Synesthesia

Authors: Abhishek Agarwal, Parth Maheshwari, Rachana Murali Narayanan
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—This project aimed to create a system capable of producing immersive light shows in response to audio inputs. Given the time and resource constraints of the course, we built a system to control four types of lights - par, laser, derby, and strobe - both in unison and independently while processing audio samples and mapping them to dynamic lighting values. The goal was to develop scalable audio processing algorithms and customization options for lighting shows, enabling small performers to reduce costs and setup time. We were able to deliver a pilot version of the system, showcasing the required functionality. The project has been completed successfully, and the system has been tested and demonstrated to work as intended.**

***Index Terms*—Audio Processing, Change Points, Derby, Expressive Lighting Engine (ELE), Feature Extraction, Laser, Lighting Logic, Lighting Queue, Light Set, Par, Performance, Real-time, Recommendation System, Show, Strobe, User Interface (UI)**

1 INTRODUCTION

Stage lighting plays a crucial role in engaging the audience during musical performances. Performers use stage lighting that is controlled using sophisticated audio interfaces, which often comes with a large time, money, and resource expense to the performers. [1] These interfaces have audio active modes that use basic decompositions such as volume thresholds or require extensive manual programming and lighting engineers to activate and control the lights during a performance [2]. While software such as QLC+ allows some stage setting and lighting customizability [3], performers are expected to spend a significant amount of time programming light behavior ahead of the performance, increasing their setup time.

To address these challenges, our project developed a dynamic lighting system that analyzes audio inputs to control lights aligned with the music. By decomposing audio into key components, identifying features, and synchronizing lights with music from multiple genres, our system enables performers to automate the lighting process and customize their shows without the need for extensive manual programming or costly equipment.

2 USE-CASE REQUIREMENTS

The aim of the system is to control one set of light pairs that included Pars, Derbies, Lasers, and Strobes using features extracted from the audio and user's manual inputs. For the scope of this project, the use-case requirements can, therefore, be divided into three distinct categories: latency and setup time, audio processing and feature extraction, and manual adjustments.

2.1 Latency and Setup Time

For any light show to be fun and engaging, it is essential that it is reflective of the audio being played by the performer. To achieve this, our system must be able to play the input audio and communicate with the lights in almost real time to achieve synchrony. According to previous analyses [4], if the lighting is within 185 ms of the audio on average, the two are indistinguishable to the human brain, so that is the chosen amount of permissible latency for this system. Further, user testing and anecdotal evidence reveal that on average it takes a performer about an hour to set up the equipment for a performance. Hence, to ensure that the system can be seamlessly integrated into traditional workflows with minimal effort, this system strives to be less than 5 minutes to the overall setup time. Finally, without any significant cost in latency, this system should be able to control different lights both independently and in unison, throughout different points of the song.

2.2 Audio Processing and Feature Extraction

The decomposition of the input audio determines the behavior of the lights at any given point of the chosen audio. Hence, it is important that relevant features of the audio are extracted. The signal processing subsystem should be able to detect and extract more than 90% of the auditory changes as compared to a hand-labeled waveform. This ensures that there is enough data for the lights to reflect all the major and minor inflections in a song. Additionally, the quality of audio processing can be broken down into two major components: the diversity and accuracy of the features extracted over the duration of the song and the associated latency during the extraction of these features. Modern audio processing tools can extract multiple auditory features, such as frequencies, amplitudes, energies, beat onsets, and vocal detection. Moreover, they can extract up to 44,100 samples per second, creating data with an average latency of 20 ms per second of processing on modern devices [5]. Therefore, it is reasonable to expect our system to extract these auditory features for a given song and process them with low enough latency to create lighting decisions synchronized with the music.

2.3 Manual Adjustments

While automating the light shows can save a significant amount of time for performers, it is crucial for the system to provide some customizability so performers may choose to make adjustments to the lights to fit their style and requirements. To accomplish this, the system must allow users to override automated execution made available by our system during playback. Furthermore, the system must also allow the user to pick their next set of tracks from a list of recommended songs generated using meta characteristics of the current audio track, as described in Table 2, or choose to record, search or upload a song they desire. While users can use these overrides as frequently as they want, we want the system to be automated, and it should not require any manual inputs from the user other than the music in order to function. Therefore, the system should not require, on average, more than 3 manual overrides per minute, as tested across different users. Given that it is difficult to quantify how many times a user will actually bypass the automation, this is an important qualitative metric for the project that will be incorporated into user testing.

3 ARCHITECTURE AND PRINCIPLE OF OPERATION

For the overall architecture of the project, we decided to divide our systems into five main subsystems, as seen in Figure 1. This was done to facilitate the division of tasks and allow for unit testing of smaller parts of the project. Together these components will cover all the functionality that we require of the entire system.

3.1 Main and Hardware

The *Main* component functions as the central shared system that interacts with the other subsystems and schedules and allocates tasks that need to be completed before the song is played. It contains information regarding the audio source and fetches the audio into a file and plays it from there. It also stores the current musical parameters for each of the other subsystems to view. Additionally, it contains information about the current Light Set, a software representation of the hardware available to us. This subsystem is a modification of the *Show* subsystem we proposed in the design, but it simplifies data management and is able to communicate more efficiently with the different subsystems. For this project, we also used a light rig called the GigBar2 connected using a Digital Multiplex (DMX) protocol as seen in Figure 12, located in the appendix.

3.2 Feature Query

The *Feature Query* subsystem allows us to identify a song and extract the musical parameters of the song. It performs the song detection using the Shazam API and then performs a song name query using the Spotify API. From that, we extract a few global musical parameters such as danceability, valence, energy, tempo, loudness, and liveness. We use these parameters to set a default mood for the light show and generate recommendations for similar songs that users may want to play next.

3.3 Web Application and User Interface

The *Web Application* is the only subsystem visible to the user to interact with the project. The User Interface (UI) is designed to be simple and intuitive to allow the user to produce seamless light shows without needing to pre-program any behavior. The user can search for songs that they wish to play or select from an expansive list of recommended or previously played songs for even quicker processing. Once a song is playing, the user can also manually override the automated light show at any instant with an effect of their choice. This subsystem differs in functionality from the subsystem we had initially designed, as rather than making users change the overall look of the show, we were able to provide them with more granular control over the commands that are executed on the lights.

