r-audio: Declarative, reactive and flexible Web Audio graphs in React

Jakub Fiala
BBC Research & Development
Centre House, 56 Wood Lane
London W12 7SB
jakub.fiala@bbc.co.uk

ABSTRACT

Web Audio is by design an object-oriented, imperative API offering low-level control over audio graphs. There have been a number of efforts to provide a more intuitive wrapper API. Designing such wrapper libraries poses challenges in addressing graph configuration, dynamic mutation and data flow. Syntax of creating directed graphs in imperative code is not representative of the complex graph shapes, making the code difficult to understand without external visualisation tools.

In this paper I describe *r-audio*[1], a Web Audio wrapper library which attempts to solve the issues of imperative graph representations by leveraging the component system of React. I compare approaches of existing wrapper libraries and discuss solutions to specific issues of declarative and reactive representations of Web Audio graphs. I evaluate *r-audio* in terms of the ability to create arbitrary directed graphs and mutate them in real time.

CCS Concepts

•**Software and its engineering → Data flow architectures; Abstraction, modeling and modularity;** Object oriented architectures;

Keywords

JavaScript, Web Audio, React, directed graphs, declarative programming

1. INTRODUCTION

According to the Web Audio API specification, the primary paradigm of the API is "the audio routing graph, where a number of `AudioNode` objects are connected together to define the overall audio rendering"[2]. The audio signal is routed from one or more source nodes to one or more destination nodes. The graph is directed and can be cyclic. As potential use cases the specification suggests "reasonably complex games and interactive applications, including musical ones". Such use cases may require considerably large graphs. The graph complexity is further increased with support for multi-channel processing and spatialised audio. Most Web Audio nodes contain `AudioParam`s which represent the state of the graph. The state needs to be managed individually for every node, which can lead to a need for higher-level abstractions.

1.1 Creating directed graphs in Web Audio

The Web Audio API is designed as an object-oriented, imperative API. `AudioNode`s are stateful class instances and connections are created by calling the `connect` method of the source node, passing the destination node as the first argument. The code for creating a linear processing pipeline is reasonably representative of the graph shape, but more complex routings are not so apparent.

```
source = audioCtx.createMediaStreamSource(
   stream);
source.connect(analyser);
analyser.connect(distortion);
distortion.connect(delay);
delay.connect(gainNode);
gainNode.connect(distortion);
delay.connect(audioCtx.destination);
```

Listing 1: This Web Audio graph may seem linear at a first glance, but it contains a loop.

In Listing 1, each `AudioNode` has its own variable binding and retains its state in its `AudioParam`s. Every connection must be explicitly declared. All state transitions and graph mutations must be performed manually which may lead to verbose and counter-intuitive coding patterns. Errors are more likely to occur in large graphs where minor mutations may have large consequences.

2. RELATED WORK

Many client-side JavaScript frameworks use a declarative-style API as an abstraction for `document.createElement` to simplify creation of deep DOM trees. A-Frame[1] creates WebVR scenes from custom elements embedded directly in the HTML document. React[11] uses JSX syntax similar to HTML and XML where elements are declared and configured using *props*[12]. Declarative APIs make it simpler to create *data bindings* between explicit state in JavaScript and implicit state in the DOM or a WebGL context. React in particular maintains a virtual DOM data structure and performs the smallest amount of work required for each state transition[3].

[1]`https://github.com/bbc/r-audio`

Web Audio Conference WAC-2018, September 19–21, 2018, Berlin, Germany.

Volke et al.[14] describe a declarative Web Audio library based on native HTML Custom Elements embedded directly into the document. Individual custom elements represent both data (`<webaudio-buffer/>`) and nodes (`<webaudio-source/>`). Connections between nodes are created implicitly. Processing branches are created as children of source nodes, but cycles and branch merging is not accounted for. The library authors do not provide examples of graph mutation.

The *virtual-audio-graph* library[8] is inspired by virtual DOM frameworks. It simplifies state management by decoupling explicit and implicit states and provides an `update` method to change the implicit state of the audio graph. The graph is represented by a plain object with keys acting as node IDs. Connections are made explicitly by specifying the destination node ID. Nested structures are created using a provided `createNode` method. While *virtual-audio-graph* can support complex graphs, explicit connections and key-based node references are little different from the imperative native API. The shape of the audio graph cannot easily be inferred from looking at the code which generates it.

react-webaudio[9] is (like our library) based on React, allowing for graph mutation and state management. However, it only supports a limited subset of Web Audio nodes and creates implicit connections from parent nodes to children only, resulting in deeply nested code.

