VISAUDIO: USER DRIVEN AUDIO VISUALIZATION DATABASE

By

Leon Zhuang

Signature work, UHP Project, submitted for
completion of the University Honors Program

June 5, 2022

University Honors Program
University of California, Davis

APPROVED

glenda drew

Glenda Drew
Department of Design

Director of the University Honors Program
Associate Dean, Undergraduate Education

Table of Contents

- **Abstract (pg. 3)**
- **Introduction (pg. 4)**
- **Background Information (pg. 6)**
- **The Project (pg. 8)**
    - **Project Purpose and Goals (pg. 8)**
    - **Project Creation and Implementation (pg. 10)**
    - **Project Evaluation (pg. 36)**
- **Project Results & Discussion (pg. 39)**
    - **Limitations (pg. 39)**
    - **Learning Outcomes (pg. 41)**
    - **Future Work (pg. 42)**
- **Bibliography (pg. 44)**

Abstract

Music evokes a variety of responses in a listener. Emotions - nostalgia, happiness, or sadness. Or memories - of better times, or of a loved one. Or even mental imagery - a serene forest, or perhaps a frozen tundra. The same song can evoke vastly different responses and experiences in different people. How can we communicate what music evokes in us? One possibility is audio visualization; it can be experiential and thought-provoking. A 2018 journal article, titled “Study on Application of Audio Visualization in New Media Art”, establishes a link between musical notes, visual imagery and emotions. *Visaudio* is a web application where users can choose a song and design a visualization based on what the song evokes. Users can then submit their visualization for others to experience. *Visaudio* uses the Web Audio API and one or more JavaScript graphics libraries for the audio visualization, and a database to store and retrieve users’ visualizations. By creating and sharing visualizations of music, my hope is that users will be able to convey their thoughts and feelings and gain a deeper understanding of what specific musical pieces mean to each of us.

Introduction

Why did I create *Visaudio*? The idea for this project was born from my own personal interests. I have been a music enthusiast since middle school. I am also a programmer (for not nearly as long as I have been a music fan, but coding is sure to be a lifelong hobby as well) with a specific interest in web programming. Thus, I decided that my project would definitely be a web application centered around *some* aspect of music. The next question was exactly *what* aspect it was going to be. One obvious idea was a music streaming app, similar to Spotify or Apple Music. However, I had already built one previously (albeit with significantly less functionality), and I wasn't interested in building another one. This time around, I wanted to go a more artistic route.

Listening to music is a mental activity. Not only can it evoke strong emotions in a listener, it can also make them recall memories, think of otherworldly landscapes, and much more. I can personally attest to that. What if we could somehow visualize these things? Everybody's life experiences are different, so everybody's headspace could be wildly different, even when listening to the same song. What if people could not only create their own visualizations, but also experience other people's visualizations as well? This is the essence of what *Visaudio* aims to achieve.

Why is *Visaudio* important? I want this project to help people understand one another better. As a real life example, a person may see somebody listening to music that they are utterly repulsed by. They just can't understand what the other person hears in the music. But perhaps, after seeing their thoughts visualized, they may begin to understand, and perhaps even start to enjoy the music themselves. Another goal of *Visaudio* is to help people discover new music. A user will definitely be exposed to new music when viewing other user's creations.

Additionally, the project could potentially have scientific value. If enough users submit visualizations to achieve a sufficient sample size, the results could lead to some interesting insights. For example, I could infer something about an average person's mental reaction to a song by looking at user submitted visualizations for that song and comparing how similar/different they are.

Background Information

The idea behind, and my decision to build, *Visaudio* came not just from myself, but multiple people as well. In the late fall quarter, I was still deciding if I even wanted to do the Signature Work. Eventually, I decided that I wanted to get the experience of working on a project with a professor in my final quarters at Davis. I wanted to work on a web application. I asked Professor Glenda Drew to be my advisor, since I had taken two web design classes taught by her, and I knew she could provide invaluable advice and critiques on my web design, a weak point of mine.

Initially, I wanted to work on one of Glenda's projects. Coincidentally, Glenda, along with Bill Mead (Lecturer in Design), were in the beginning stages of a collective memory project. Essentially, a collective memory project was a project focused around collecting specific info from many people (based on the theme of the project) and displaying it in a fashion appropriate to the theme of the project. It sounded interesting to me, so I asked to work on their project. Over the winter break, I had several meetings with Glenda and Bill to brainstorm ideas for the project (it had not been set in stone yet). I brought forth several ideas, but the one that I was particularly invested in was the idea of a project that gave anybody the power to visually express the emotions and things they thought of when listening to a song. For example, if somebody thought of a freezing tundra when listening to a song, they would be able to draw that tundra on the screen and share it with others.

Eventually, I realized that our visions for the project differed greatly; Glenda was aiming for a learning-themed project, and I wanted to do something music related. Both Glenda and Bill encouraged me to work on something I was truly passionate about, so I ultimately decided to work on my own project that was separate from the faculty project. Of course, the idea that I

chose was the music visualization idea which would eventually become *Visaudio*. I am thankful to Bill and especially Glenda (who advised me from start to end) for pushing me to pursue a project I truly cared about.

The Project

Project Purpose and Goals

The purpose of this project was to build a platform where anyone could create and share visualizations of their thoughts and feelings when listening to a song. To achieve this, there were several goals I defined and strove to achieve. One, accessibility: the platform should be accessible to the general audience. Two, user freedom: the visualization creator should give users the tools to make their visualization on the screen as close as possible to the visualization in their minds. Three, sharing capability: users should be able to share their visualizations with other users.

To achieve accessibility, the platform had to be easy to access, learn and use by anyone, anywhere. To make the platform accessible by anybody around the world, the only reasonable choice was to implement it as a web application. To make the platform easy to learn and use, the user interface had to be user-friendly. Additionally, the visualization creator had to be easy to learn and use by anyone. For example, the application could not require users to write code to create visualizations, as not everybody was a programmer. To determine exactly what the most user-friendly and easy to use interface was, I planned to do several iterations of brainstorming, prototyping, and user testing.

To achieve user freedom, the visualization creator had to give users the ability to create anything they wanted. For example, the user had to have the ability to create any type of shape that they wanted. If users could create anything they wanted, then it followed that they would be able to create a visualization matching the one in their heads. Unfortunately, I realized this goal was difficult to realistically achieve. The amount of customization features that could be added to

an application was infinite. Nevertheless, I planned on working towards this goal by brainstorming and implementing as many customization features as possible.

To achieve sharing capability, I needed to implement a database that users could save their visualizations to and request visualizations from. Fortunately, there was a variety of existing database solutions for web applications. I planned to use the Back4App database.

Project Creation and Implementation

The creation of *Visaudio* involved a combination of brainstorming, researching, digital sketching, getting user feedback, and programming. These actions were not accomplished in any particular order; for example, at one point I was simultaneously designing the interface of one part of the application and coding another part of the application. However, there were some tasks that, as a general rule, were completed before others. For example, I designed and got feedback on the user interface before I implemented it in code. In this section I list in detail each major step in the creation of my project in chronological order.

The first step was to conduct a comparative analysis of 3 existing web-based audio visualization projects. For each project, I documented the technologies used, major features, and similarities/differences to my project. Technology-wise, I found that for graphics, WebGL, HTML Canvas, d3.js, and p5.js were all equally viable options; for audio visualization, the Web Audio API was the only option for analyzing audio data to create audio-reactive graphics; for the source of music, the SoundCloud API could be used. None of the projects offered a custom visualization creation system. The most freedom a user had was the ability to switch between pre-defined visualization options. Ultimately, the comparative analysis gave me an idea of how the visualization system might be implemented, including viable technologies and possible features.

Next, I conducted a technology research and assessment. I wrote down the main features of the application. For each feature, I researched and assessed some viable technologies I could use to implement it and ultimately decided which one to use. First, the application was essentially a full-stack web application, so I needed to decide the main technologies I would use to power the front-end and back-end. For the front-end, I had a variety of choices. HTML was

the standard format to display content in the browser. On the other hand, I could choose among a variety of CSS and JavaScript frameworks and libraries. The main advantage of frameworks and libraries was that they made life easier for the developer by making it easier to accomplish certain tasks and in fewer lines of code. However, from my personal experience, they also came with a learning curve, and they could interact badly or could be incompatible with other libraries. Even though I could see how a framework like React or Vue would be appropriate for my application, I decided to stick with vanilla JavaScript. Given the limited timeframe to complete the project, I had concerns that if I used a framework, implementation of features could be too slow, as I would have to make any libraries I use function within the constraints of the framework. Also, if I ran into unacceptable issues halfway through the implementation, such as incompatibility of libraries, I likely wouldn't be able to find a solution or migrate to vanilla JavaScript in time. I was confident in my ability to implement the core of the application in vanilla JavaScript. Anything I could accomplish using a language framework, I could accomplish using the vanilla language itself; after all, the framework was itself written in the language. A similar thought process went into my decision to use vanilla CSS.

Next was the back-end of the application. I would have to host my application somewhere. Github Pages was the simplest option. It was free and easy to get running. However, it was unable to execute server-side code. Despite this, it could still interact with database services that provide an API via calls to the API, which could be done in the client side code. My planned features didn't require any server-side code, and a database service met my database needs, so I settled on Github Pages.