3.4 Signal Processing

The *Signal Processing* subsystem is responsible for fetching the audio file for the audio that the user requests and processing it to extract the relevant features. The subsystem concurrently processes multiple audio files and generates outputs for each individual file. It decomposes the audio to first extract the times a beat falls or beat time stamps, and then extracts other features, such as maximum frequencies, and energy values for each individual beat. These features are logged and relayed to the lighting logic for further processing.

3.5 Lighting Engine

The *Lighting Engine* subsystem is divided into two key components: lighting logic and execution queue. The lighting logic takes in the logs produced by the Signal Processing subsystem and processes them for the duration of the song. Logic interprets the values of the various signal attributes in the context of the song and uses it to find key change points throughout the music, such as instrument change, choruses, or voiced and unvoiced areas. The logic then uses these change points to characterize how the light should behave at any point during the playback of the song. These decisions are then passed on to the lighting queue, which ensures an appropriate functionality is displayed on the lights at a given time during playback. It also allows for the UI subsystem to overwrite the value at any point during the execution with the behavior decided by the user.

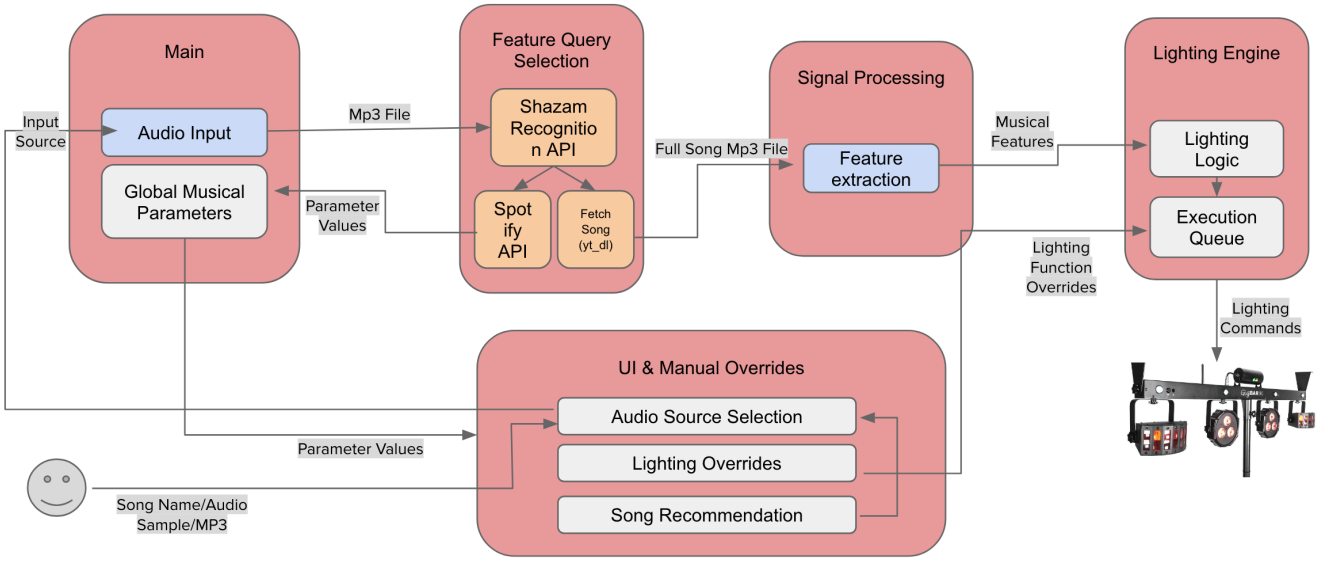

Figure 1: System Architecture

4 DESIGN REQUIREMENTS

4.1 Latency

To calculate the end-to-end latency of the system, we need to consider the entire workflow of how sound passes through the system, and how lighting calls are processed.

This workflow begins at the front end where the user uploads a file, records a short snippet of a song, or searches for a song. In each of these three modes of entry, we have different latencies to account for. The worst-case end-to-end latency comes from recording a snippet, so that will be discussed here in detail. Recording a snippet consists of a user recording an audio file for roughly 5 seconds ($record\_t$), and this is then fed into Shazam API to recognize the song ($recognize\_song\_t$) and get the name of the song. This is further used to poll Spotify for a set of global parameters for the song i.e. its danceability, valence, energy, liveness, and tempo ($features\_song\_t$). The features help us generate recommendations for songs that can be pre-fetched and pre-processed while the current song is still playing. We concurrently fetch the entire .mp3 file ($search\_mp3\_t$) based on the song title generated from the Shazam API for the signal processing engine ($signal\_processing\_t$) to get more localized audio features. We run this on a separate thread to reduce latency. Once we get the local audio features, we transfer control to the light execution engine. Lighting logic generates log files that have command queues for the lighting system to execute based on beat times ($logic\_t$). The time taken to execute logs is negligible, and lighting call to the lights is also a lightweight operation. Overall, end-to-end latency for one song after recording:

$$\begin{aligned} end\_end\_latency &= recognize\_song\_t + (features\_song\_t \text{ or } search\_mp3\_t) \\ &\quad + signal\_processing\_t + logic\_t + ajax\_t \\ &= recognize\_song\_t + search\_mp3\_t \\ &\quad + signal\_processing\_t + logic\_t + ajax\_t \end{aligned}$$

Since searching for an mp3 and signal processing happen in one thread and that takes more time than polling Spotify, we can eliminate $feature\_song\_t$ from our calculation as seen in the above equation.

4.2 Light and Channel Selection

For the scope of this project, we developed around a specific lighting fixture even though the system is dynamic and can work with arbitrary sets of light fixtures. We determined that the most cost-effective fixture to purchase is the Gigbar2 which has various different types of lights and multiple of each. This would allow us to prove that our system can work with different types of lights and it can coordinate the lighting between individual lights as well.