3. LIBRARY DESIGN

3.1 Dependency on *React*

r-audio is designed as a collection of React components. React is one of the most popular UI frameworks and its component system and virtual DOM implementations have a number of useful properties:

- Components are inherently modular and reusable
- JSX syntax is suitable for compact representations of hierarchical structures with properties
- Virtual DOM offers separation and management of explicit and implicit state
- React's reconciliation mechanism ensures that the least amount of DOM updates are performed to achieve the effect specified by explicit application state. [2]
- Unidirectional data bindings make it easy to reason about data flow
- Component connections can be represented implicitly by parent-child relationships.

React encourages a "single source of state" model where application state is stored in a single JavaScript object. The object is treated as immutable, so every update is a map between the current state object and a new, different state object. Such state transitions are easier to reason about in large applications and can be tracked precisely, which simplifies debugging[7].

[2]Suppose a list of JavaScript objects is transformed into HTML `<li>` elements. If some of the objects are changed in application code, React only updates the DOM elements affected by the change.

3.2 Context and inheritance

r-audio includes three base classes which inherit from `React.Component` — `RAudioContext` represents an `AudioContext` instance and a node registry; `RComponent` represents a component interacting with `RAudioContext`; `RAudioNode` extends `RComponent` and adds parameter and connection management methods for `AudioNode`s.

All `RComponent`s are given a reference to the parent context using React's context API. This reference is used to instantiate nodes on the audio graph. Each node type is represented by a component which extends `RAudioNode`. This component is responsible solely for instantiating an `AudioNode` on the parent context and specifying its parameters.

3.3 Constructing graphs

React components don't expose internal variables to their siblings. In order to create implicit connections between adjacent JSX nodes, *r-audio* stores references to all nodes in a `Map` bound to the `RAudioContext`. When a node is created, its reference is added to the map under a unique Symbol key. The key can refer to a single node or a list of Symbols pointing to one node each.

The JSX syntax can denote *parent-child* and *sibling-sibling* relationships between nodes. I designed the following graph construction components, which can be composed to represent any arbitrary graph using the two relationships:

- `RPipeline` connects its child nodes in a series, directing its input to the first child and connecting the last child to its output
- `RSplit` connects its input to its child nodes in parallel, and connects all children to its output at the end
- `RSplitChannels` is similar to `RSplit`, but connects a different channel of the input signal to every child
- `RCycle` connects each child's output to the child's input and to its destination

I found the only graph configuration not possible purely by composing graph construction components is that of a branch without a destination. To allow such a configuration any node in *r-audio* can receive the `disconnected` prop, preventing it from connecting to any other nodes.

Fig. 1 shows how graph construction components modify their children by providing a *destination function*, which returns a list of nodes the child is required to connect to. The child connects to an `AudioParam` of its destination nodes if the `connectToParam` prop is added to it.

3.3.1 *Connectable and non-connectable component types*

A number of Web Audio nodes act as signal sources and cannot be connected *to*. This becomes an issue with serial connections, i.e. within a `RPipeline`. If such a node appears between two other nodes, an inbound branch must be created. Consider the graph configuration in Listing 2.

```
<RPipeline>
    <ROscillator start={0} frequency={440}
        />
    <RBiquadFilter frequency={5000} />
    <ROscillator
        start={0}
```

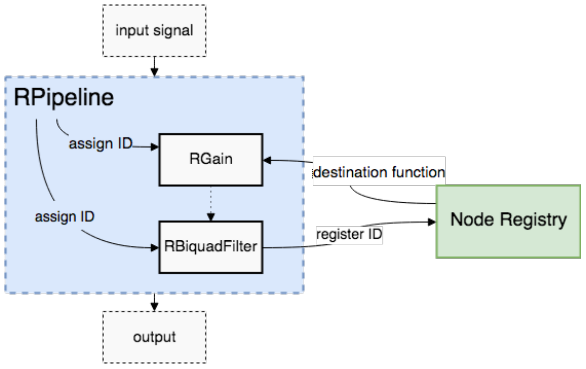


Figure 1: `RPipeline` connects two nodes by assigning Symbol IDs to them and providing the `RGain` with a destination function which returns the `RBiquadFilter` node based on its ID