I also needed to decide what technologies I would use for audio visualization. First, I needed a music streaming API. My one essential requirement for the API was that it had to be

able to play a wide variety of music, because my target audience, the general population, listened to a wide variety of music. The more music my application offered, the more likely it was that a user would be able to find a song they wanted to visualize. After some research, I had a few candidates that met this basic requirement. The Napster API was easy to set up and provided an easy way to implement searching functionality, which was very important, as users needed to be able to search for songs. A downside of this API was that only 30 seconds of audio are provided for each song, which limited user freedom. For example, a user may have wanted to visualize a particular section of the song, but would be unable to do so if the 30 seconds provided didn't overlap. Another option was the SoundCloud API. It also provided searching functionality, and it provided full track playback, an advantage over the Napster API. Unfortunately, from my research, access to the API closed around 2019, and it was unlikely that it would open again in the near future. So the SoundCloud API was not a viable option. The last option I researched was Spotify's Web Playback SDK. It also provided full track playback and search functionality, but a major caveat was that it required a premium subscription. That meant my target audience would shrink to only premium Spotify users, and it introduced additional work from the user to login before creating visualizations. I wanted my application to be usable by anybody, and I wanted users to jump right in and start playing around with the visualization creator without doing any additional work beforehand. As a result, Spotify's SDK was not acceptable. Ultimately, I decided to use the Napster API, despite its limitations.

Now that I had the "audio" component of "audio visualization", I needed the "visual" component, which meant I had to research graphics libraries. D3.js was one option that I had worked with in the past. One of the projects I assessed in my comparative analysis utilized D3.js, so I knew it was a feasible option. However, from prior experience, it was low level and

unintuitive, and therefore not easy to work with. Another option was p5.js. Drawing a shape to the canvas took just one line of code, and the library was well documented, simple to set up and easy to learn. As a bonus, an addon library, p5.sound, made interacting with the Web Audio API simple. For example, it provided access to the FFT (fast fourier transform) algorithm, which was the de facto method for audio visualization. FFT could isolate individual frequencies in an audio waveform, which made it possible to accomplish things such as making one graphic element react to the bass of a song, and another element react to the treble of the song. The p5.sound library made all of this readily achievable. Some other libraries I considered were Three.js, which was centered around making 3D graphics, and Tone.js, which provided extensive audio manipulation capabilities. I ultimately settled on p5.js, since it was easy to learn and use, could draw both 2D and 3D graphics, and it simplified the process of audio analysis.

The final requirement was sharing capability. To that end, I needed a database. I decided to use Back4App's database service, since it provided an easy to use API that can upload data to the database through client-side code, which was perfect for the limitations of Github Pages.

After deciding on the tools for the job, the next important step was to start brainstorming on how users would interact with my application, what the application would do in response to those interactions, and what the user interface would look like, based on the features I had planned. Users had to be able to view other users' visualizations and create their own. Thus, my application would be divided into two main sections: a "gallery" of some sort, dedicated to viewing other user visualizations, and a "visualization creator" where users would build, preview, and submit their visualizations of their favorite songs. When users clicked "submit" in the visualization creator, their visualization would be converted into a format storable by the database and stored. Then, when they or another user navigated to the gallery, the database

would send back the newly created visualization, along with other previously saved visualizations.

Now I had to think about how exactly this gallery and visualization creator would look like and function. The visualization projects I had assessed previously had visualization creators, but they didn't have the sharing capabilities I was aiming for, so I had to do more research on other projects that did for inspiration on the gallery. Eventually, I came across vertexshaderart.com, which allowed its users to view other users' visualizations in a gallery, and create their own visualizations in a visualization creator. The gallery displayed non-animated preview thumbnails of visualizations in rows (figure 1). When a thumbnail was clicked, the user was taken to another view where the actual visualization was shown (figure 2). I decided that I would model my application's gallery after vertexshaderart's gallery, my reasoning being that the visual appearance of the gallery was simple but effective and easy to implement, considering the time constraints I was working under. I could always come back and change it if I came up with a fancier presentation format in the future; that was the advantage of software.

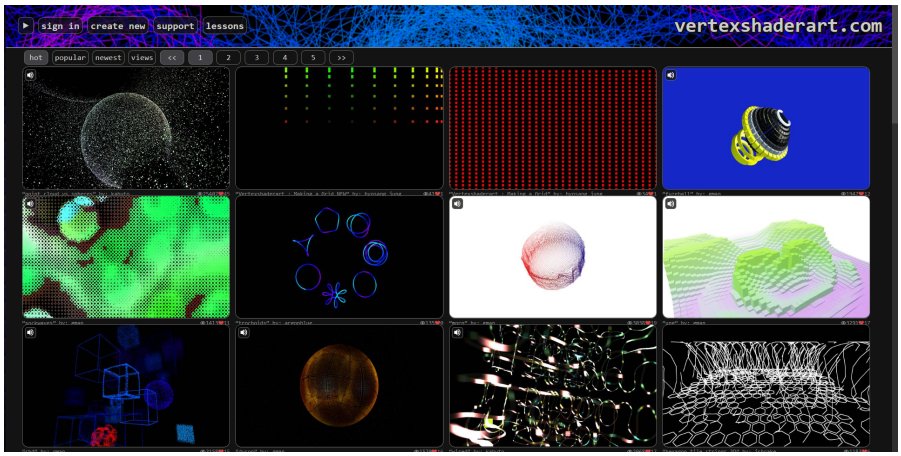

Figure 1: vertexshaderart's gallery

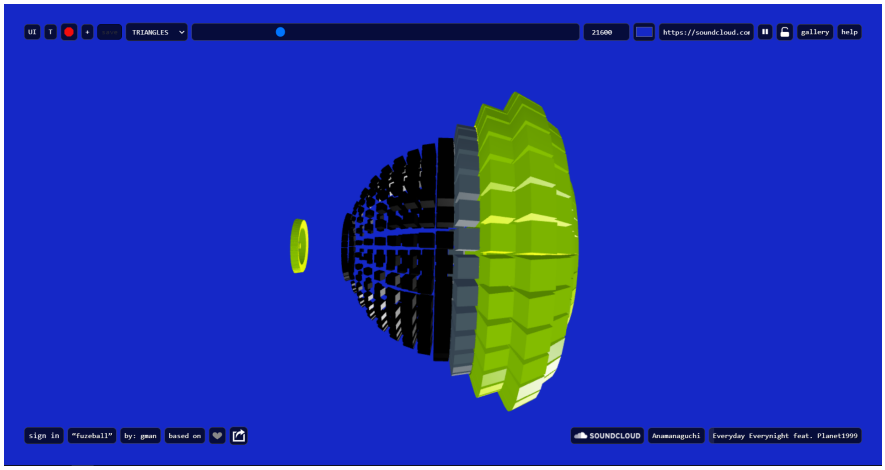

Figure 2: vertexshaderart's visualization view

Deciding how the visualization creator would work was a more complicated matter, since it was an entire visualization system that I would have to design and required more careful consideration. As mentioned before, I wanted my system to give users as much freedom as possible, but it needed to be easy to learn and use, two potentially conflicting goals. Again, to get some inspiration, I looked to existing projects to see how well their visualization systems met this criteria. Vertexshaderart required its users to create their visualizations by coding in GLSL. Code offered a high degree of freedom to the user, and would undoubtedly give them the power to create exactly what they envisioned. Unfortunately, this limited the user base to people who know how to code, or willing to learn to code, in GLSL, which clashed with my goals. The other projects had the opposite problem. Their visualization systems mostly offered one or several preprogrammed visualization styles and offered some minor customization options, such as changing colors or switching between styles. The complexity of the visualizations varied greatly, but they all offered little to non-existent user freedom. They were less of a visualization creator, and more of a visualization viewer. However, they were all usable by a general audience,

primarily because the user had to do less work. Now that I had some ideas of directions I could take my system in, I had to decide where it would fall on the spectrum. On one end, creating visualizations in pure code, and on the other end, visualizations preprogrammed by me. As I moved from one end to another, I would be forced to make the tradeoff between user freedom and general usability. I would lose some of one, and gain some of the other.

There was also a third important variable, code complexity. If users wrote all the code for the visualization, all I had to do was find a way to run this code and make it appear on the screen. On the other hand, if I preprogrammed one or several visualizations, I only needed to write the code for displaying the visualization. However, implementing a visualization system that gave adequate user freedom and easy to learn by anyone would require the most code to write. The code I would have to write for my visualization system would be the simplest on both ends of the spectrum, but get more complicated as it approached the middle. If I let it get too complicated, I would have trouble implementing all the features within the project timeframe. Taking these three variables into account, I decided that my visualization system would go the preset visualization route. However, to increase user freedom and increase the chance that users could create a visualization matching what was in their mind, I would provide a wide variety of preprogrammed visualizations for users to choose from, and offer a limited set of customizations for each visualization. At the time, I decided that this was an acceptable compromise, but I would later overhaul my visualization system, as I will discuss later.

Next, it was time to start designing the visual appearance of the gallery and visualization creator. I started by creating a non-interactive digital wire flow in Figma. In the wire flow, I visualized the main tasks that a user would want to accomplish with my application: namely, viewing a visualization and creating a visualization (figures 3 & 4). I basically created a rough

sketch of every possible view of the application, and indicated where the user should click to navigate from one view to another and thus get closer to accomplishing their task.

