Figure 1: The `maestro-cli` icon, which I made myself in Linearity Curve (inspired by MacOS's Terminal icon).

Diving into the music visualizer in

`maestro-cli`: A command-line music app

https://github.com/PrajwalVandana/maestro-cli

1. How it works

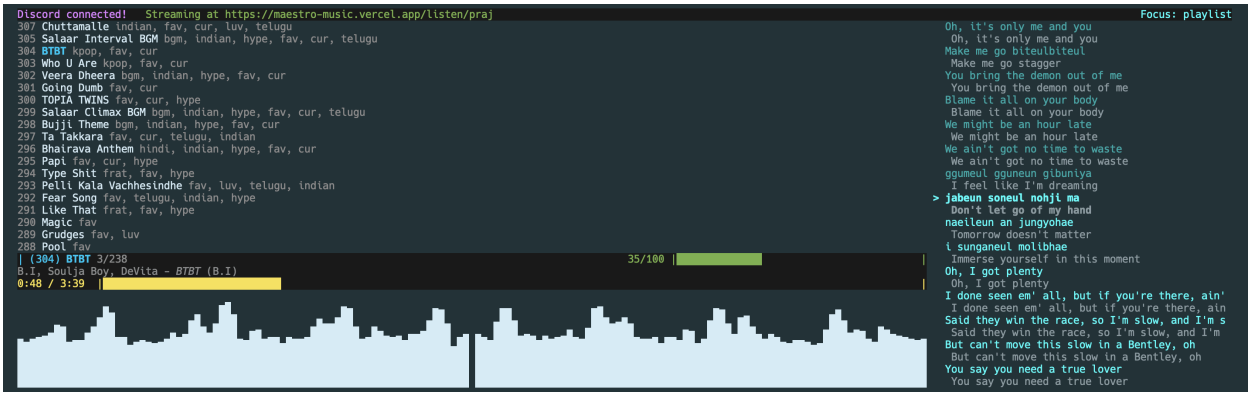

Figure 2: The music player screen, with the visualization playing gloriously at the bottom.

Ever since I first saw music visualizers on YouTube, I've been fascinated by their bouncing bars and pulsing dots. Creating my own, though proved to be a daunting challenge, even in the face of the programmer's most formidable tools: Google and StackOverflow. It took months, but I was finally able to create a visualization that I was proud of.

The visualization process runs in a background thread, so that it can be rendered at the bottom of the screen while songs play; but the thread also needs to be able to communicate with the main thread, so that it knows when the user pauses, moves to a new song, etc.

Here's how it works (for just one song):

1. First, we load the song data into memory using the `librosa` package. The song data is stored as a 2D `numpy` array, where each row represents a channel, and each column represents a time.

$$\begin{bmatrix} [L_0, L_1, L_2, L_3, ..., L_n] \\ [R_0, R_1, R_2, R_3, ..., R_n] \end{bmatrix}$$

   Here, L_t and R_t represent the left and right channel values, respectively, at timestep t. Some songs may have more than 2 channels, such as those with 5.1 surround sound (6 channels)—or less than two channels, such as those with mono sound (1 channel); these are converted to stereo sound (2 channels) for the visualization.

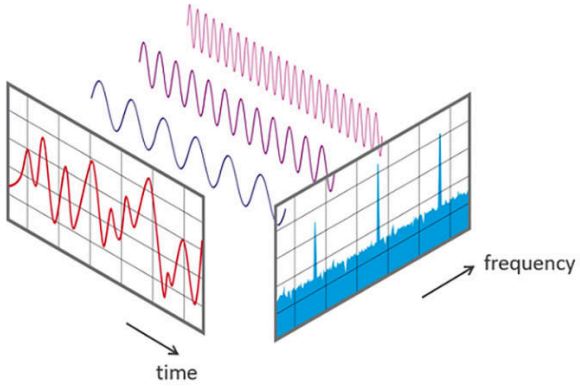

Figure 3: A graph depicting how the Fourier transform works for a single window (from Microsoft Community Hub).

