercent(value));

    if (levelCount > AVAILABLE_AUDIO_LEVELS[AudioLevelKeys.three]) {
      return normalizedValues;
    }

    // длина одного диапазона
    const bins = Math.floor(normalizedValues.length / levelCount);

    // индексы диапазонов
    const rangesIndices: number[][] = Array.from({ length: levelCount
}, (_, i) =>
      // eslint-disable-next-line @typescript-eslint/no-shadow
      Array.from({ length: bins }, (_, j) => i * bins + j)
    );

    return rangesIndices.map((indices) => {
      const values: number[] = indices.map((index) =>
normalizedValues[index]);
      const sum = values.reduce((acc, val) => acc + val, 0);

      return values.length > 0 ? sum / values.length : 0;
    });
  }
}

class CenterPositionModel extends BasePositionModel {
  processLevels(pcmData: Uint8Array, levelCount: number): number[] {
    const baseLevels = super.processLevels(pcmData,
levelCount).sort((a, b) => a - b);
    const leftLevels: number[] = [];
    const rightLevels: number[] = [];

    baseLevels.forEach((level, index) => {
```

```
      if (index % 2 === 0) {
        leftLevels.push(level);
      } else {
        rightLevels.push(level);
      }
    });

    return [...leftLevels, ...rightLevels.reverse()];
  }
}

class RightPositionModel extends BasePositionModel {
  processLevels(pcmData: Uint8Array, levelCount: number): number[] {
    return super.processLevels(pcmData, levelCount).reverse();
  }
}

export default class PositionModel extends BasePositionModel {
  readonly position: FieldModel<AudioPositionStyle>;
  readonly centerMode = new CenterPositionModel();
  readonly rightMode = new RightPositionModel();

  constructor(position: AudioPositionStyle) {
    super();
    this.position = new FieldModel(position);
  }

  processLevels(pcmData: Uint8Array, levelCount: number): number[] {
    const baseLevels = super.processLevels(pcmData, levelCount);

    switch (this.position.value) {
      case AudioPositionStyle.center:
        return this.centerMode.processLevels(pcmData, levelCount);
      case AudioPositionStyle.right:
        return this.rightMode.processLevels(pcmData, levelCount);
      case AudioPositionStyle.left:
      default:
        return baseLevels;
    }
  }
}
```

Модель AudioVisualizerModel

```
import { FieldModel } from '@ktsstudio/mediaproject-stores';
```

```
import {
  AudioLevelKeys,
  AudioLineStyle,
  AudioPositionStyle,
  AVAILABLE_AUDIO_LEVELS,
} from 'entities/audio';

import { PositionModel } from './PositionModel';

export type AudioVisualizerProps = {
  levelCount?: number;
  position?: AudioPositionStyle;
  line?: AudioLineStyle;
  opacityMode?: boolean;
};

export default class AudioVisualizerModel {
  readonly levels: FieldModel<number[]>;
  readonly positionModel: PositionModel;
  readonly line: FieldModel<AudioLineStyle>;
  readonly opacityMode: FieldModel<boolean>;

  constructor({
    levelCount = AVAILABLE_AUDIO_LEVELS[AudioLevelKeys.short],
    position = AudioPositionStyle.left,
    line = AudioLineStyle.bottom,
    opacityMode = false,
  }: AudioVisualizerProps) {
    this.levels = new FieldModel(Array<number>(levelCount).fill(0));
    this.positionModel = new PositionModel(position);
    this.line = new FieldModel(line);
    this.opacityMode = new FieldModel(opacityMode);
  }

  get position() {
    return this.positionModel.position;
  }

  updateLevels = (pcmData: Uint8Array): void => {
    const normalizedLevels =
          this.positionModel.processLevels(pcmData,
this.levels.value.length);
    this.levels.changeValue([...normalizedLevels]);
  };
```

```
  // обновление числа делений
  updateCount = (newCount: number): void => {
    this.levels.changeValue(Array<number>(newCount).fill(0));
  };

  reset(): void {
    this.levels.changeValue(Array(this.levels.value.length).fill(0));
  }
}
```

Заключение

Итого, мы научились создавать звуковую спектрограмму на React с помощью MobX и Web Audio API, которую пользователь может настраивать по своему усмотрению. Надеюсь, этот опыт будет полезен вам на ваших проектах. Если где-то не хватило деталей — пишите, буду рада рассказать подробнее.

А о том, как мы делаем другие спецпроекты, можно почитать в статьях моих коллег:

- Как интегрировать миниапп (активность) в Discord
- Three.js с нуля на практике: как за несколько часов создать аркадную 3D-игру. Часть 1
- Как сделать анимацию разными способами: CSS, WebP, Canvas, Lottie, Spine и секвенции
- Летающий Санта и танцующие снегири: опыт реализации и оптимизации CSS-анимации