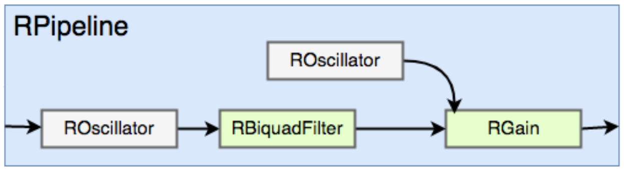


Figure 2: Inbound branch created when a non-connectable node is placed between connectable nodes in a pipeline.

```
        frequency={2}
        connectToParam="gain" />
    <RGain gain={0.5} />
</RPipeline>
```

Listing 2: A pipeline with a non-connectable node in the middle.

The pipeline is resolved from the end, so `RGain` will be connected to the nearest preceding connectable node — `RBiquadFilter`. The second oscillator will constitute an inbound branch and connect to the gain parameter of `RGain`. The resulting graph is depicted in Fig. 2.

3.3.2 *Dealing with channels*

r-audio implements `RChannelSplitter` and `RChannelMerger` nodes along with props which determine source and destination channel indices for a given node. Since JSX cannot express a many-to-many relationship as needed for channel splitting[3], it is necessary to create an abstraction around it.

The `RSplitChannels` component (Fig. 3) is composed of a pipeline containing a channel splitter, a `RSplit` component and a channel merger. Children are rendered within the `RSplit`. Since `RChannelSplitter` is configured to connect a different channel to each of its destinations, a branch is created for every input channel. The branches are forced to connect to a different channel of the channel merger each.

3.4 Mutating graphs and updating state

Graph mutation support is a core requirement of full Web

[3]i.e. connecting many channels of a node to many destinations

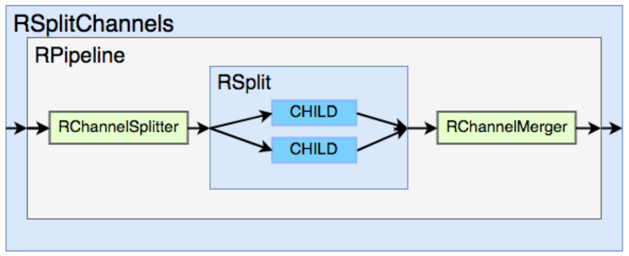
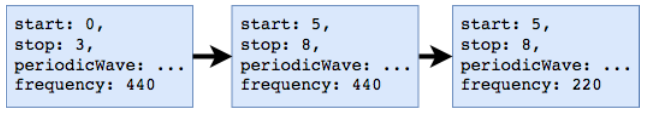


Figure 3: Composition of the RSplitChannels component.



```
start: 0,
stop: 3,
periodicWave: ...
frequency: 440
```

```
start: 5,
stop: 8,
periodicWave: ...
frequency: 440
```

```
start: 5,
stop: 8,
periodicWave: ...
frequency: 220
```

Figure 4: `r-audio` uses React's state management model where there is a single state object at all times. Updates are performed by creating a new object, so it is always clear where mutations originate.

Audio compatibility. In a native Web Audio application, mutations are performed by explicitly disconnecting and reconnecting nodes. This is in contrast to AudioParam updates, which are the other kind of state updates in Web Audio.

To remove a connected node the developer needs to maintain references to multiple surrounding nodes. React's lifecycle methods provide useful hooks to facilitate mutation of the audio graph without the need to explicitly execute reconnections.

The first step in performing a mutation is to disconnect any node being removed from its destinations. This can be easily achieved by calling `disconnect` in the `componentWillUnmount` method of the audio node. In *r-audio* all sibling relationships are created using `RPipeline`, making it the only other component concerned with mutation. The pipeline provides parent references to every child. The child can then disconnect itself from the parent as part of the child's unmount routine. New connections are created implicitly in the `render` method of the pipeline and include nodes surrounding the removed node and any added nodes.

3.4.1 *Scheduling*

In line with React's state management model (Fig. 4), scheduling in *r-audio* is driven by state changes. All nodes which in native Web Audio inherit from `AudioScheduledSourceNode` are subclasses of `RScheduledSource`.

To use these nodes, we can provide a `start` prop with optional `offset` and `duration` props. To stop playback at a certain point, we provide a `stop` prop. The node is responsible for reconciling and scheduling the start and stop times. If the start time occurs after the stop time, playback does not start.