2. Then, we use the `librosa.stft` function to run the short-time Fourier transform on the song data. This function splits the song data into a series of (slightly) overlapping time windows, and then runs the discrete Fourier transform on each window (the red graph in Figure 3 represents one window). The result is a 3D array, where each row represents a channel, each column represents a window, and each "layer" represents a frequency.

$$\begin{bmatrix} [l_1, l_2, ...] \\ [r_1, r_2, ...] \end{bmatrix}$$

   Here, l_1 and r_1 are arrays that represent the left and right channel frequencies, respectively, at the first window, l_2 and r_2 represent the left and right channel frequencies at the second window—and so on. Each array l_i or r_i is of the form

$$[F_1 \;\; F_2 \;\; ... \;\; F_n],$$

   where F_i is the amplitude of the i-th "frequency bin", i.e. how strong the frequencies in that bin are at that point in time (the discrete representation of the blue graph from Figure 3). The i-th frequency bin corresponds to the frequencies centered at $i \times \frac{s}{n}$, where s is the sampling rate (how many values we have for each second of audio) and n is the total number of frequency bins (in my code, I used the `librosa` default of $n = 2048$). The concept of frequency bins is necessary because we can only store a finite number of values in a computer, so the continuous output of the Fourier transform must be discretized into a finite number of bins.

3. We're close to the finish line! Now, we just need to convert the frequency data into a format that we can use to draw the visualization. First, we convert the amplitudes to decibels using the `librosa.amplitude_to_db` function, which returns values from −80 to 0; we then shift this range to 0-80 by adding 80 to each of the decibel values.

4. Finally, before we render to the screen, we need to check how wide the terminal screen is. The visualization has two parts, the left and right channels, separated by a gap for visual clarity. If the screen width (in characters) is odd, then we can simply divide the screen width by 2 and round down to get the width of each channel. If the screen width is even, then we need to subtract 1 after dividing, so that there is a gap. Also, we have 2048 frequency bins, but the screen width (let's call it w) is much smaller than that, so we split the frequencies evenly into new bins such that there are w bins. For each bin, we take the maximum value (taking the average tends to skew the values lower, which looks bad), and then scale it down to the visualizer height (48). This gives us the height of each bar in the visualization, which is then rendered using Unicode block characters (6 different heights, visualization is 8 characters tall, for a total of 48 possible heights).

2. The code

Here's a (very) simplified and trimmed version of the code for rendering the visualization:

```
def bin_max(arr, n, include_remainder):
    remainder = arr.shape[1] % n
    if remainder == 0:
        return np.max(arr.reshape(arr.shape[0], -1, n), axis=1)

    head = np.max(arr[:, :-remainder].reshape(arr.shape[0], -1, n), axis=1)
    if include_remainder:
        tail = np.max(
            arr[:, -remainder:].reshape(arr.shape[0], -1, remainder), axis=1
        )
        return np.concatenate((head, tail), axis=1)

    return head

def render(  # the function that renders the visualization
    num_bins,
    freqs,
    frame,
    visualizer_height,
):
    gap_bins = 1 if num_bins % 2 else 2
    num_bins = (num_bins - 1) // 2  # total screen width ==> width of each channel

    num_vertical_block_sizes = len(config.VERTICAL_BLOCKS) - 1
    # compress to visualizer width and round/scale down to visualizer height:
    freqs = np.round(
        bin_max(
            freqs[:, :, frame],
            num_bins,
            (freqs.shape[-2] % num_bins) > num_bins / 2,
        ) / 80 * visualizer_height * num_vertical_block_sizes
```

```
    )

    arr = np.zeros((2, visualizer_height, num_bins))  # convert to block heights
    for b in range(num_bins):
        bin_height = freqs[0, b]
        h = 0
        while bin_height > num_vertical_block_sizes:
            arr[0, h, b] = num_vertical_block_sizes
            bin_height -= num_vertical_block_sizes
            h += 1
        arr[0, h, b] = bin_height

        bin_height = freqs[1, b]
        h = 0
        while bin_height > num_vertical_block_sizes:
            arr[1, h, b] = num_vertical_block_sizes
            bin_height -= num_vertical_block_sizes
            h += 1
        arr[1, h, b] = bin_height

    res = []  # convert block heights (from `arr`) to strings for rendering
    for h in range(visualizer_height - 1, -1, -1):
        s = ""
        for b in range(num_bins):
            s += config.VERTICAL_BLOCKS[arr[0, h, num_bins - b - 1]]
        s += " " * gap_bins
        for b in range(num_bins):
            s += config.VERTICAL_BLOCKS[arr[1, h, b]]
        res.append(s)

    return res


# load song data
audio_data = librosa.load(song_filename, mono=False, sr=config.SAMPLE_RATE)[0]

# convert to stereo
if len(audio_data.shape) == 1:  # mono -> stereo
    audio_data = np.repeat([audio_data], 2, axis=0)
elif audio_data.shape[0] == 1:  # mono -> stereo
    audio_data = np.repeat(audio_data, 2, axis=0)
elif audio_data.shape[0] == 6:  # 5.1 -> stereo
    audio_data = np.delete(audio_data, (1, 3, 4, 5), axis=0)

# run short-time Fourier transform, convert to decibels, and shift range to 0-80
visualizer_data = librosa.amplitude_to_db(
  np.abs(librosa.stft(audio_data)),
  ref=np.max
) + 80

# move cursor to start of visualizer
stdscr.move(stdscr.getmaxyx()[0] - config.VISUALIZER_HEIGHT, 0)
```

```
rendered_lines = render(  # calculate visualization
    stdscr.getmaxyx()[1],  # screen width
    visualizer_data,
    min(round(pos * config.FPS),visualizer_data.shape[2]-1),  # frame #
    config.VISUALIZER_HEIGHT,  # visualizer height (constant)
)
for i in range(len(rendered_lines)):  # render visualization
    stdscr.addstr(rendered_lines[i])  # draw line
    if i < len(rendered_lines) - 1:
        stdscr.move(stdscr.getyx()[0] + 1, 0)  # move to next line
```

