Generating Waveforms for Podcasts in Winds 2.0

hackernoon.com/generating-waveforms-for-podcasts-in-winds-2-0-82a32a1c77fa

As developers at Stream, an API for building scalable newsfeeds and activity streams, my colleagues and I have been hard at work on creating Winds 2.0. This version of Winds is an open-source desktop app with support for RSS feeds and easy podcast listening, which we're building in Electron, React, and Node.js.

For more info on Winds 2.0, check out the announcement blog post.

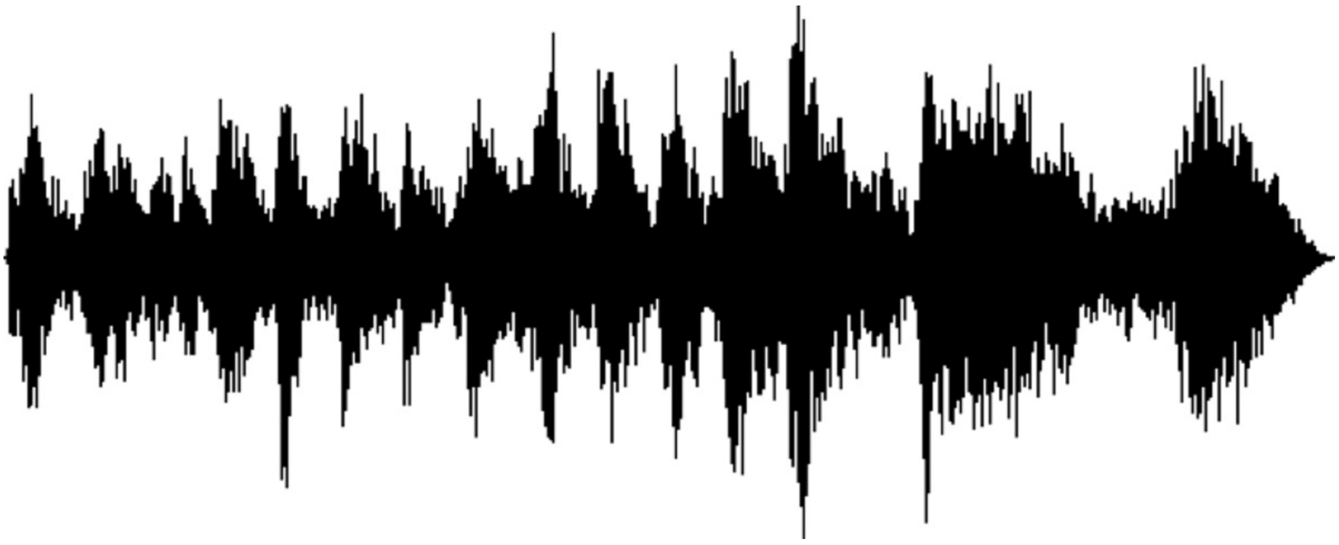
One of the cool new features of Winds is the waveform functionality. In this blog post, I'll talk a little bit about what a waveform is exactly and how to go about creating it.

Want to follow along? Check out the source code on GitHub and then see the finished example.

What is a waveform?

A waveform is a visual representation of the volume over time of an audio file — making it pretty easy to pick out the loud/quiet parts of a song or podcast.

For those visual learners, here's what a waveform looks like:



In Winds 2.0, we've created a stylized waveform component that, in addition to showing the volume over time of an audio file, also shows the progress through the current track. Here's what our designer (who put together the Based UI Kit and Based UI Kit — Mobile) was looking for:

There's gotta be a library for that…

There is! react-wavesurfer is a React wrapper around wavesurfer.js, which is a JavaScript library for playing and displaying audio files in a browser.

Wavesurfer is an awesome project, but didn't quite satisfy our requirements. Wavesurfer uses the HTML5 canvas (which wasn't going to work for us — more on that later!), and it looked like we were going to have to work around a lot of issues integrating the old-school JS library into our new-school React/Redux/Electron app.

We've already built the audio player — we just need a waveform*renderer*…

Winds 2.0 already has a lightweight audio player, which is just a wrapper around react-audio-player. All we need now is a waveform renderer — some kind of component that can take an audio file, and turn it into a waveform.

It's also important to note that we also need to display the **progress** on the waveform, so the user knows how many glorious minutes of sultry radio announcer voice they have left before another pledge drive break.

Part 1: Getting Waveform Data

Before we can render a waveform, we have to take the audio data and convert it into a format that we can actually use to put on a page.

Keep in mind that digital audio files are literally just huge arrays of numbers that, when played in sequence quickly enough, sound like a podcast! Or dark incantations, if played in reverse.