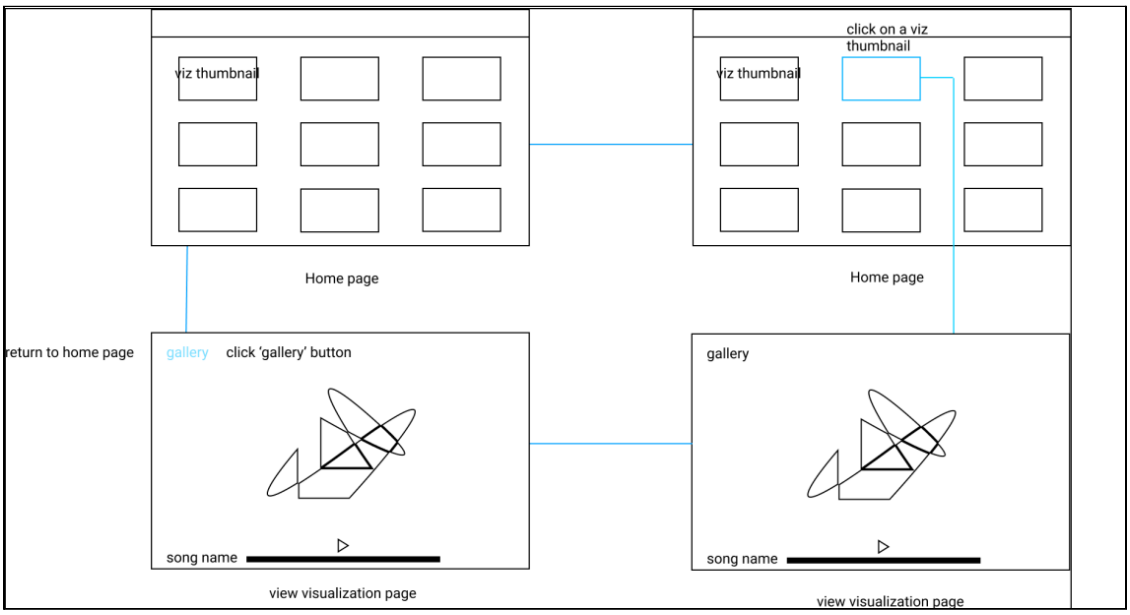

Figure 3: Steps to view a visualization.

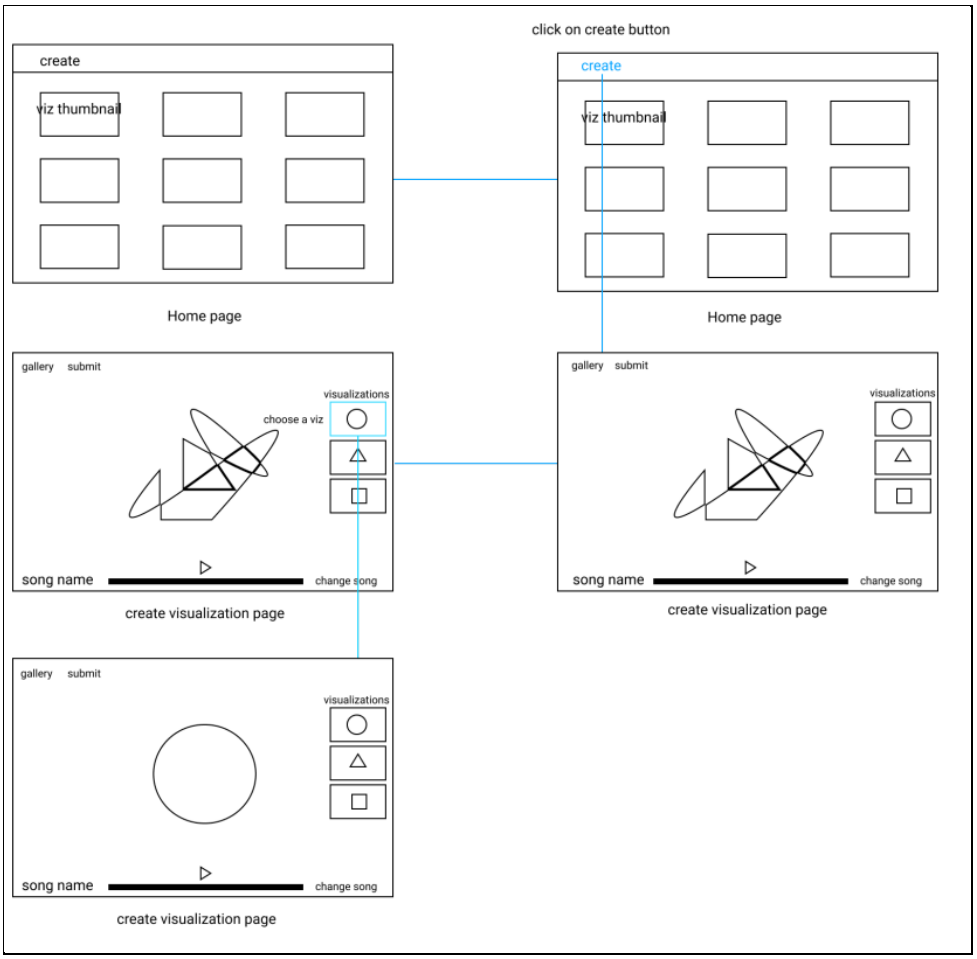

Figure 4: Steps to create a visualization.

After creating the wire flow, I realized there were some potential additions to some views or views that I hadn't included in the wire flow that I wanted to sketch out, such as displaying view counts under each visualization thumbnail, or an overlay for changing visualization settings. A particularly important thing for me to sketch out was the song selector, which was essential to include as a feature of the visualization creator. So I did some quick thumbnail sketches on paper of each of these ideas, with particular attention paid to the song selector (figures 5 & 6).

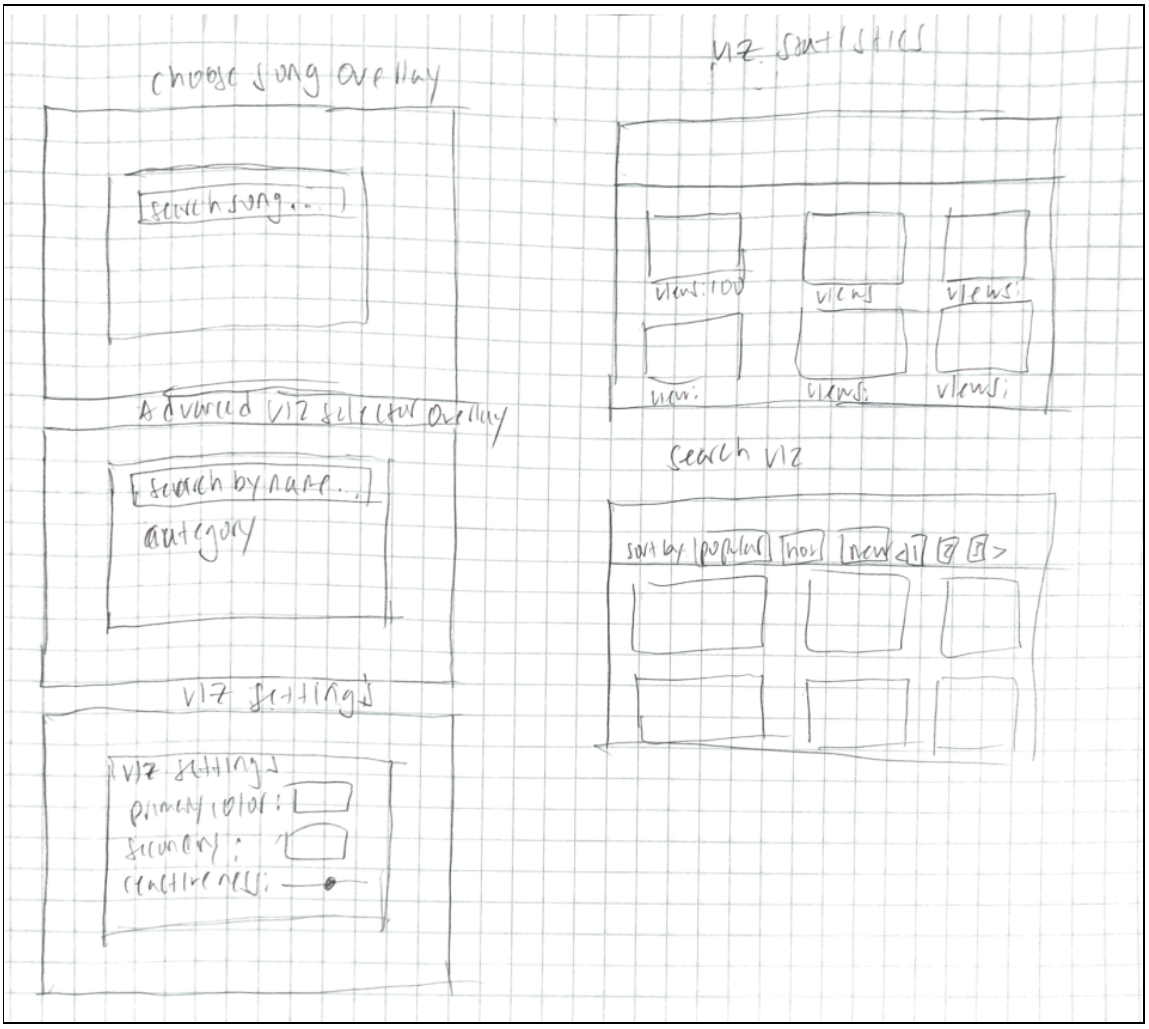

Figure 5: Thumbnail sketches of additional features.

