Synesthesia: Audio Visualizer

COS 426 Spring 2022
Professor Felix Heide
Dawn Luong

Abstract

This project is an audio visualizer that draws a trail based on user input and the audio data of a given audio file. The user is able to control where the trail moves while the audio file determines the shape of the trail. The user can also customize the visualizer's color and mode. The end result is an art piece that is representative of the input audio with the influence of the user.

Introduction

The goal of this project was to create an audio visualizer that would take in user input. This would create an interactive experience for the user and result in an art piece that is dependent on the data of audio. Given a user-inputted audio, my goal was to process the features of the audio as input into an audio visualizer. The audio visualizer would draw a trail based on the control of the user and while changing based on the audio data. This would create an art piece that is unique to not only the audio but also to the user's input. The visualizer would ideally be representative of each unique song.

There have been a multitude of audio visualizers as the area is so vast and can be done in many ways. Most audio visualizers represent audio data at an instance of time and as the audio changes, the entire visualizer changes. As a result, only instances of the audio are captured rather than an encompassing representation of the entire audio. Other audio visualizers take in user input however they do not always use the input to affect the output of the visualizer. For example, previous audio visualizers have used input to move the user around in relation to the positional audio without impacting the visualizer itself [1].

For this project, I took the approach of having user input control the trail that is drawn by an audio visualizer. Rather than solely being a curve made by the user, the trail is altered by the frequency data of a given audio. At each sample of an audio, the trail grows and the position at that point of the trail is determined by the user input in addition to the audio data. This should work well for any audio. Since the visualizer is simply a representation of the frequency data of an audio, any audio with frequency data can be represented. However, the visualizer should be run on a laptop or desktop with a keyboard, and a chrome browser. If a user wants to have the visualization be more representative of the audio, they can minimize their input and leave most of the trail dependent on the audio instead.

Methodology

To execute my approach, I need to implement an audio loader, a third person camera, and a trail. For the audio loader, I used three.js audio api. This allowed me to use abstracted functions to load a song, given a file, and compute the frequency audio of the data at each instance. For the third person camera, I needed the camera to follow the trail. Therefore, I created an object that was controlled by the user. This object was used as the target for the camera to follow. In order for the camera to be in third person perspective, I had to include an offset for the position of the camera in regard to the target that the camera looked at. To do this process, I followed a tutorial [2]. To control the object and therefore the camera, event listeners were added to detect user input. The keyboard was used as input to control the camera, W = up, A = left, D = right, S = down. To load the audio, an event listener was needed to detect when a user uploaded the file. Once this occurred, the visualization would start. To pause and play the song, and therefore the visualization, keyboard input was used, shift = pause and space = play.

For the trails, I used Line from three.js with its colors dependent on the user input. At each animation frame, a sample of the audio was taken and based on that sample and the position of the camera, an additional vertex on the line would be rendered. To create a desirable helix shape, I used the current position as the center of the circle and took a point on the circle, using the current time elapsed as the angle. The vector that the angle was taken from was computed by determining a vector that is perpendicular to the direction vector of the camera. The radius of this point is dependent on the value of the frequency data of the audio. For each frequency bin, a line/trail is drawn, each point on the line is representative of the amplitude of that frequency bin at the given time.

I also included multiple modes and a dat.gui so that there could be more user interaction. The GUI takes in input for the rate of turns that the helices of the trail spin at, the radius of the helix, the window size that the frequency data is sampled at (fftsize), trail color, and background color. Turns of 0 means that there is no turning of the helices and the trail will be straight, given no user input. Fftsize is the window size of a Fast Fourier Transform. A higher value results in more details in the frequency domain while a lower value results in more details in the time domain.

The different modes indicate different visualizations. Default uses the frequency data, as explained above, as the radius of each point on the corresponding trails. Color-code mode uses the same method as default; however, each frequency bin is color coded based on its frequency. Higher frequency bins are colors of higher frequency (i.e., purple), while lower frequency bins are colors of lower frequency (i.e., red) [3]. Single mode uses the same concept of making the radius of the helix based off of frequency data, however, the visualizer is only a single trail as it represents the average frequency of a sample at each point.

What I wanted to implement but did not was the ability to move the camera around by clicking and dragging once the song was finished. What was implemented

instead was to move the object/camera around the way it would be moved when the song was playing without the trails continuing to build. This allows the user to maneuver around and see the finished product but in an awkward and unintuitive way. Another feature I wanted to implement was to add more modes of visualization. As audio visualization is a very broad category, even within this visualizer, there were so many different ways the trails could've been drawn. Given more time, I would've liked to play around with the data that could be obtained from the audio to represent different aspects of the audio within different aspects of the trails, instead of just the radius and color of the helices. These features were not implemented due to time constraints and difficulty of learning three.js.

Results

I measured success by determining if all the MVP features required for the project to be usable were implemented correctly. I debugged through my implementation and took advantage of JavaScript's ability to output values through console.log to ensure that the values that I expected were actually being set. It is important to note that the current implementation has some technical bugs. When the trail is drawn, an error occurs and shows up in the console, however, there is no visual indication of this bug, so the audio visualizer still works as desired. Additionally, the calculation of the vector perpendicular to the direction of the camera movement has a bug where if the user changes the direction of the trail, there are instances where the vector isn't computed properly, resulting in a disturbance in the visualizer. Additionally, the way that the controls and implementation are set up, the user is able to press space to play the song again once it has ended, allowing for the trail to continue building. While this may be an advantage in some cases, the original plan for the project was to not have this occur.

The results indicate that while the audio visualizer works as intended, there are a few bugs and changes that need to be made to have a fully-fledged product.

Discussion

The approach taken in this project as to how to make an audio visualizer seems relatively promising. The project resulted in an aesthetically pleasing, user-controlled visualizer that is essentially a drawing tool, dependent on audio input. Although there may have been more efficient ways to implement this project, the overall approach works in regard to what I planned for it to do. Other approaches may have considered different ways to process audio or represent audio. Follow-up work could be done to explore these possibilities and examine their results. By simply changing the way audio is processed or the audio data that is used, the visualizer can change drastically. By implementing a full graphics project, I learned more about audio processing and programming with javascript/three.js/npm.

Conclusion

I think that my goal was attained relatively effectively. While there are some bugs and features lacking, the project serves as a proof of concept that this kind of audio visualizer works and can serve as a way to create audio-based art. Next steps would first be to add in-app instructions as a way to onboard the user into the visualizer and explain how features work, using javascript/html. As of right now, the only way for the user to know how to use the visualizer is by reading the readme. After that, additional steps would be to revisit the bugs that were discussed earlier as well as optimize any code to keep the program from running inefficiently. Additionally, further features as discussed in the Results section could be implemented.

Works Cited

[1] https://github.com/mrdoob/three.js/blob/master/examples/webaudio_visualizer.html
[2] https://github.com/simondevyoutube/ThreeJS_Tutorial_ThirdPersonCamera
[3] https://stackoverflow.com/questions/10731147/evenly-distributed-color-range-depending-on-a-count