In order for us to grab that data and process it, we'll need to load it into an AudioContext, then use the `getChannelData` method to grab the array of audio values. Basically, we're just turning a JS `ArrayBuffer` into a `Float32Array`:

> The AudioContext object is part of the new Web Audio API, a comparatively new kid on the block for browsers, but is still available in all modern browsers. Totally possible to do this with ffmpeg and other audio libraries if that works better for you!
>
> Also, we're using axios instead of XMLHttpRequest, just because it's a teensy bit easier to grok.

For example, if we use the most recent Car Talk episode, #1743 (well, the most recent "Best of" Car Talk episode, because it went off the air in 2012—RIP Tom) and `console.log` out the `decodedAudioData`, I get something that looks a little like this:

So now I have an array of 143,669,376 32-bit floats. The MP3 has a sample rate of 44,100 samples per second, and runs for 54 minutes and 18 seconds—so, math:

(44100 samples / sec) * (3258 seconds) = 143,677,800 samples total

Which is pretty close to our array's 143,669,376 samples. Neat!

Converting the waveform data into something we can render

At the end of this process, I want an array of 100-or-so values from 0 to 1 that I can map to the heights of each bar on the waveform. (I mean, I could try to map all 143,669,376 values to bars on the waveform, but I feel like that *might*cause some usability problems…)

We're going to use a bucketing algorithm to generate N buckets—each bin will be a representation of the maximum (or average, or minimum, depending on what you're looking for) of the volume of that time interval of the audio file.

As you see above, audio data can be negative! That's just how audio data works—the audio sample can have a value of -1 to 1. However, we don't want negative values in our waveform, so we'll just use the maximum value in each bucket.

Our highly stylized waveform doesn't have to be a 100% precise reproduction of the data in the audio file. We just want an array of 100 bars, each of which indicates the general "loudness" of the audio file in that section.

Here's the bucketing algorithm:

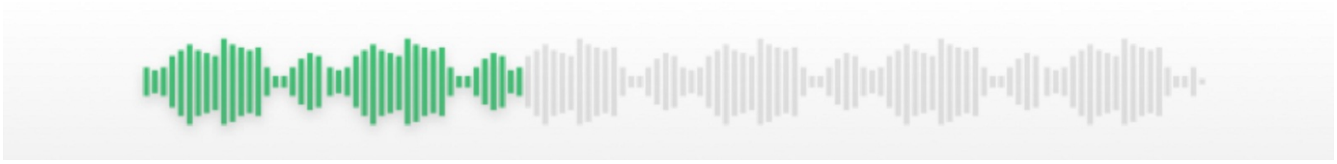
After running our decodedAudioData (the `Float32Array` with 143,669,376 samples) through this algorithm with 100 buckets, we get something that looks like this:

Much easier for us to render this to the page!

Part 2: Rendering our waveform

Okay so—now we've got our array of 100 values from 0 to 1, each of which indicates the max loudness of that section of our audio file. We can use this data to actually render the waveform to the page.

Remember, this is what we want our waveform to look like at the end:



Based on this mockup, we're going to have 3 main requirements:

1. The waveform is an approximation of the podcast volume over time—we've already got that data from Part 1.
2. We need to have some control over the styling of the component: the number of "bars", the spacing between them, color, etc.
3. We need to have the progress bar "shine through" the waveform bars as the track plays ← **this is the tricky one**!

Implementation options

There's a couple of different ways that we can go about implementing this — each with a couple tradeoffs.

Canvas— most of the existing waveform libraries out there use the HTML5 Canvas element.

Pro: Canvas is awesome for 2D and 3D games as well as pixel-perfect implementations of layered or animated visual elements. Sounds perfect for us, right?

Con: Canvas doesn't really satisfy our #2 requirement, which is easy control over the styling of the component. Scaling the waveform "bar" height, width and spacing is tough, and it's also difficult for us to provide easy parameters to change.

(Canvas does provide some ways to use clipping masks, but that wasn't something that I dove super deep into.)

DOM— just a bunch of `div` s!

Pro: It's super easy for web developers to throw a bunch of customization on a DOM waveform using CSS. It scales and stretches nicely, we can set padding, height and width params easily, and we can even add animations as the podcast loads or plays.

Con: What the DOM won't do easily for us is masking/clipping — e.g., having the progress bar "shine through" the waveform as it plays. Sure, we could probably set up some situation where we calculate how many bars to fill based on the progress through the audio file, then calculate the % to fill the "currently playing" bar, but it turns into a bit of a mess.