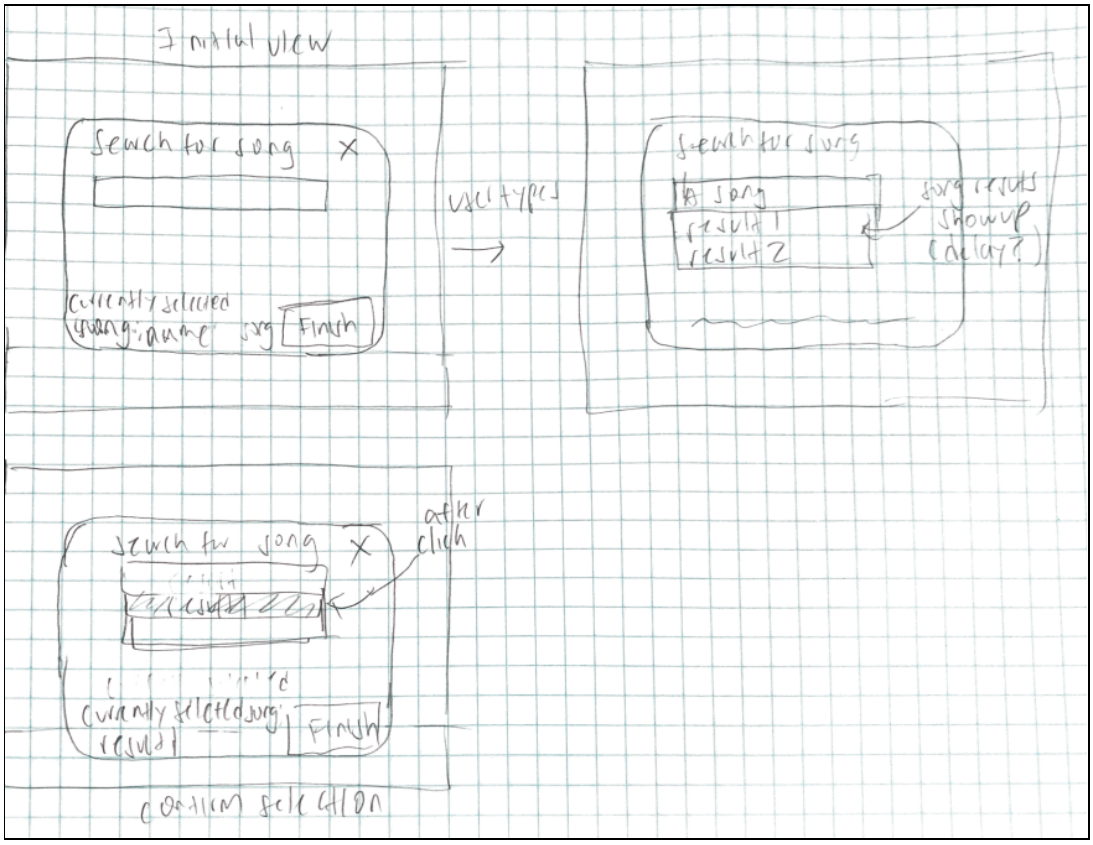

Figure 6: Detailed sketch of the song selector.

These rough sketches weren't detailed enough, so I had to do some more iterations. Next I created the first version of my interactive design composition, which was essentially an interactive prototype of my application. Here, I refined the rough sketches of each view of the application and tried to get it as close as possible to the final product (figures 7 to 9). Additionally, I added transitions between different views when certain elements were clicked, such as navigation bar items and visualization thumbnails (a powerful feature of Figma). The result was a prototype of my application that a user could interact with in a similar way to the real thing.

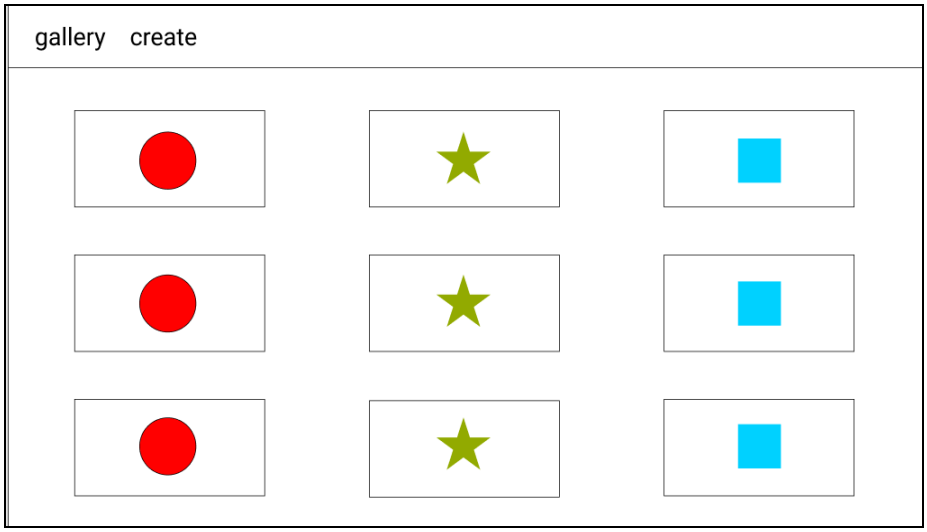

Figure 7: Refined visualization gallery design.

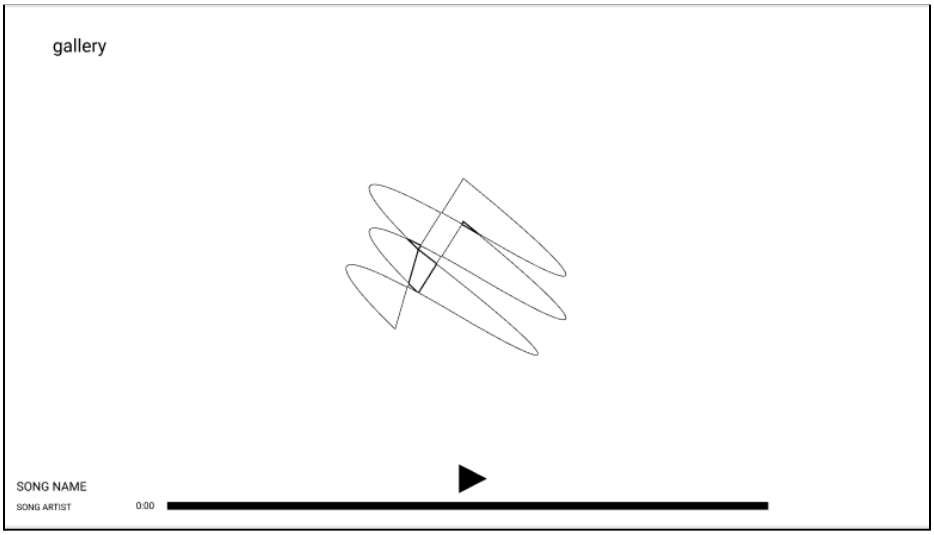

Figure 8: Refined visualization viewer design.

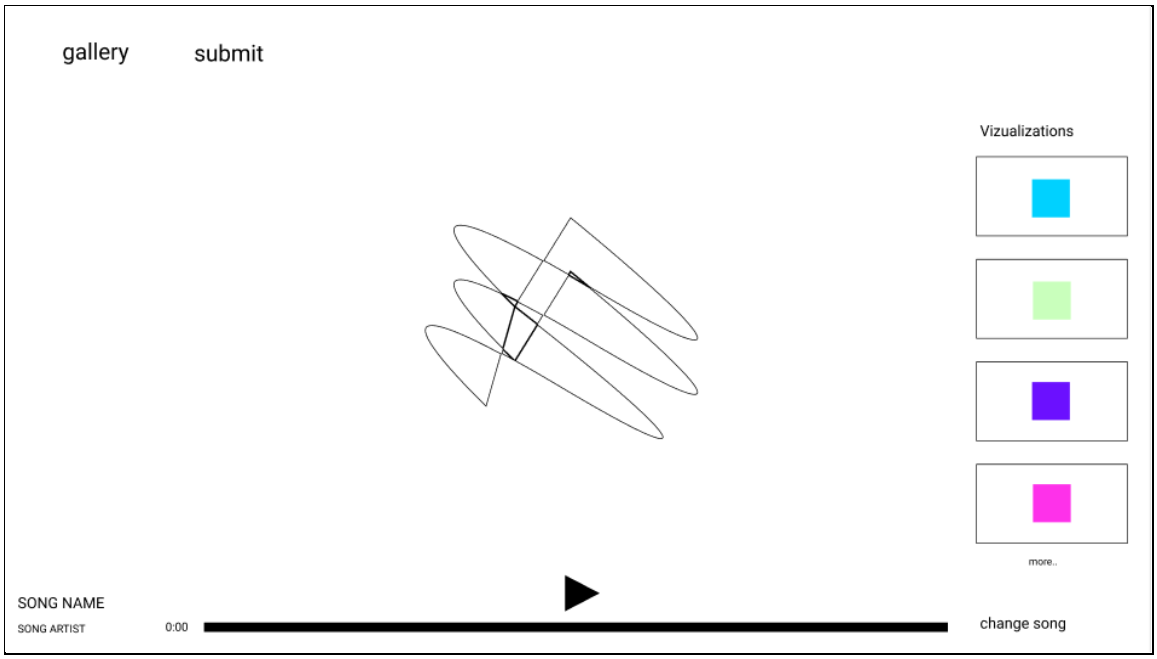

Figure 9: Refined visualization editor design.