The Gigbar2 comes with two Par lights which shine washes of RGB and UV. It also comes with two Derbys which shine many spots on the ground which can be red, green, and blue individually but does not mix colors. The Derbys also have motors that can move the spots around clockwise and counter-clockwise. The Gigbar2 also comes with a Strobe bar which can strobe white and UV light. Finally it comes with a laser that shines little laser dots in green and red and can function similarly to the derby.

With this in mind, we also had to select which channels we wanted to manipulate. These professional lights have many channels for manual control and for pre-programmed controls. We decided to use only the manually controlled channels so we would have better control over exactly what the lights are doing. Below is a table with the channels we

decided to use, and our system was required to have the ability to control each of these channels individually at any point in time.

Table 1: Light Channels

Unit	Channel Number	Channel Description
Par 1	1	Red
	2	Green
	3	Blue
	4	UV
Par 2	6	Red
	7	Green
	8	Blue
	9	UV
Derby 1	11	Color
	13	Rotation
Derby 2	14	Color
	16	Rotation
Laser	17	Color
	19	Rotation
Strobes	21	White
	22	UV

4.3 Intuitive User Interface (UI)

The goal of the UI was to provide an entry point for users to enter their audio files through different ways, and allow them to override the existing light execution queue. The guiding principle of the UI was to keep it intuitive, easy to follow, and lightweight while still maintaining its functionality, and its ability to communicate with other subsystems. For this purpose, we locally hosted our application using Django. Django provides for fast and rapid development, is scalable, and integrates with Python easily, which was the foundation for the rest of our codebase. This multitude of reasons motivated us to use Django as our framework for implementing the (Model View Controller) MVC architecture that is crucial for a successful UI.

The user is able to stream local audio files, search for songs, and have microphone input to demonstrate what their song sounds like, and our system is required to able to process these inputs effectively to generate light shows and other recommendations side by side. The application allows for all audio formats such as .mp3, .wav, .flacc, and .mp4. To be successful in this operation, our system was required to have a 100% accuracy in taking user inputs, which was met.

According to our use-case requirements stated earlier, taking user input was a qualitative metric that we tested alongside the ease of UI use, aesthetics of the show, and the quality of the recommended songs. These requirements were tested by running user tests on our system, as summarized in the testing section.

5 DESIGN TRADE STUDIES

5.1 Feature Querying Choices

We opted for Spotify as the application to poll our audio clips because it provided a well-documented, extensive dataset of a variety of audio labels categorizing each song. Table 2 below summarizes the most important meta-features that can be used to summarize a song.

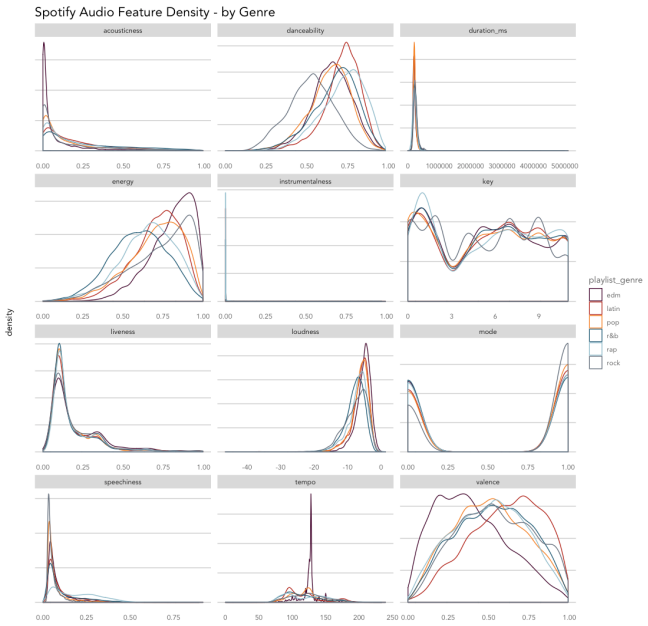

Figure 2: Spotify Audio Feature Density by Genre

Based on an R Studio Analysis we found in Table 2 [6], it was seen that 6 out of the 12 Spotify-based audio features - danceability, valence, energy, tempo, loudness, and liveness - showed maximum variability across different genres. The final features selected are summarized in Table 2 below.

This subset of features made more sense in the context of the song as this would allow us to get a holistic overview of the audio. Since these are values that show more variability, we see that these could be good determiners of the audio features a user likes in a song. Hence, they were used to generate and fetch recommended songs for the user to play next.

Table 2: Spotify Audio Feature Descriptions

Feature	Description
Danceability	Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is the least danceable and 1.0 is the most danceable.
Valence	A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).
Liveness	Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live.
Loudness	The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track. Values typically range between -60 and 0 dB.
Tempo	The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, the tempo is the speed or pace of a given piece and derives directly from the average beat duration.
Energy	Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy.

5.2 Global Feature Extraction: ML Classifier vs Shazam-Spotify workflow

We initially explored an ML genre classifier to facilitate the extraction of genre-specific characteristics. We were going to keep a sample set of 5 songs per genre for 12 genres and extract their audio features based on the genre we classified them into. The accuracy for genre classification with respect to the Decision Tree, Random Forest, Convolution Neural Networks (CNN), and K-nearest neighbors (KNN), was around $60\% - 70\%$ making it less reliable, and decreasing accuracy when we combine it with the rest of the workflow [6].

To tackle the accuracy issue, we implemented two things: detecting the song using the Shazam API, and then polling Spotify for the audio features associated with the song we detected earlier. This method made feature detection more accurate because it is dependent on Spotify's powerful audio feature extraction. However, we would not be able to test it effectively on audio clips that are absent from Spotify's database. We are assuming that any audio input when detected by Shazam will have a Spotify counterpart to help us complete the workflow even though in reality this is not the case. But, if we are not able to find a Spotify counterpart, we will not be able to generate useful recommendations. We will have error messages to alert if the audio input is not recognized by Shazam or cannot be polled from Spotify to get audio features.