3. Further exploration

I've also played around with visualizations outside of `maestro-cli`, creating a small tool that generates a visualization video for any song, complete with color based on the intensity of the music. The effect doesn't translate well to a still image, but here's a link to the visualization of "Smooth Criminal" by Michael Jackson, which I generated for a friend (unfortunately, before I implemented colors): https://drive.google.com/file/d/17mnncW9mrrE4MlFmOAcA9Gd5smzMw7or/view

Here's a still from "All Time Low" by Jon Bellion, which features colors:

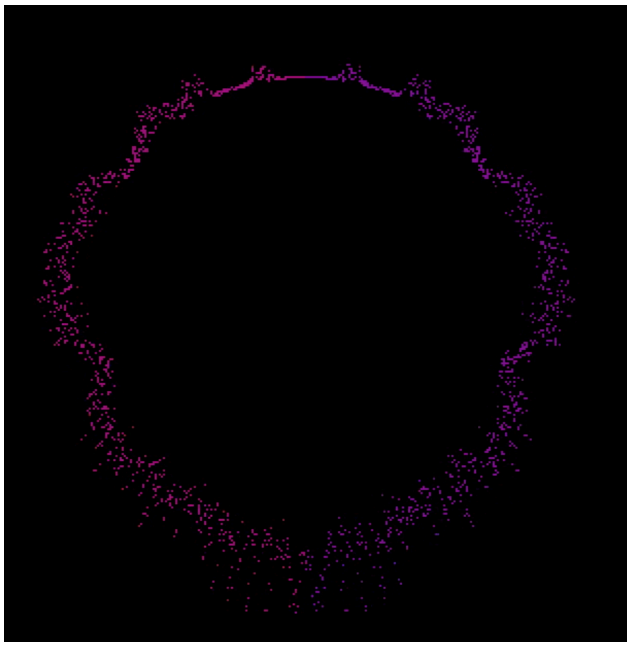

Figure 4: A still from the visualization of "All Time Low" by Jon Bellion.

4. Conclusion

This brief document doesn't truly do justice to all the dead ends I hit as I researched how to create a visualization, or the roadblocks I faced in coding not only a working, but fast and smooth implementation. Yet I hope that this overview gives you a sense of the complexity of the application, and how much I learned from creating it. Oftentimes in CS, the most difficult problems are the most rewarding to solve, and this project was no exception.