Naturally, this was a perfect time to get some user feedback. My user this time was Glenda Drew. She gave me a lot of useful feedback on all views of the application. The navigation bar, an essential part of my application, should stay the same between views, otherwise it might confuse the user, and the phrasing of each navbar item should be similar. Additionally, when a user navigated to the visualization creator, there was no guidance on how to start creating a visualization. She suggested onboarding, which would be a brief tutorial on how to use the visualization creator that would be the first thing shown when a user navigated to the creator. All of this feedback was implemented in the next iteration of the design composition.

At this point, how the gallery would look and function was mostly set in stone, so I decided to start writing code to display the gallery, as well as display a visualization when its thumbnail was clicked. At this point, the visualization and song choice were pre-programmed for testing purposes, since the visualization creator hadn't been implemented yet. Once I

implemented these features, I conducted another round of user testing with my peers. The main criticisms were that it was weird for the width of the navigation bar to not fit the entire screen, and the gallery user-interface wise was a bit barebones. I kept these critiques in mind, but held off from implementing them until later. Next, I added a pseudo-visualization creator. The only features it offered were song searching and selecting functionality and a submit button to upload their selected song to the database. At this point, users could submit visualizations and view them in the gallery (figures 10 to 12).

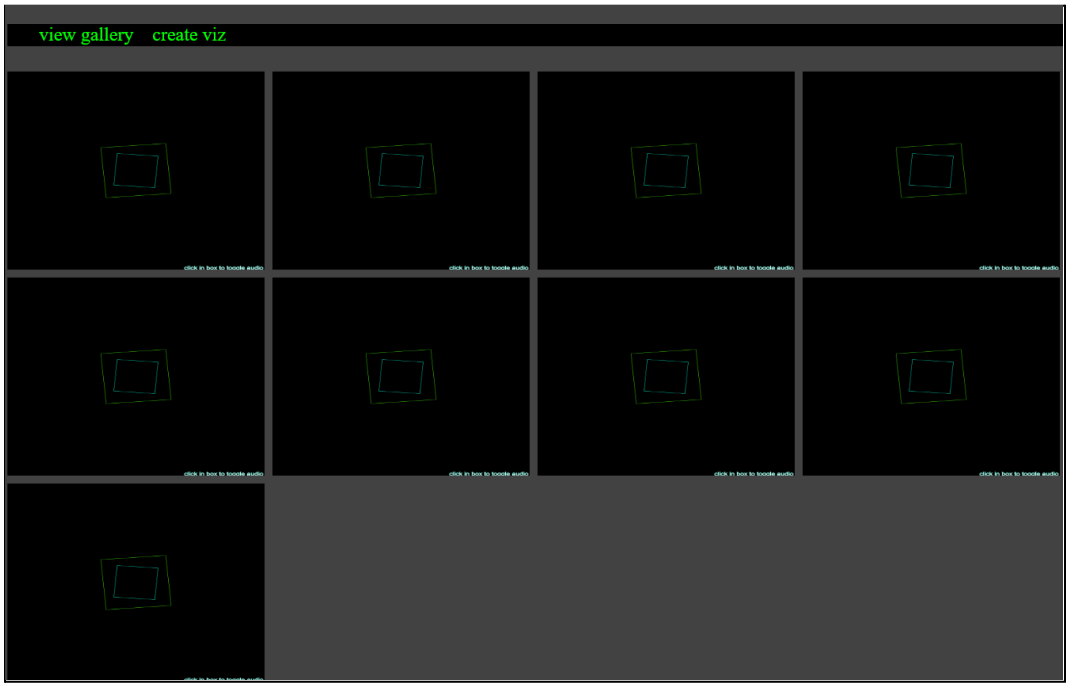

Figure 10: Visualization gallery.

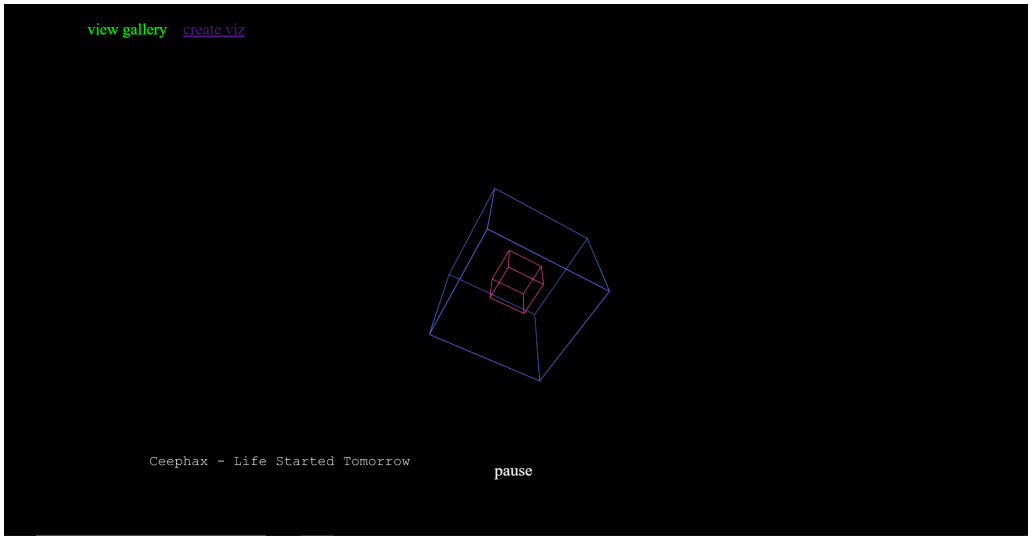

Figure 11: Visualization viewer

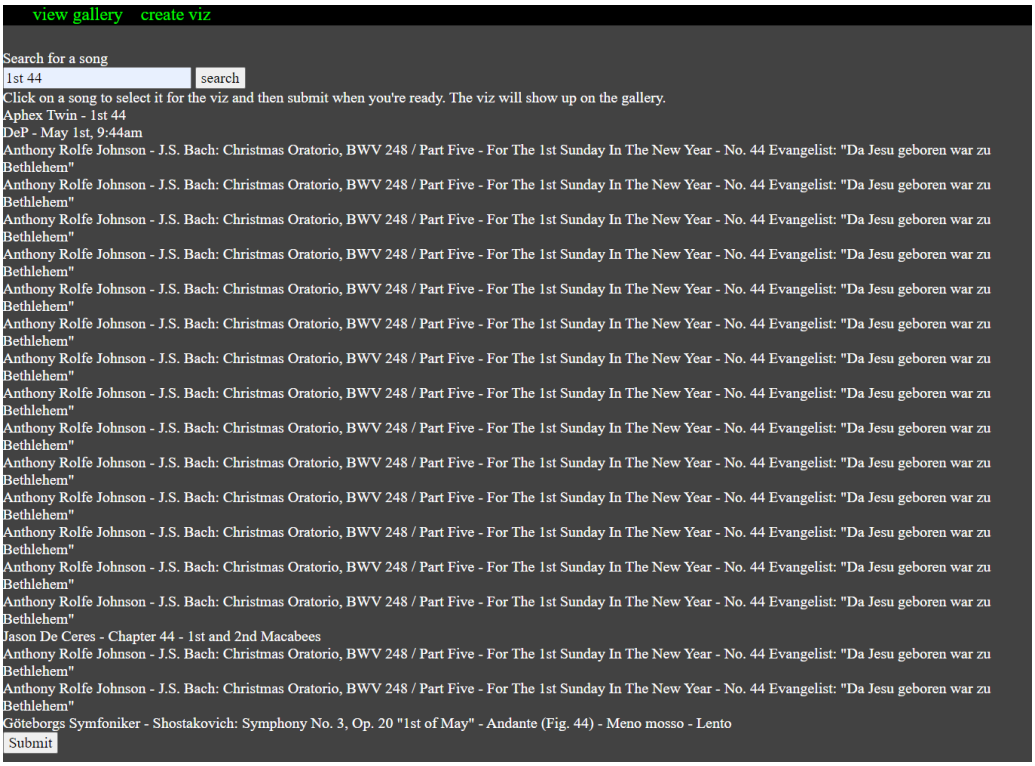

Figure 12: Visualization creator.

The core interactions had been implemented, so although a lot of the finer details hadn't been implemented yet, it was a good idea to conduct another round of user testing. This time, I conducted a more structured usability test. To prepare, I came up with 3 tasks that each user had to complete during the test. These tasks mostly involved testing the core interactions, namely, viewing an existing visualization, and creating a visualization and viewing it in the gallery. The idea was that by having users test the core interactions, any major problems would be caught. During testing, the tasks were displayed to the user as the first thing they saw when they entered the site. Fortunately, testing results gave me many crucial insights. Gallery-wise, one user said they didn't really know how to toggle playback of the visualization and only figured it out by getting lucky. Another user said they would like more diversity in the visualizations (at the time, I only had one type), for example, having more irregular shapes like blobs. The majority of struggles users had concerned the visualization creator. One user said it wasn't obvious to tell which song they selected. Another said they didn't know what to do after pressing submit, since it didn't give any visual cues on where to view their visualization. Also, they didn't know if the submit button actually worked, because there were no visual cues generated by the page. As a result, they inadvertently submitted multiple identical visualizations to the gallery. Yet another user stated that it would be nice if there was a tutorial explaining how the application worked. The general sentiment was that it was hard to figure out how to accomplish things. To resolve this, I planned to add a play/pause button to the visualization view, add onboarding to the visualization creator, and add general visual cues where needed. I initially thought that the feedback on the visualization creator wouldn't be very useful, as its current state was nowhere near the final version. However, I realized that a lot of the feedback was still applicable to the final version, as most of the core features, like submitting and choosing songs had been

implemented. Nonetheless, I knew it was essential to do one more round of user testing once I had gotten close to fully fleshing out the visualization creator.