5.3 I/O Communication with Lights: PyDMX vs DmxPy

To interface with the lights we wanted to use a library that would work with the Enttec Pro USB interface. The first and most commonly used interface that we found was PyDMX, however, upon initial attempts to make it work, we realized that it requires a lot of additional packages and its documentation is not great. We decided because of that to switch over to DmxPy which is a more bare-bones version but allows controlling individual channels. We determined that controlling the channels individually is all we needed given that our logic is done by the software. This means that the only communication with the interface is when we want to set channel values. Upon testing, we were able to achieve the desired functionality.

5.4 Feature Selection in Signal Processing

Signal processing attributes were initially divided into three different types of musical features: *Pitch*, *Timbral*, and *Rhythmic* attributes. This was done to ensure we were holistically analyzing the audio clip and extracting the maximum possible attributes from a clip. We used Librosa, an industry-standard audio processing tool in Python to extract these features. However, upon analysis, we found that while these metrics were useful for detecting specific elements in music, such as vocal ranges, they did not generalize well across different kinds of audio. For instance, we explored the use of Euclidean distance similarities between the differentials of the MFCCs (Mel-frequency Cepstral Coefficients) to detect instrumental change. We further tuned the hyperparameters, but it proved to be difficult to simultaneously optimize the parameters to produce accurate inflection points for multiple genres. To tackle the issue of inflection points, we found that breaking the audio into different energy divisions gave us a better representation of the audio data as listeners. Processing of this data will be elaborated on in the lighting logic section that is used to make function, light, and parameter selections.

We stuck to extracting *Rhythmic* features as before because this was a more intuitive way of analyzing audio data. The Rhythmic features provide the main beat that spans the track, and the strength of the track. Using Essentia, an open-source C++ library for audio analysis and audio-based music information retrieval, we were able to get all the times during the song when the beat landed, or beat time stamps. This maps to a more Spotify's attributes and its relevance to rhythmic features has also been explored in the Design Trade section.

We explored different types of *Pitch* features including trying to identify the key notes in any section of the song using constant-Q transform of the audio signal, and detecting onsets and transforming it to more important pure tones of the audio signal. This gave us a lot of data to work with, but we were not able to translate these to values across different octaves because a few values out of the range led to a noisy graph. Getting the frequency bins al-

lowed us to have a wider range of frequency values that could be detected easily without too much noise. This also used the constant Q-transform function, but we avoided filtering out those values more to preserve raw data per the beat time stamp.

5.5 Real Time vs Pre-processing Signal

Originally, we wanted to process audio in chunks to allow for real-time signal processing by the system. After a few attempts at chunking and streaming audio using different Python libraries, the look-ahead within the audio file was difficult to predict, and real-time alignment with the beats of the audio file would have been very difficult to achieve. Making appropriate lighting decisions based on local minima and maxima was also very challenging, and did not capture the entire flavor of the song. There was an increasing latency overhead because of the time delay in processing every subsequent audio chunk. This delay added up cumulatively, resulting in a misaligned set of decisions with the beats. For fast-tempo songs with a quicker beat, it became increasingly difficult to do a delay calculation.

We opted for a more adaptive approach that allows us to bypass the problems with real-time processing, but still maintain reasonable end-to-end latencies from a user's perspective. We chose to recommend songs based on the user's currently playing track and pre-fetch tracks that they could potentially click on i.e. the recommended songs for a given song. We got its associated audio features and requested songs from Spotify API for other songs similar to the current one. This method of loading the resource before it is required to decrease the time waiting for the resource guarantees that on consequent iterations, the user waits for a lesser time. During the buffering period of the first song, we are looking ahead and fetching these songs. If the song is not available on Spotify, we cannot generate recommendations for the song, so then we default to showing the history of playbacks.

The tradeoff because of this is that we store a few audio files on recommendations generated that might not be accessed, so storage space is slightly compromised. But, we keep recommendations to around 3 tracks to balance this tradeoff as well. We also have a history of playbacks offered for users who want to explore samples of the lighting execution. These are dynamic so the user never gets the same show every time, adding to the aesthetic appeal of the system.

While comparing latencies from the approaches, we found that our expected latency from this approach showed significant drops across more number of songs played, making it a robust and adaptive approach for the system. We expect this approach to do better than even real-time signal processing from a latency standpoint because we also avoid buffering across chunks.

6 SYSTEM IMPLEMENTATION

Our main subsystems are the Main class, Feature Query, User Interface (UI), Signal Processing, and Lighting Engine. An example workflow of how these systems interact with each other is shown in the Appendix, in Figure 12. Find our full project on our GitHub repository: https://github.com/aasuper1/Synesthesia

6.1 Main

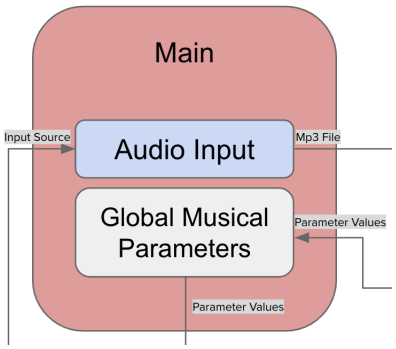

Figure 3: Main Subsystem

The main class consists of the audio input processed for its global music parameters, local audio outputs extracted from the Signal-Processor-Lighting Logic workflow, and recommendations pre-fetched in the background. This serves as the main point of contact for the UI, Signal Processor, Feature Query System, and Lighting Engine to seamlessly access the parameters it needs for process execution. We schedule tasks from the Signal Processor, Feature Query System, and Lighting Engine concurrently on sequentially depending on which one of the points of entry, Upload, Record, and Search.

In each of these cases, tasks are scheduled differently. For the upload method as a point of entry, we send the file from the User Interface to the main class. Using the Feature Query system, we recognize the song using the Shazam API using the first few seconds of the file, and we poll Spotify for features using the title. This runs concurrently with the Signal Processor as we already have the file, and we have a way to play the song and interact with the light execution queue as part of the Lighting Engine as the user can manually override existing function calls.