After a scheduled node's playback has ended, Web Audio removes it from the graph. *r-audio* responds to the `onended` event on the node by creating a new instance as soon as playback ends. This ensures the virtual graph is always in sync with the audio graph.

3.4.2 *Updating AudioParams*

Similarly to playback scheduling, changes to parameters in the graph are entirely driven by state updates. By default, when a component's prop changes the AudioParam value is changed immediately. The scheduling can be changed using the `transitionTime` and `transitionCurve` props. The transition time is the scheduled target time for the ramp, and specific transition times and curves can be assigned to props by passing a dictionary to `transitionTime` and `transitionCurve`.

3.5 Data objects and non-audio interaction

3.5.1 *Separation of concerns*

By design, *r-audio* does not provide abstractions for data types specified in the Web Audio API specification. This includes the `AudioBuffer`, `AudioListener` and `PeriodicWave` classes, and media element or stream objects. Separating the audio graph from any data which does not alter it makes it easy to centralise state and correctly model situations where data may not be readily available at the point of instantiation of the graph.

In certain situations data objects are integral to the function of a node. In these cases React lifecycle methods are again a useful paradigm. `RBufferSource`, for instance, expects a buffer object to be provided as a prop. If it is not provided, the node is idle and waits for the `componentDidUpdate` method to be called. Once the method is called and a buffer provided, playback starts automatically. The state of the node is completely driven by its props, so there is no need to call methods explicitly.

3.5.2 *Audio analysis*

In native Web Audio Analyser and BiquadFilter nodes generate analysis data. This data may not necessarily be used in sync with (or at the frequency of) the audio stream. Application developers can call the node's methods and provide a typed array to be filled with frequency- or time-domain data, or a filter's frequency response. This action cannot therefore be entirely state-driven.

r-audio exploits React's ability to render functions as children of components to expose a proxy object to the internal Analyser or BiquadFilter node. The proxy only contains the data extraction methods and its properties are frozen using `Object.freeze`.

Listing 3 shows simple usage of `RAnalyser`. In practice, the proxy object can be stored in the component's state and accessed as needed, e.g. in a call to `requestAnimationFrame`.

```
<RAnalyser fftSize={2048}>
{
  proxy => {
    const data = new Float32Array(
      proxy.frequencyBinCount
    );
    proxy.getFloatFrequencyData(data);
  }
}
</RAnalyser>
```

Listing 3: An example of using the proxy object provided by `RAnalyser`.

3.5.3 *AudioWorklet*

The recently introduced[4] `AudioWorkletNode` allows custom audio processing separated from the main thread. It depends on loading a worklet script and registering it with the desired audio context. Unlike `BufferSourceNode`, its *r-audio* abstraction cannot be mounted into the audio graph without obtaining the data (in this case, registering the worklet script) first. This prevents audio from being processed without the worklet effect before the worklet script is registered. If the node is mounted without a registered worklet, a `TypeError` is thrown.

3.6 Combining *r-audio* and HTML

By default, r-audio components do not render any DOM elements. This is possible since React v16.0 which removed the requirement to wrap JSX blocks in a single parent element[5]. In debug mode, the audio graph is rendered as a nested list where every node is represented by a `<li>` element, and its parameters rendered as key-value pairs. The debug mode can serve as a standalone tool or an accompaniment to tools such as the Firefox Web Audio developer tool[6].

In addition to rendering debug HTML, *r-audio* components can be mixed with arbitrary HTML elements or React components. There is a number of potential use cases for HTML embedded in audio graphs:

- visualising the audio graph
- providing media elements acting as audio sources (the elements can be passed as props using the React refs system[13])
- creating mixed components which include both UI and audio subgraphs[4]

4. EVALUATION

4.1 Graph update latency and integrity

In Web Audio applications, many updates are scheduled ahead of time and the audio thread ensures precise execution of changes. To support real-time processing in the context of live performance or game audio, instant updating of the audio graph is sometimes necessary. All mutations in *r-audio* are facilitated by React's core algorithms, incurring performance penalties.

To evaluate the impact of React processing overhead, I measured the start and stop routines of two scheduled nodes — `OscillatorNode` and `AudioBufferSourceNode`, and their *r-audio* counterparts. Time stamps were obtained using the User Timing API[10]. Table 1 shows a significant difference between execution latencies.