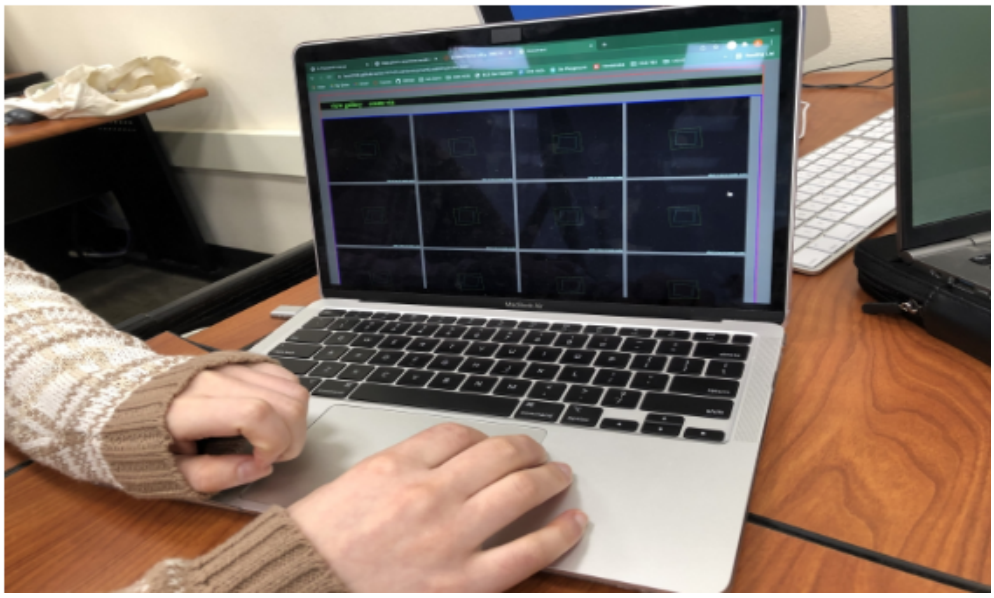

Figure 13: User testing in action.

Now, it was finally time to start implementing the true version of the visualization creator. Before coding, I did some research on control panel design. A control panel would be the place where a user would customize almost every aspect of their visualization. Therefore, it was essential to design an intuitive control panel. Most of my research focused on graphics programs, such as Figma and Adobe Illustrator, because I could see the parallels between them and my visualization creator. For example, some of the desired features of my creator's control panel, such as changing background color and position of selected shapes, were also present in graphics programs (figure 14). So I knew that my control panel design could take lots of inspiration from these programs.

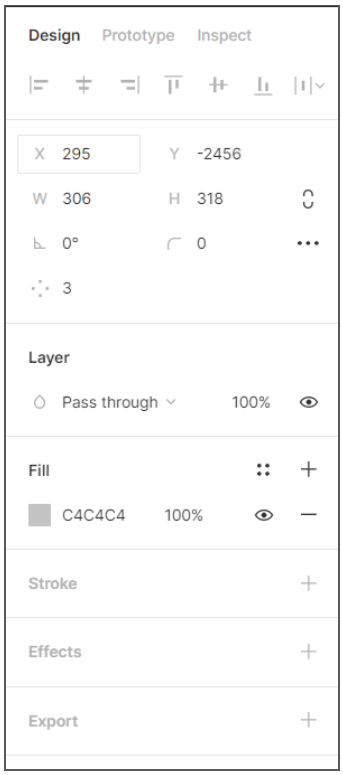

Figure 14: Figma's control panel layout.

Besides graphics programs, I took a look at control panels in existing audio visualization creators. SongRender particularly caught my attention (figure 15). Its audio visualization system offered a few core functionalities. Graphics wise, 4 primitive objects could be added to a canvas: text, image, audio waveform, and song progress bar. Once on the canvas, these objects could be easily resized and moved around by clicking and dragging with the mouse. For more fine grained control, when an object was selected, the control panel would update to show the settings for that object and the user would be able to input their desired values in the control panel. Audio visualization wise, the user could load in audio files and play them. When playing audio, any waveform objects on the canvas would visually change to match the audio's waveform in real

time. Ultimately, SongRender's interface and functionality was very similar to that of Figma's and other graphics programs. The main difference was audio visualization.

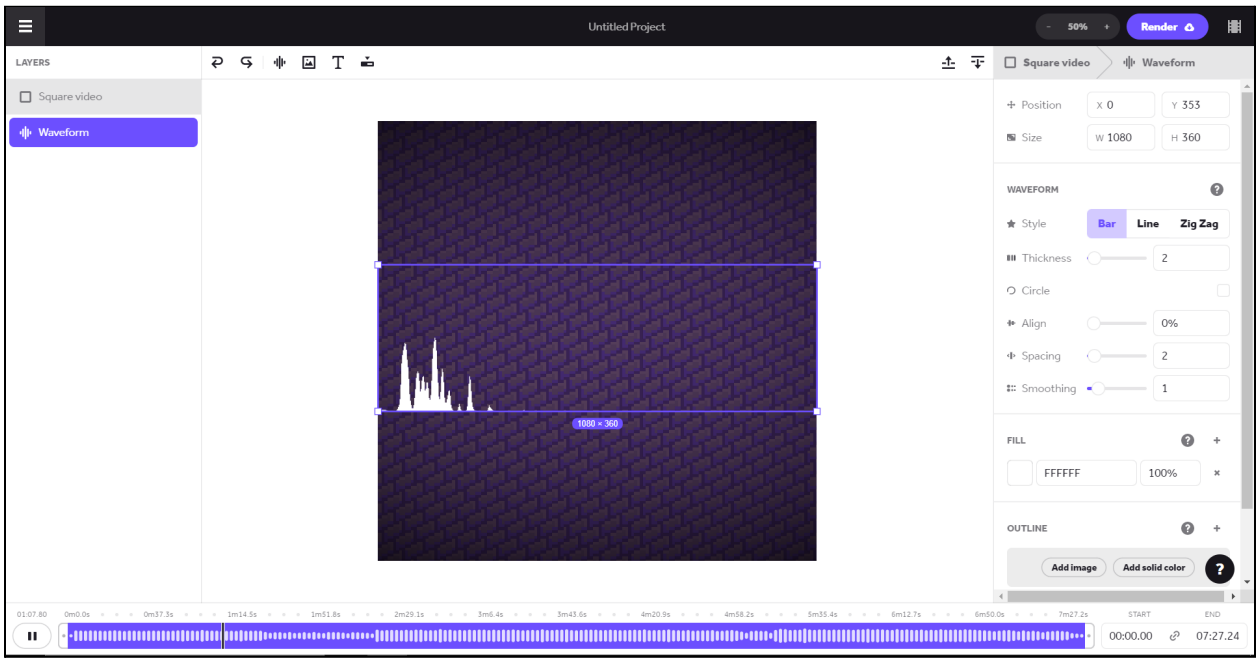

Figure 15: SongRender's visualization creator.

At this point, I realized that SongRender's visualization system could be a better alternative to the audio visualization system that I had decided on previously. As a refresher, the current planned system was to offer a variety of diverse, but preprogrammed visualizations with limited customization options to the user. I compared the two systems based on the three metrics: ease of use, user freedom, and code complexity. SongRender's system would be harder to learn than the current system, since there would be more functionalities to keep track of. However, the vast increase in user freedom would more than make up for the slight learning curve. Preprogrammed visualizations restricted user freedom by a great deal. Even if there were some customization options available, there would always be some fundamental parts of the

visualization unable to be altered. Of course, I could try to change this by giving the user full control over the visualization. But at that point, the system was no longer a preprogrammed visualizer and would essentially be closer to that of SongRender's system, which did let the user control everything about how their visualization is presented. The codebase involved in implementing SongRender's system would definitely be more complicated than a preprogrammed visualization system. For example, I needed to implement a system to drag and resize objects with a mouse, and I would have to make the control panel update to display an object's settings whenever that object was clicked, which were all things that weren't necessary with preprogrammed visualizations. However, I was confident I could implement these features with enough time. At the end of the day, user freedom was the most important goal of my project, and SongRender's visualization system provided a much better balance of user freedom and usability than a preprogrammed visualization system. Thus, I decided that the final version of the visualization creator would be modeled after SongRender's creator.

Now, I still needed to decide what specific functionalities my new visualization creator would have, how the interface would look like, and how the user would interact with the interface to accomplish their goals. First, the interface was inspired by both Figma's and SongRender's interfaces. There would be 5 distinct "views" of the interface: the object toolbar, the canvas, the context menu (control panel), the song selection overlay, and the audio playback controls. The object toolbar was simple; it contained an icon of a rectangle and an icon of a circle. When an icon was clicked, the respective object would be created and show up on the canvas. The canvas displayed all the objects the user had created; it functioned both as a preview of what the final visualization would look like when submitted, and as a way of manipulating the

positions of objects in the visualization. The user could change the position of an object on the canvas by dragging it with the mouse.