For the record method as a point of entry, we send the recorded snippet. Using the Feature Query System, we recognize the song using the Shazam API using the first few seconds of the file, and we poll Spotify for features using the title. This runs concurrently with searching for the mp3 and the Signal Processor. The rest is similar to the previous workflow talked about in the above upload method.

For the search method as a point of entry, the system is able to locate the song and downloads an mp3 if it does not exist. Since we are polling for the song by title, we do not need the Shazam API, and we can directly poll for Spotify

features as part of the Feature Query System. This runs concurrently with downloading the mp3. We run the Signal Processor and the rest is similar to the upload method.

This method of modularizing subsystems, and having each subsystem access the main class when it needs makes it easier to debug problems as we can isolate the problem based on the subsystem's access point with the main class. This also allows multiple processes to run in synchrony because all subsystems are not dependent on each other for the inputs, and can process inputs concurrently bringing down latency times.

6.2 Signal Processing

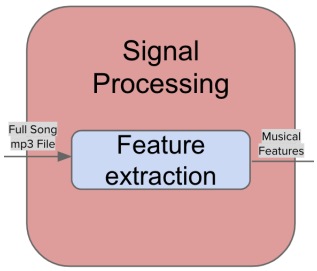

Figure 4: Signal Processing Subsystem

The Signal Processing subsystem initially consisted of windowing, followed by feature extraction, but since we are processing the entire audio file before, it consists of feature extraction after normalizing the audio to remove background noises, removing unwanted frequencies. This is followed by feature extraction through a couple of methods. The main library used to process signals is Librosa, a Python library for audio processing.

We started off with a couple of different metrics. We extracted beat time stamps, amplitudes, frequencies, energy divisions, and voiced and unvoiced parts. For amplitudes, frequencies, energy divisions, voiced, and unvoiced parts we were extracting them with a default sampling rate of 22050 samples per second. However, since beat time stamps aligned with the audio and gave us pre-determined points to execute lighting calls, we chose to align all our metrics with these time stamps. This also allows easier execution of different light groups because they are of the same lengths, and we can run them concurrently with no delays.

We realized that energy divisions per beat have a significant contribution to the overall appeal of an audio file, so we decided to weight energy divisions higher for light processing decisions as well. We wanted to see how these features would combine together. This led us to find spectral centroids. Spectral centroids allow us to weigh both frequencies and amplitudes together using Fourier Transforms to give us the brightness content of an audio file. After finding a frequency for a particular frame, we find the nearest bin and resample this according to beats. This ended up leaving some information about voiced and unvoiced parts which were captured better through changepoint detection that will be explored in section 6.4: Lighting Engine.

We discussed our other methods of feature selection in Design Trade Section 5.4: "Feature Selection in Signal Processing" and the trade-offs associated with it. We save the logs of audio features under a song's specific directory so that the Lighting Engine can access these files for further processing.

6.3 Feature Query

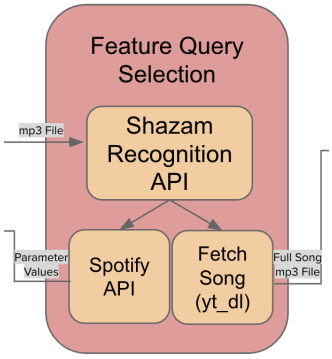

Figure 5: Feature Query Subsystem

The Feature query subsystem consists of a few main parts: Shazam API is used to detect the song that is passed through the UI. After the song is uploaded through the UI, the data from an uploaded file is chunked into smaller bits to allow songs to be recognized quicker. Since it could be labeled differently from what the audio file is, we take the safer route of sending it through the song recognizer to find the title of the song.

For the record option, we pass the 5-second audio clipping as it is. We use the Shazam API to first read a chunk of the song passed in, and retrieve a signature generator object. This signature generator object gives us many parameters about the song that it identified.

Using the title of the song, we then poll the Spotify API with a fixed set of client credentials and a secret key with an access token. Polling Spotify gives us the ability to extract audio features with respect to the song on the database. Here we are assuming that whichever song we detect from the Shazam API is present on the Spotify database. Spotify requests for a global set of parameters are concurrently executed with the signal processor that finds the main local features per the beat time stamp. If it breaks in either one of these steps, we have certain fail-safe defaults to still have a history of playbacks presented in the recommended section. We are still able to run our signal processor, so we do have local audio features to work with to make lighting decisions. We query Spotify with the song title using the track URI because that's a unique identifier, and we get back a data frame with a few audio features stacked. From this nested dictionary or data frame, we are able to get the danceability, valence, energy, tempo, loudness, and

liveness values. The user has the authority to override these features, and accordingly control the lighting.

6.4 Lighting Engine

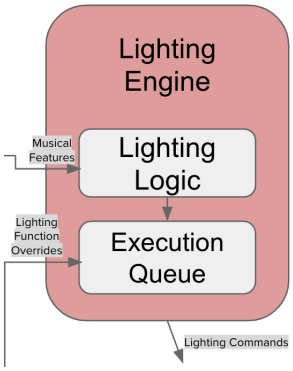

Figure 6: Lighting Engine Subsystem

6.4.1 Lighting Logic

The Lighting Logic is the key component for the functionality of this system. Logic receives arrays of signal parameter values from the Signal Processor and reads them. Once the raw values are read, Logic processes each of those values independently depending on the decision it is trying to make. For instance, in order to interpret the energy divisions, Logic needs to convert negative decibel values to a positive range, then remove any outliers that may be causing additional noise. Following this, it normalizes all of these values on a proportional scale and then runs an ML classifier to find the critical places where some key component of the music is changed. However, this also requires tuning the hyper-parameters such as the sensitivity, so that an adequate number of change points are generated. An example of detecting such change points for the energy values is shown in Figure 7 below.