SVG— `rect` s, `viewBox` and `preserveAspectRatio` .

Pro: SVG gives us *some* styling options — (fill color, rectangle border radius and other SVG attributes can all be set via CSS). Most importantly, SVG allows us to do "masking" (or clipping) on DOM elements.

Con: One drawback of using SVG is we don't get layout control via CSS. Instead, we'll have to use absolute positioning for each of the "bars". However, SVG does give us a couple stretching and scaling options, like `preserveAspectRatio="none"`. With JavaScript, we can specify those parameters (# of bars, space between bars, etc), and use *math* to figure out where to draw them.

How SVG masking works

We talked about the different ways we could implement our waveform, but the main requirement we were circling around was “masking”, or letting the progress bar “shine through” the waveform.

There’s 3 parts to making our SVG mask work:

1. The element that’s going to serve as a “progress bar”—animating a rectangle from 0% width to 100% width.
2. The actual mask: the “fill” lines and zones where the progress bar should “shine through”.
3. The CSS to attach the two together - `clip-path`.

Now, remember this is SVG, which means that style declarations and attributes will look slightly different, but operates on the same principles when styling DOM elements.

The first part—our “progress bar” element along with a “background” element:

The `viewBox` attribute defines the cartesian grid space for our SVG, with (0,0) starting in the upper left corner of the element, going to (100, 100) in the lower right.

The `preserveAspectRatio` attribute is the opposite of what you normally see on SVGs - instead of using “meet” or “slice” (which ensure the image is expanded/shrunk to fit in the SVG box, or cropped to fill the SVG box), we’re using “none”, which forces the image to stretch and scale to fit whatever CSS dimensions we throw at it.

Finally, we define two SVG rectangles—the first one is our waveform “background” (the gray color across the entire waveform) and the second is the actual “progress bar” that animates the width as the track plays.

Those two SVGs aren't the bars on the waveform! They're just the background and progress bars. In order to create the bars, we need to create another SVG element, this time with a height and width of 0:

This SVG has a height and width of 0, because we don't actually want to render it to the page. Instead, we'll reference it when connecting it to the progress and background SVGs. We're also using some lesser-known SVG tags like
`defs` (sort of like SVG "templates") and `clipPath`, which is *exactly* what we want to use in this situation.

Because we don't know what the bars look like until the page loads (and fetches the audio data), we'll use JavaScript to create a bunch of `rect`s— one for each bucket:

> In Winds 2.0, we've built everything with React, so it'll look slightly different in the open source repo :) Vanilla JS is great for illustrating the concept!

Keep in mind, we're drawing these bars in a 0-to-100 coordinate space, because that's what the other SVGs on the page use. The bars will stretch and scale with `viewBox` based on whatever CSS height and width are set.

Remember that buckets is just our array of N values from 0 to 1, indicating the general loudness of the track over this time interval. We iterate over all the buckets, and then use some Math™ to figure out exactly where to place these rectangle bars:

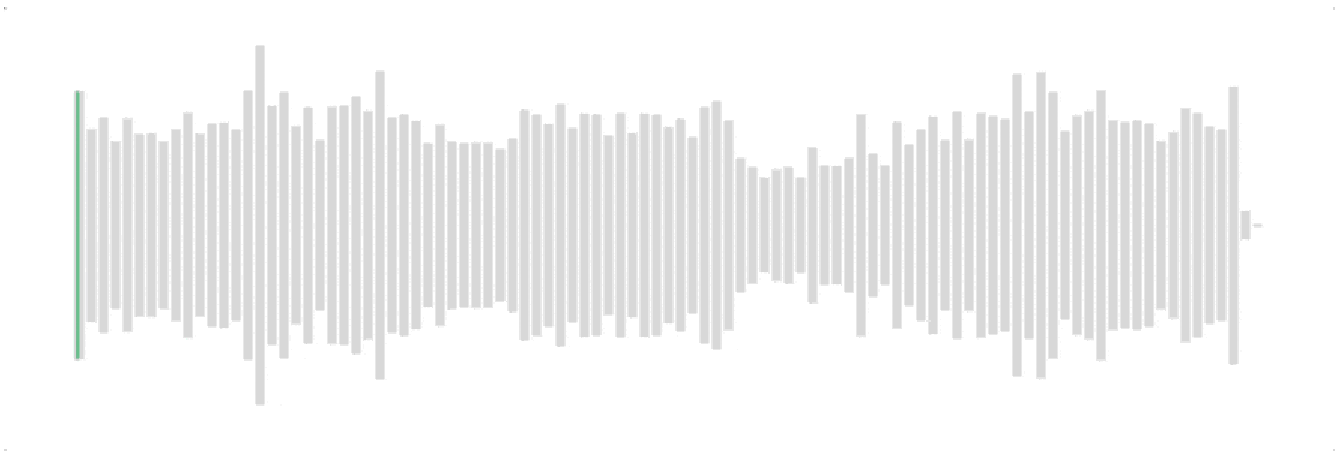
1. The starting X coordinate of the rectangle, from the left side of the SVG box, plus ½ the spacing between each bar.
2. The width of the rectangle, which is the number of buckets / 100, minus the spacing.