The context menu provided a way to change every setting that was changeable in the creator. It had two top-level states, “design” and “audio” (figures 16 & 17) . The context menu displayed different settings depending on what state it was in. Naturally, when it was in design mode, the context menu only displayed settings related to visuals, such as canvas background color, object position and dimensions, and object stroke width. Conversely, when it was in audio mode, the context menu only displayed settings related to audio, such as song selection. Furthermore, each context menu mode view was divided into two logical sections: the project section and the object section. Naturally, the object section contained settings of an individual object. The project section contained more general settings of the visualization that didn’t fit into the object section, such as canvas background color or song selection. Again, these sections would display different settings based on the mode the context menu was in. As an example, if the context menu was in design mode, the project section would display a canvas background color picker, and if it was in audio mode, the project section would display the “change song” button. This context menu design took more inspiration from Figma’s context menu than it did SongRender’s. The latter made no distinction between design settings and audio settings, and displayed all the settings at once, which took up more space and cluttered the interface. Figma’s design was cleaner and more organized, and put less mental load on the user.

The song selection overlay was where the user could search for a song. It appeared as an overlay when the “change song” button in the context menu was clicked. The user could see the currently selected song and type in a search bar to find a song. Once they pressed enter, a list of songs appeared and they could click on a song to load the new song. Finally, the audio player

controls provided a way to start/stop audio and change the current position of the song. When audio began playing, any objects on the canvas that were set to react to audio would react accordingly.

To allow the user to accomplish certain goals, most of the views had to interact with each other to some degree, both visually and non-visually. For example, what if a user wanted to choose a song, change the canvas background color to red, and create an ellipse with position (500, 600)? To choose a song, the user would have to change the context menu mode to “audio” (if they weren’t on “audio” already), and click the “change song” button in the Project settings. The song selection overlay would have to be notified of this action and display itself to the user. Then once the user selected a song, the audio player had to be notified to load the selected song. Next, to change the background color of the canvas, the user needed to change the context menu mode to “design”, and pick the color. When the color changed, the canvas had to be notified to update its color, otherwise the canvas and the context menu would be out of sync. Next, to create an ellipse, the user would click the ellipse icon in the toolbar, and the canvas would have to be notified to display the ellipse at some default position. Then, if the default position of the object wasn’t at (500, 600), the user would have to move the object. The user could try to drag the object, but dragging was imprecise and required a lot of careful mouse movement to get to exactly (500, 600). So the user would probably change the position via the context menu. When the user inputted a new coordinate, the canvas would have to be notified to display the object at the new location. However, what if the user did it the hard way and dragged the object? Then the context menu would have to be notified to show the object’s new coordinates as the object was dragged on the canvas. I needed to figure out how to implement all of these interactions between views in my code.

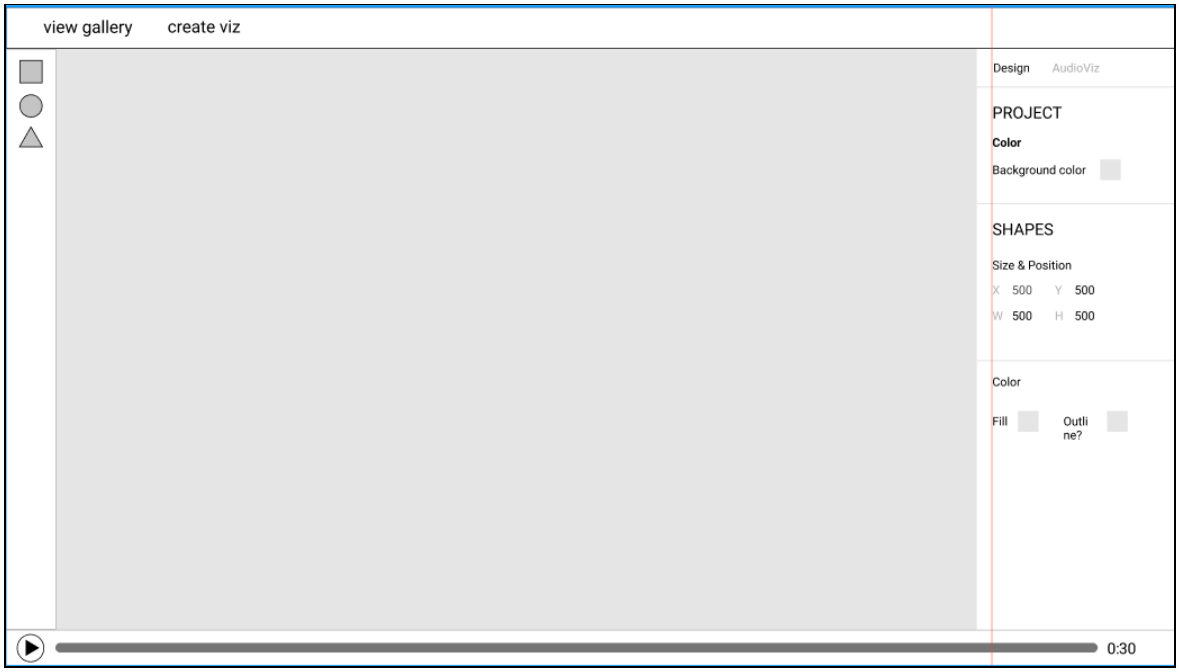

Figure 16: Visualization creator in “design” mode.

Figure 17: Visualization creator in “audio” mode.

Once I had designed the visualization creator's interface and refined it with Glenda's advice, it was finally time to begin implementing it in code. After writing some code, I realized that I should think carefully about how to design my code. As a codebase grew larger and larger, it would become more and more important to choose a good architecture for the code. Otherwise, it would get difficult to understand and maintain if the code wasn't structured well. In my opinion, the code I wrote for the gallery wasn't very well designed, and when I looked at it again, I could see room for improvement. However, since it was relatively small in terms of lines of code, it was still understandable. Thus, the code was acceptable for the time being, but I definitely planned on restructuring it in the future. The visualization creator was a different story. I knew the codebase would be substantially greater than that of the gallery, because the creator had much more complex features that had to be implemented. If I didn't choose a good design pattern at the start, my code would run a very high risk of becoming hard to read and maintain. And, it would be very difficult and time consuming to switch to a good design pattern once a bad codebase had grown really large, because in the worst-case, it would be equivalent to building the entire thing again from scratch. I also knew that my codebase could have the potential to get really large, as I planned on adding new features after the project duration. So, I needed to choose a design pattern that kept my code readable and easy to maintain and add new features.

One ubiquitous design pattern for developing graphical user interfaces (GUIs) was the Model-View-Controller (MVC) design pattern. As my web application was definitely a GUI, this design pattern was a possible candidate. Traditionally, MVC split the application logic into three separate objects: a model, a view, and a controller, each handling a different responsibility. The model stored pure data needed by other parts of the application. The view was a visual representation of the model, responsible for knowing how to render any visual interface element

to the screen and display it to the user. The controller received user input and if needed, told the model to update its data or told the view to re-render. Of course, there were many different interpretations of MVC that diverged from the traditional description to fit different use cases. Indeed, I felt that the traditional specifications didn't match my use case, and that it was necessary to modify the traditional pattern for the requirements of my application. For example, the traditional MVC pattern had the controller take user input. However, in JavaScript, user input was captured by attaching event listeners to HTML elements when they were rendered, which was more of a responsibility of the view. So it made more sense for the View to take user input.

The MVC variant I used in the visualization creator had one model and one controller. Instead of one view, there were five views: the object toolbar view, the canvas view, the context menu view (control panel), the song selection overlay view, and the audio player view. The model stored data that is needed by other parts of the application, but it had no ability to modify the data itself. The views had similar responsibilities to the views in the traditional MVC: they only knew how to render visual elements, and whenever it needed to conditionally render something based on some state of the application, it needed to query the model for this information through the controller. However, the views accepted user input and passed this input to the controller. The controller was responsible for modifying the data in the model and telling the views to render visuals. The view, model, and controller all interacted with each other to help the user accomplish a task. When the user wanted to switch the context menu mode to "Design", they would click "Design" at the top of the context menu. The context menu view would capture this input, and tell the controller that the user wants to change the mode. The controller would then compare the currentContextMenuMode, a variable held in the model that kept track of the current state of the context menu, with the new desired mode given by the view. If the new mode

was different then the current mode, then the controller would update the mode stored in the model, and tell the context menu view to re-render. Else, nothing would happen, because the context menu mode was already in the mode the user wanted. If the context menu view did render, it first would get the current mode of the context menu by telling the controller to get the data from the model and pass it to the view. Then, it would render the audio or design settings based on this mode. After I decided on the design pattern, I started implementing, and eventually finished, the visualization creator.