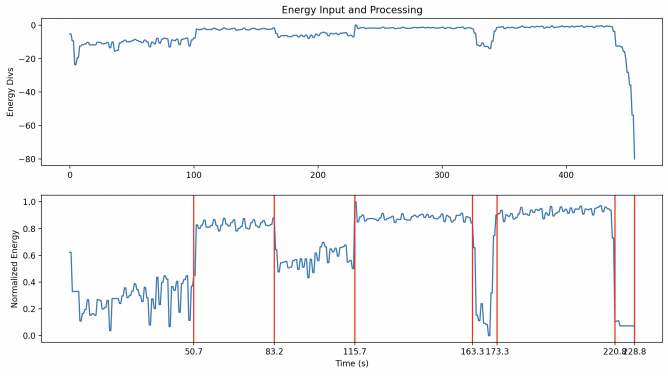

Figure 7: Processing Raw Energy Input and Automatic Change Point Detection for Teenage Dream by Katy Perry

Once the time stamps for different sections of the song are generated, this data is used to make decisions about various attributes for each light. The attributes that the Logic decides for the GigBar2 can be summarized as follows:

1. Lights: Logic first selects which lights should be on at any given point during the song
2. Function: For each light of the rig, the function that should be executed on the light is selected for every beat of the song. This process is probabilistic, based on an elimination scheme, so that the light shows look dynamic.
3. Color: The colors for each light are selected based on 3 criteria: hues (number of colors and color palette), saturation (depth of color), and value (brightness of the color)
4. Rotation Speed: The direction in which the light should be rotating and the speed of the rotation is determined
5. Effect Duration: The number of beats for which a function should last is selected. Effect durations can range from 1 beat to 4 bars depending on the current state of the song
6. Effect Frequency: The number of times an effect is displayed in a given duration is determined

Once all of these parameters are determined for each light for the entire duration of the song, they are logged and sent to *Main* to be relayed to the Execution Queue.

6.4.2 Lighting Communication and Execution Queue

The lighting library follows an object-oriented model, where light is defined as an instance of a class. The class determines the basic functionality of the light and defines how the light should behave when it is asked to execute different commands such as fade, rotate, and blackout. Different kinds of lights have different functionalities, so the lights inherit from a super-class and then define this unique functionality.

This modular approach to creating a light setup allows the user to scale the execution library to multiple lights with ease. Further, lights can be combined into a group to ensure that the operations of these lights can happen in sync. This is defined by the library as a LightGroup. The LightGroups are available if none of their member lights are currently executing any lighting commands. Once a LightGroup is chosen, a function is chosen from the available LightGroup functions. Finally, the parameters for that function are set. Each of these decisions is influenced by the values that are sent from the Lighting Logic.

The Lighting Execution Queue then takes these function calls and places them in a queue that handles any concurrency. This queue communicates directly with the light and calls the respective functions in harmony with the beats.

6.5 Web Application and User Interface

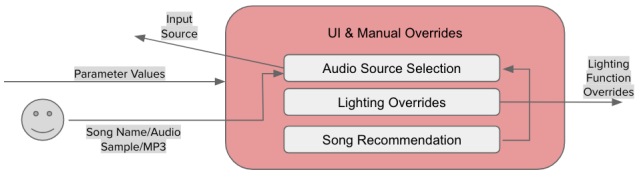

Figure 8: Web Application and User Interface Subsystem

The home page has 3 methods of song uploads as visible here. You can upload a song, search for a song or record a song.

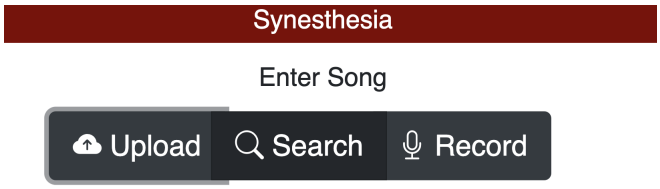

Figure 9: Home Page of the User Interface

The entry points are broken up into 3 main categories. In each of these cases, we have different processing methods.:

- Upload: Upload gives users the ability to upload any audio file from their local computers. This also feeds into the recommendation engine that we talked about.

- Record: Record gives users the ability to feed the system what the song they are thinking of sounds like. They play the song for a sum total of 5 seconds, and our system is able to generate a playback of the entire song with a dynamic light show within a couple of seconds.

- Search: Search gives users the ability to poll our system for a song title, and a dynamic light show is outputted within a few seconds.

  Users can manually override existing function calls sent to the lighting engine. These overrides will happen through simple button presses of setting a color, activating a strobe, or putting a hold or a blackout at a given instance. Overrides are dynamically updated on the UI and relayed to the lighting execution queue through AJAX requests.

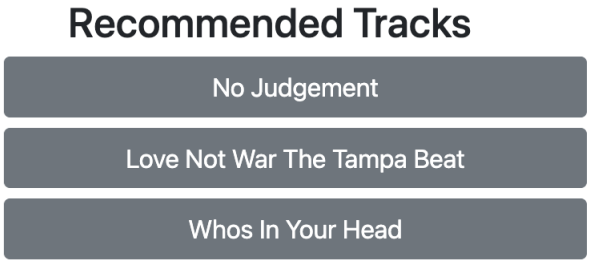

Figure 10: Recommended Tracks for "Levitating" by Dua Lipa

For each time we get the *track_uri* from Spotify, we try to run recommended songs and pre-fetch the tracks as well. This way we can pass it through the signal processor, and retrieve its audio metrics for Lighting Logic to execute, and generate an execution queue of calls to the Gigbar2. This helps save time and makes it more tuned to user preferences as we talked about earlier

7 TEST & VALIDATION

7.1 Lighting Execution

For the lighting execution testing, we focused on ensuring that all the functionality that we intended to be able to output on the lights was possible. This involved stress testing each light with each of the functions to ensure that given arbitrary features from the signal processing units, the light execution would be able to handle the load and display the lights effectively.

7.1.1 Functionality

In terms of functionality, we wanted to ensure that each of the lights could handle all of the lighting functions sequentially. To this end, we created a simple testing plan to ensure that no functionality was missed. In order to execute these tests, for each light and each function, we randomly generated parameter values and tested if the lights matched those parameters. The results of these trials are shown in the table below. Note that the Blackout function only had one trial per light because it does not take any additional parameters.