While React does incur a performance cost, the reconciliation mechanism also optimises graph mutations so the smallest possible amount of work is done while transitioning to a new state. For instance, when a node is replaced by a node of the same type but with different props, React detects the difference between the replaced components is only in the prop value and performs an update accordingly.

[4]At the time of writing, *r-audio* cannot use audio components embedded *within* HTML elements. The suggested way to create mixed components is to use `RPipeline` as the wrapper and including all HTML in a single child component.

Table 1: Average latencies of immediately starting and stopping scheduled nodes (average of 50 measurements, in milliseconds)

Operation	*r-audio*	Native
Oscillator start	56.2	3.28
Oscillator end	46.4	10.2
BufferSource start	53.52	4.12
BufferSource end	50.68	11.24

It is important to note that to fully evaluate the latency impact of both the *r-audio* and React layers more thorough performance testing may be necessary. This would include testing more node types and kinds of updates. It remains to be seen if latency issues prove an issue for users of the library.

4.2 Web Audio coverage

At the time of writing, *r-audio* supports all AudioNodes specified in the Web Audio API. Immediate and ramp-based updates to AudioParams are also covered. Functionality of other AudioParam methods such as `setTargetAtTime`, `cancelAndHoldAtTime` and `setValueCurveAtTime` is not currently covered. Enabling it requires a redesign of the parameter interaction model and is worth investigating as a separate issue. `OfflineAudioContext` is also currently not included in *r-audio*. I decided not to implement `ScriptProcessorNode` as it is deprecated and its functionality can be replicated with `AudioWorklet`.

4.3 Future work

To fully assess the design of *r-audio* it is necessary to test the library within many small and large user-facing applications. More work is required for better interoperability with HTML in React so *r-audio* components can be nested inside HTML structures. Advanced scheduling of AudioParams and offline processing could also be enabled. Lastly, I would like to improve reusability of *r-audio* components even further âĂŞ currently graphs can be reused and shared if exported as functional components (rather than classes). It would be preferable to offer an extensible `RComponent` class to third-party developers.

5. CONCLUSION

r-audio provides an alternative, but nearly complete interface to Web Audio API primitives. In this paper I have shown its design enables rapid creation of complex Web Audio graphs in React. The library simplifies state management and improves the ergonomics of working with Web Audio. The graph components remove the need for explicit connection management and leverage React's reconciliation mechanism to optimise graph mutation. While some performance issues were found, *r-audio* is suitable for numerous audio processing applications and has potential to improve the experience of developing with the Web Audio API.

6. ACKNOWLEDGEMENTS

The author would like to thank Chris Needham, BBC R&D, for invaluable assistance with producing and publishing this work.

7. REFERENCES

[1] A-Frame – Make WebVR. https://aframe.io, Apr 2018. [Online; accessed 16. Apr. 2018].

[2] P. Adenot, R. Toy, C. Wilson, and C. Rogers. Web Audio API. https://webaudio.github.io/web-audio-api, April 2018.

[3] CACM Staff. React: Facebook's functional turn on writing javascript. *Communications of the ACM*, 59(12):56–62, 2016.

[4] H. Choi. Enter AudioWorklet | Web. https://developers.google.com/web/updates/2017/12/audio-worklet, Mar 2018. [Online; accessed 19. Apr. 2018].

[5] A. Clark. React v16.0 - React Blog. https://reactjs.org/blog/2017/09/26/react-v16.0.html, Apr 2018. [Online; accessed 19. Apr. 2018].

[6] Web Audio Editor. https://developer.mozilla.org/en-US/docs/Tools/Web_Audio_Editor, Apr 2018. [Online; accessed 19. Apr. 2018].

[7] Flux: Application Architecture for Building User Interfaces, Dec 2016. [Online; accessed 13. Aug. 2018].

[8] B. Hall. virtual-audio-graph. https://virtual-audio-graph.netlify.com, Apr 2018. [Online; accessed 16. Apr. 2018].

[9] G. Haussmann. Izzimach/react-webaudio. https://github.com/Izzimach/react-webaudio, Apr 2018. [Online; accessed 16. Apr. 2018].

[10] J. Mann, Z. Wang, and A. Quach. User Timing Level 2. https://w3c.github.io/user-timing, Apr 2018. [Online; accessed 23. Apr. 2018].