Finally, we've got the CSS that ties the two (well, three, if you count the background) SVG elements together:

This selects both of the "background" and "progress bar" rectangles and says "use the `#waveform-mask` SVG as a mask" - as if the `#waveform-mask` SVG was just a paper cutout over the progress bar, and we can see the pixels through the cutout.

Don’t forget! We need to update the width of the `.waveform-progress` SVG while the track is playing. Depending on the audio player implementation, there might be an `onListen` callback, or maybe we’re just using `setInterval` to fire every 500-or-so milliseconds. All we have to do is divide the track’s current progress by the length of the track, multiply by 100 and change the width of the `.waveform-progress` rectangle.

The end result



Boom! Look at that — our episode has some loud parts, some quiet parts, the audio plays, the progress bar fills in… and somehow we’ve wasted another perfectly good hour listening to Car Talk.

If you’re interested in implementing something like this in your application, you can check out this full working example and the source code.

In this example, I’ve just used an `audio` element as the audio player. I’m checking the progress of the audio player every 100ms, then adjusting the width of the progress bar.

In nontrivial applications, the audio implementation will vary a lot — for example, in Winds 2.0, we’ve got separate React components for `PlayerContainer`, `ReactAudioPlayer` and `Waveform`. `ReactAudioPlayer` calls an onListen prop every 500ms, which calculates the progress, stores in PlayerContainer’s state, then renders the `Waveform` component with a `progress` prop.

What’s next?

Just one network call, please.

In the sample code, you might notice that there’s actually two network calls to get the /car-talk.mp3 file—one from our script, which fetches the audio in an `ArrayBuffer` format, and one from the `audio` element on the page.

Downloading 30 seconds of bluegrass music twice? Not a big deal! 2 hours of a Dungeons and Dragons game, downloaded twice? Probably going to hammer the user’s network connection.

Unfortunately, this is a far trickier problem than it initially seems, and something I’m still sorting out. My best suggestion is to either:

- Grab the audio data (via XHR, in the `ArrayBuffer` format) and load it into an `audio` element.
- Or, do the reverse—have the `audio` element grab the audio data, then pull the data out of the element.
- Or, begin the dark incantations to invoke `AudioContext`, `AnalyzerNode` and `AudioDestinationNode`—and play audio only using the Web Audio API.

Maybe maximum loudness isn’t the variable we should use?

I’d also like to improve the visualization a little bit—some podcasts have nice little soundscape sections in-between chapters (thanks, Jad and Robert), but most spoken word podcasts are pretty level across the whole episode (thanks, Roman Mars). The waveform is accurate, but not necessarily useful—if I want to skip to another chapter, I’m usually looking for the dip/spike in audio where there’s intermission music or silence.

It might be as simple as taking the average or minimum volume over the bucket interval. Alternatively, we can get more complex by figuring out some kind of nonlinear scale to use, or calculating the maximum volume and scaling everything from there. Just something to experiment with!

But what if I want more bars? Or to round the edges of the bars?!

And make the component responsive!?

These are mostly just limitations of SVG—it's a 90% solution. Rounding the corners of a `div` is trivial, but rounding the corners of really tiny SVG rectangles doesn't work well. And, if we resize the overall SVG, the number of bars stays the same, but the space between the bars scales. The waveform isn't very responsive, so we're better off sticking to a fixed width value when sticking it on a page.

Cue the outro music

Other than the few SVG limitations, this component is awesome! Instead of having to do all this server-side with ffmpeg or some other audio processing library, we can have the user's client handle all waveform generation.

We're working hard on Winds 2.0 so that you can listen to all of your favorite podcasts and read your favorite RSS feeds within one app. It's coming along very nicely, so look for it sometime early next year. For the time being, check out another one of our sample applications, Cabin.

> This is a collaboration from the team at GetStream.io, led byKen Hoff, Developer Advocate at GetStream.io. The original blog post can be found at https://getstream.io/blog/generating-waveforms-for-podcasts-in-winds-2-0/.