7.1.2 Modularity

In terms of modularity, we wanted to be able to use multiple lights in conjunction with each other. To this end we tested calling functions on pairs of lights at the same time. This test allowed us to ensure that we could control multiple light groups simultaneously. For example we tested a setColor command on the pars and the derbys at the same time. We did this test for every combination of lights on the Gigbar 2. We found that every pair of lights

were able to execute their respective commands simultaneously. This allows for full functionality once the signal processing functions are passed in.

7.1.3 Concurrency

For some of the light functions, there is the chance of concurrency issues where multiple light function commands are requesting the same light resource. To handle these requests, we implemented locking on each of the hardware light resources so that only the most recent light function is able to execute their command and any existing command will be overwritten. In order to test this, I tested the lighting calls that arrive from the logic on full songs. These logic files will call all of the different functions available to each light and test that there are no deadlocks. The deadlocks would be detected by the system when we can see that none of the subsequent lighting functions are being called. We isolated each of the hardware lights since they all hold unique locks, and tested the diverse set of functions on each of the lights, and then tested them all together. We were able to prove the robustness of the system over the course of 5 different songs where there were 0 deadlocks.

7.2 Latency

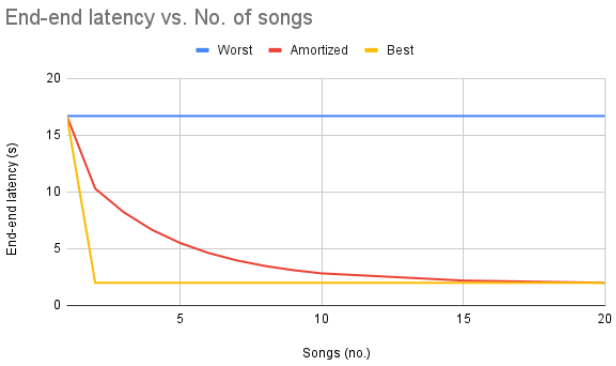

Figure 11: End-to-end latency across the number of songs

Based on a few latency tests we ran, we graphed the worst-case latencies, amortized latency due to recommended songs, and best-case latency if all songs from history or playback are clicked by the user. We see that the amortized latency from the recommendations on subsequent songs gets better and converges towards the best-case latency. This allows us to effectively find a feasible alternative to the issue of chunk processing, while still maintaining a user-focused approach.

7.3 Change Point Detection

Regarding change point detection, in each song there are audible change points that are present. In order to test the accuracy of the change point detection algorithm, we labeled the song with the timestamps of these changes. Then we compared those timestamps to the timestamps of changes that the algorithm came up with. Below we have a table with the result of the percentage of change points that the algorithm was able to pick up on.

Table 3: Accuracy of Change Point Detection Across Songs of Different Genres

Song	Change Points	Accuracy
Teenage Dream	7	100 %
Jiya Jale	13	82 %
Flowers	11	100 %
Rap God	17	80 %
Dandelions	14	95 %
Pepas	9	96 %
	Average	**93 %**

7.4 Song Recognition Accuracy

It was important for the robustness of our system to test the ability of our song recognition algorithm. In order to do so, we curated a list of 15 different songs and ran the Shazam detection algorithm on each of these songs to determine if we would be able to recognize them. We played a 5 second clip of each song and recorded either a success of fail depending on if the algorithm was able to detect the song correctly, or either fail to detect the song or detect the incorrect song. We found that across all 15 songs, the Shazam algorithm was able to correctly identify the song.

For additional testing purposes, we also tried to sing the songs that it was able to pick up on to see if the system could be expanded to detect songs from user voice. This however failed for each of the 15 songs.

7.5 User Testing

In order to get a feel of how a user would interact with our product, we asked a group of 5 volunteers to rate their experience with the project on a scale from 1 to 5. They ranked the project based on four different criteria listed below.

Table 4: User feedback on a scale of 5

Metrics	Avg Rating (out of 5)
Setup time	5
Recommendation effectiveness	4.1
User Interface Intuitiveness	3.5
Aesthetics	4.7
Average rating	**4.3**

8 PROJECT MANAGEMENT

8.1 Schedule

Our project was ambitious and involved the development and integration of multiple subsystems that include audio processing, communicating with lights, a modifiable user interface, logic to map lights and feature extraction. To achieve this, we set up and followed an aggressive timeline, trying to get a headstart on development and testing, and saving time wherever possible.

While we tried to follow the schedule we had established at the beginning of the project, we were cognizant of the changes we made along the way and were flexible enough to accommodate them into our workflow. As the project evolved and increased in complexity, we ended up spending a lot more time integrating the various subsystems than we had initially anticipated. Additionally, we spent a lot more time fine-tuning and optimizing each subsystem than we had planned. Our final schedule breakdown can be seen in the Gantt Chart in Figure 14 at the end.

8.2 Team Member Responsibilities

Given the project was divided into multiple subsystems, each team member was responsible for a different part of the project as listed below:

- Abhishek worked on integrating logic, representing physical lights in software, and executing lighting commands on the lighting hardware.
- Parth was responsible for processing signal parameters and developing the lighting logic to map audio features to light outputs.
- Rachana gathered audio information from the UI and decomposed and extracted local signal parameters and global parameters. She was responsible for the User Interface, and synchronizing processes with the different subsystems.
- All members contributed to the testing and integration of their subsystems into the overall project.

8.3 Bill of Materials and Budget

The bill of materials and budget for our project can be found in Table 3 below. Procuring the light rig is the biggest expense, taking up a large proportion of the budget. All the other equipment, including the 3-to-5 DMX pin, the DMX-USB interface, and DMX and power cables were borrowed from the CMU IDeATe lab. The only additional hardware was a personal computer to run the program on. Overall, we ended up using all of the equipment that we had procured at the start of the project.

Table 5: Breakdown of Expenses

Product	Manufacturer	Quantity	Cost	Total
DJ GigBAR 2 4-in-1	CHAUVET	1	$419	$419
DMX USB Pro 512	Enttec	1	$120	$120
DMX 5F 3M Adapter	CHAUVET	1	$8.99	$8.99
AC3P DMX Cable	Sweetwater	2	$8.99	$17.98
IEC6 Power Cable	Sweetwater	2	$5.99	$11.98
				577.95

8.4 Risk Management

While designing our system, we conducted an initial assessment of potential risks and developed strategies to mitigate them. Additionally, we created several internal checkpoints and test metrics for each subsystem to ascertain the project's feasibility despite concerns about its complexity and timeline. The list below outlines the risks and the mitigation strategies that ensured this project's growth.