[11] React - A JavaScript library for building user interfaces. https://reactjs.org, Apr 2018. [Online; accessed 16. Apr. 2018].

[12] Components and Props - React. https://reactjs.org/docs/components-and-props.html#rendering-a-component, Apr 2018. [Online; accessed 23. Apr. 2018].

[13] Refs and the DOM - React. https://reactjs.org/docs/refs-and-the-dom.html, Apr 2018. [Online; accessed 19. Apr. 2018].

[14] S. Volke, B. Bechtold, and J. Bitzer. HTML Web Audio Elements: Easy interaction with Web Audio API through HTML. In *Proceedings of the Web Audio Conference*, August 2017.

APPENDIX

A. EXAMPLES OF R-AUDIO GRAPHS

A.1 Stereo delay line with feedback

If the input signal is stereo, `RSplitChannels` *can be used instead, and* `RStereoPanner` *is not necessary.*

```
<RAudioContext>
    <RSplit>
        <RCycle>
          <RPipeline>
            <RDelay delayTime={.1} />
            <RGain gain={.4} />
            <RStereoPanner pan={-1}/>
          </RPipeline>
        </RCycle>
        <RCycle>
          <RPipeline>
```

```
            <RDelay delayTime={.3} />
            <RGain gain={.4} />
            <RStereoPanner pan={1}/>
          </RPipeline>
        </RCycle>
      </RSplit>
    </RAudioContext>
```

A.2 Stereo delay line with feedback

In this example, `this.audio` *is a reference to a HTML5 Audio element.*

```
<RAudioContext>
    <RPipeline>
      <RMediaElementSource element={this.
          audio} />
      <RCycle>
        <RPipeline>
          <RDelay delayTime={.3} />
          <RGain gain={.8} />
        </RPipeline>
      </RCycle>
      <RGain gain={2} />
    </RPipeline>
</RAudioContext>
```

A.3 AudioParams with transitions

This example also shows a number of advanced routings — e.g. a non-connectable oscillator between a panner and filter in a pipeline; a pipeline leading to a disconnected filter node; an oscillator controling an AudioParam.

```
<RAudioContext>
  <RSplit>
    <ROscillator start={0}
        frequency={330}
        type="triangle"
        detune={detune + 3}
        transitionTime={.5} />
    <RBiquadFilter frequency={1000}
      gain={gain}
      Q={1}
      type="lowpass"
      detune={detune}
      transitionTime={{ gain: 5, detune:
          10 }}
      transitionCurve={{
        gain: 'exponential',
        detune: 'linear'}} />
    <RPipeline>
      <RBiquadFilter frequency={1000}
        gain={1}
        Q={1}
        type="lowpass"
        detune={5}
        transitionTime={.8}/>
      <ROscillator start={0}
        frequency={1}
        type="sine"
        detune={0}
        connectToParam='pan' />
      <RStereoPanner />
    </RPipeline>
    <RPipeline>
      <RBiquadFilter
          frequency={1000}
          gain={1}
          Q={1}
          type="lowpass"
          detune={3}
          transitionTime={.8}
          disconnected />
      </RPipeline>
    </RSplit>
</RAudioContext>
```

A.4 FFT extraction and worklet processing of a MediaStream

```
<RAudioContext>
  <RPipeline>
      <RMediaStreamSource
        stream={this.state.stream} />
      <RAnalyser fftSize={2048}>
      {
        proxy => {
          const data = new Float32Array(
            proxy.frequencyBinCount
          );
// when this function first runs
// there will be no data yet
// in reality one might want to save
// the `proxy` object and call it
    independently
// e.g. inside a `requestAnimationFrame`
    call
          setTimeout(() => {
            proxy.getFloatFrequencyData(
                data);
            console.log(data);
          }, 3000);
        }
      }
      </RAnalyser>
      <RDelay delayTime={.3} bitDepth={4}
          />
      <RSplitChannels channelCount={2}>
        <RAudioWorklet worklet="bit-
            crusher"
            bitDepth={4}
            frequencyReduction={.5}/>
        <RPipeline>
            <RDelay delayTime={.5} />
            <RAudioWorklet worklet="bit-
                crusher"
                bitDepth={4}
                frequencyReduction={.5}/>
        </RPipeline>
      </RSplitChannels>
      <RGain gain={0.4} />
  </RPipeline>
</RAudioContext>
```