Project Evaluation

Once I finished the visualization creator, it was time to do one last round of user testing. Unlike the usability testing I had conducted previously, the prompt this time was more open-ended. The goal was to simulate how a user would use the website in real life: without any explicit directions on what to do or where to go. This would be the best method of gauging how well the website was designed. After the user was done testing, I would ask them a variety of questions to get their opinions on the user experience and see how well the goals of the project were met. Most of my users said that the purpose of the application was to let people express the music in a visual form, which was more or less my original intention. Their responses when I asked them who they thought the target audience was were more interesting. They mostly said that the target audience would be people that were involved or interested in music more than the average person. Musicians or DJs could use the application for visuals for music videos or when DJing. One user said that electronic music fans in particular were the target audience. This feedback told me that my application was more niche than I realized. This was potentially supported by the fact that the best feedback I received was from a user who was much more of a music fan than the others. During testing, it was obvious that they were more engaged with the visualization creator, and put more effort into designing a visualization than other users. This told me that the average person probably wouldn't be interested in this sort of application. I also asked users what the strengths and weaknesses of the application were. Users said the application in general was very straightforward and easy to use and navigate. It was easy and quick to jump in and start creating something without any of the hassle that one might find when using other websites. For the visualization creator, a user said that it was very simple and easy to use, but had lots of depth. One could "go crazy" and create very complex visuals if they were dedicated

enough. I was happy to hear this as one of my fundamental goals was to have the visualization creator be simple, yet powerful. The visualization creator had the greatest strengths, but it also had some of the greatest weaknesses. Some obvious problems that users pointed out were the lack of essential operations that should be in digital graphics creation applications, such as deleting, copy and pasting, layers, and group selection. The lack of these features meant that it would be hard to perform certain operations. For example, what if a user wanted a shape to be behind another shape, or accidentally created something and wanted to get rid of it, or wanted to move a bunch of objects to another location? It would be very hard to do all these things in the current version of the creator. Besides that, users wished they had the ability to go back and edit their creations after they had exited the page; essentially, they wanted user accounts. Finally, I informed the users about the actual purpose of the application, which was to give them the power to visualize anything in their minds when listening to a song, whether it was just a simple color or a forest. Almost everyone agreed that creating abstract visualizations would be simple, as basic shapes could already be created. However, it would get very difficult to create more complex things; for example, if somebody got, in the words of one of my users, "gas station vibes", it would be very difficult to draw a gas station in the current version of the visualization creator. A free draw tool could help greatly with this.

From this final user testing, I learned that out of the three fundamental goals, accessibility, user freedom, and sharing capability, my application in its current state achieved accessibility and sharing capability, and partially achieved user freedom. Users definitely found the application easy to use, regardless of programming experience. Users also could submit a visualization to the gallery and it could be viewable by anyone. Users did have a lot of freedom in creating visualizations, but there were some visuals that simply weren't achievable yet, which

meant that users did not have the ability to visualize anything they wanted. Thus, the goal of user freedom was still a work in progress, and would likely always be a work in progress, as I stated earlier. But I will continue to work towards achieving this goal in the future, and thankfully, my users have given me a lot of ideas on how to continue this work.

Project Results & Discussion

Limitations

There are numerous limitations of *Visaudio* that conflicted with my stated goals, some more controllable than others. One major limitation was the restrictiveness of the music API I chose. The Napster API only provides 30 seconds of audio for each song, which means that visualizations are limited to only 30 seconds. Additionally, the user cannot choose which 30 seconds to use. From testing, it seems that the 30 seconds are usually taken from the beginning of each song. This means that if a user wanted to visualize a certain section of the song, it would be unlikely that the 30 second clip provided would match. This led to some disappointment when I tested out visualizations of different songs. As I found out, visualizations were especially boring for songs that don't really have anything interesting going on for the first minute or so. It was frustrating knowing that a song got really good halfway through, but the visualization only showed the uninteresting parts. Even more frustrating was a song cutting off just as it got to the good part, leaving the user with a feeling of incompleteness. Although, there actually might be some good that comes out of that, as it may encourage people to listen to the song in full in their favorite music streaming platform, which supports the artist, but I digress. Ideally, users should be free to visualize any part of the song they want, whether it is just 30 seconds or the entire song. Ultimately, the limitations of the API interfere with my goal of user freedom.

Additionally, the audiovisual options are limited. Currently, users can only choose between two primitive objects: the rectangle and the ellipse. This means any visualization is limited to featuring these two shapes, which reduces diversity and individuality of the visualizations. Also, users do not have much freedom in deciding how objects react to

visualizations. Users can only choose between bass and treble frequencies. The only attribute of objects that changes when audio plays is the dimensions.

Learning Outcomes

I've learned something new with every programming project I've done, and this one was certainly no exception. Technology-wise, I learned a new JavaScript library, p5.js, and improved my Figma skills. I also learned how to set up a CORS proxy server on Heroku. Learning new technologies is always beneficial, but I learned something much more useful, something that makes this project stand out from the previous web development projects I have worked on. I was able to get more familiar with essential, industry-standard steps in the development process. Or in other words, how to properly develop a web app. In all but one of my previous projects, once I had an idea, I jumped straight into the coding step, not even considering sketching out the interface or doing anything involving planning out the look of the webpage beforehand. I never seriously considered the look of my site before I started working on it. The result of this was that I was essentially making up the interface as I went. And I never did any user testing, so who knows how usable my apps actually were by real people? Fortunately, for *Visaudio*, Glenda pushed me to seriously brainstorm and sketch out all the possible views and user interactions of my app before coding it up. I spent as much time researching and designing the UI as I spent programming. Not only did I improve my visual design skills, I also improved my software engineering skills. This was the first major project where I seriously considered how my code should be architectured. It was the first time I was able to apply the design pattern knowledge from my software engineering class to a personal project. This project, I really felt like I truly *designed* something. Even if I'm not going to be a web designer in the future, this project has made me feel more confident in my ability to communicate with web designers, which is an important skill for web developers, even if their responsibility is more implementation than design.

Future Work

Because my project is implemented in software, it is easy to do future work on it without having to redo it from scratch. I would work on addressing the limitations mentioned earlier. First, I would choose a different method of streaming music. The current version of the project uses the Napster API despite its limitations because of my prior experience with it and its ease of use, and because I couldn't find any suitable alternatives. I am confident that I can find a better alternative with more research.

Additionally, I plan to look into refactoring my code to make it more readable, efficient, and maintainable. One way I could accomplish this is to do a complete overhaul of the codebase and rebuild it from scratch with a JavaScript framework, such as React or Vue. Currently, the core of the application is powered mainly by vanilla JavaScript. One drawback of vanilla JavaScript I am aware of (and that I noticed while programming) is that performing some operations can get tedious and can take lots of lines of code to accomplish. For example, updating parts of the page on user input requires me to explicitly write code telling the browser to change those parts. A larger codebase is harder to maintain. Additionally, some parts of my code are not efficient. For example, in the visualization creator, when the user clicks on an object on the canvas, the context menu on the sidebar updates to show the properties for that object, which naturally involves executing some code to update the context menu. However, a user may click the same object multiple times in a row. In that case, it is unnecessary to execute any code to update the context menu, as the object selected hasn't changed. However, my code does perform this unnecessary update, which makes it inefficient. To solve this, I would have to write some logic to check if the new selected object is the same as the previously selected object and decide whether to update the menu. Ideally, I wouldn't have to explicitly handle this.

Frameworks could alleviate some of these issues. Frameworks like React and Vue automatically handle updating of the page, which means I don't have to explicitly write the code to do it. Additionally, they perform updating in a smart way; when updating a section of the page, the framework will compare the new version with the old version, and update only the parts that have changed, making it very efficient. In essence, these libraries and frameworks do a lot of the work for the developer, which saves time and increases the speed of development.

I also plan to add additional features. One important feature would be user accounts. Currently, a user can submit a visualization to the gallery, but they cannot edit this visualization after submission. They also cannot save an in-progress visualization and come back to it later. User accounts would solve all these issues. Once users have logged in, they would be able to view, edit, publish, and delete the visualizations they have created. I also didn't get to implement the complete version of the audio player. Currently, a user can only play and pause the audio. Ideally, I want users to be able to not only play and pause, but also seek to a certain position in the song. I could also look into making the audio fade out as it reaches the end of playback. Users could also be able to load in an audio from their local filesystem if the music API doesn't have the song they want. In this case, they wouldn't be able to submit to the gallery, since the entire audio file would have to be uploaded, which takes a lot of space, and the database probably wouldn't be able to handle it. Also, it would cause copyright issues.

I would also like to make more of an object's attributes react to the audio. Currently, only the dimensions of the object react to the audio. I could also make attributes such as position, color, and stroke weight react to the audio.

Bibliography

Berg, Jordan. *Butterchurnviz*. https://butterchurnviz.com/. Accessed 19 Jan 2022.

Greggman. *vertexshaderart*. https://www.vertexshaderart.com/. Accessed 13 Feb 2022.

Hernandez, D. Rafael. "The Model View Controller Pattern - MVC Architecture and Frameworks Explained". *freeCodeCamp*, 19 Apr 2021, https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/.

McCarthy, Lauren. "Global and instance mode". *Github,* 4 Jul. 2020, https://github.com/processing/p5.js/wiki/Global-and-instance-mode. Accessed 16 Apr 2022.

McNerney, Alex. *raVe.* https://rave.ajm13.com/visualizer. Accessed 19 Jan 2022.

Preziotte, Mathew. *Partymode*. https://preziotte.com/partymode/#. Accessed 19 Jan 2022.

Shiffman, Daniel. "draggable class". *p5.org*, https://editor.p5js.org/codingtrain/sketches/U0R5B6Z88. Accessed 7 Mar 2022.

The Figma Team. *Figma*. figma.com. Accessed 5 May 2022.

The SongRender Team. *SongRender*. https://songrender.com/. Accessed 5 May 2022.

Yannakopoulos, Yannis. "Creative Audio Visualizers". *Codrops*, 5 Mar. 2018, https://tympanus.net/codrops/2018/03/06/creative-audio-visualizers/. Accessed 19 Feb 2022.

Z. Yingfang, P. Yi, Z. Junren. “Study on Application of Audio Visualization in New Media Art.” IOP Conf. Series: Journal of Physics: Conf. Series 1098. 2018. doi :10.1088/1742-6596/1098/1/012003.