8.5 Controlling Lights and Procuring Equipment

The primary risk for the project was efficient communication with the lights. Given our team's inexperience with using the DMX protocol, it was critical to test if we could communicate with lights on a smaller scale. To test this, we borrowed individual pars from the CMU IDeATe lab and produced various proof of concept demos for controlling different functionality of lights with our program. Once we were confident in our ability to control lights, we allocated time to research and procure the light rig that we used. This was done to ensure that we were using our budget judiciously and avoiding delays caused by a light rig that was not suitable for the needs of this project.

8.5.1 Latency and Fail-Safe Defaults

A high enough latency would have caused the light commands to run out of sync with the music. As discussed in the design trade, we let go of real-time processing of individual audio chunks in favor of pre-processing the entire file to combat this. Additionally, we incorporated a lighting queue in our design that allowed us to store a set of lighting instructions, execute them simultaneously for different lights, and replace them with user inputs if needed.

Further, we replaced the need for Fail-Safe defaults by processing the entire file ahead of time and getting rid of all real-time behavior.

8.5.2 Scaling and Independent Operation of Lights

Due to limited time and resources, we were uncertain if we would be able to make our project work for multiple different kinds of lights present on our light rig. However, continually testing smaller subsystems ensured that we could coherently operate each light independently and in sync, and allowed us to scale without many setbacks. Further, by using an object-oriented design for our lighting engine, we were able to make it easily scalable and adaptable to different kinds of rigs in the future.

8.6 Audio Processing Reliability and Integration

The final risk for our project was our ability to get useful features from the audio that we could reliably base all of our lighting decisions. As highlighted in our system implementation, we tried extracting many features across a wide variety of songs. This was a necessary step because while some features worked on certain kinds of music, they were not as reliable for others. Finally, we developed the lighting logic keeping modularity in mind, so that if in the future, we decided to add more signal processing features or replace the existing ones, we could do that without any additional development time. Such a development style significantly expedited our integration process.

9 ETHICAL ISSUES

Given that this project was designed keeping the needs of entertainers and musical performers in mind, the ethical quandaries that might arise are minimal. However, there are still some important implications to consider.

9.1 Displacement of Light Engineers

While automation offers the benefits of speed and precision, it may result in job loss for people who previously performed these tasks manually. This may be particularly concerning for individuals who have invested time and resources in developing their skills as light engineers. While our system is relatively simplistic to cause such a disruption, a future system with enough customizability might remove the need for a human. To mitigate this impact, it may be important to offer training opportunities for these individuals so that they can create even more engaging shows and performances with the supplemental aid of automation as opposed to competing with it.

9.2 Public Health and Safety

Strobe lights or rapidly changing colors can trigger seizures in individuals with epilepsy or other neurological conditions. In addition, exposure to bright or flashing lights can cause eye strain, headaches, and other discomforts. A sudden failure or malfunction of the lighting system could also put performers and audience members at risk of accidents or injuries. To mitigate these risks, it is important to provide warnings about the potential risks of the light show and to include fail-safe defaults and bypass mechanisms to quickly shut down lights in the event of an emergency. Additionally, it may be important to conduct extensive testing and quality assurance to ensure that the product meets safety standards.

9.3 Public Welfare

The environmental impact of the lighting system is another consideration. In addition to the extensive computation required to automate light shows, certain types of lighting can consume large amounts of energy, contributing to carbon emissions and climate change. To address this, it is important to consider the environmental impact of the lighting system in the design phase. This may involve selecting lights with low power consumption and developing an efficient program that requires low computation power and does not do unnecessary calculations. It may also involve balancing economic and environmental considerations, where cheaper lights may be less environmentally friendly. By prioritizing sustainability in the design phase, it may be possible to reduce the environmental impact of the lighting system and promote public welfare.

10 RELATED WORK

There are a variety of systems available in the market that address the need for customizable lighting in different circumstances. The closest match to our project is a new startup, MaestroDMX, that aims to produce automated light shows for live audio using artificial intelligence [7]. Other projects have tried to tackle scenarios other than performances. For instance, Protopixel has developed dynamic lighting systems for fully reactive light-art installations [8]. Another system focuses on mood lighting that expresses the emotions evoked by the audio [9]. This type of system analyzes the emotional impact of music and color and seeks to maximize their impact when used together.

Our project was inspired by the real-time element of dynamic lighting, which is used to reflect tension in gaming [10]. We examined the possibility of creating our Lighting Engine using this approach. Games, like performances, simulate elements of traditional media such as plot, characters, sound, music, and lighting. The interactive experience in games presents player challenges that are similar to audience-performer interactions.

11 SUMMARY

We started development on this project with the goal of creating automated light shows that were as engaging as manually programmed shows but took a fraction of the time. By creating classes for lights and translating auditory features to lighting logic, we are able to create systems that are modular, testable, and scalable. Having an interface that allowed the user to modify the light show, this project successfully demonstrated the capabilities of such a system at a smaller scale, with four light pairs.

11.1 Future work

The complexity this project was able to achieve in a short time span is impressive, but there are ways to expand it in the future.

11.1.1 Machine Learning for Logic

Currently, our lighting logic relies exclusively on the signal parameters that we extract from the songs. The accuracy and complexity of logic can be improved by incorporating machine learning models trained on manual light shows. This would enable the system to learn from the intricate and nuanced lighting techniques of experienced light engineers. However, the lack of good-quality data that the model can be trained on remains a challenge.

11.1.2 Added Customizability

The system currently allows the user to modify the current command that is executing on the light. However, different performers might prefer different lighting aesthetics, and for that purpose, may require added customizability options. For instance, allowing the user to select a color palette or modifying the overall speed and energy of the show can be an improvement.

11.1.3 Improved Latency and Scaling

Our system in its current state is already orders of magnitude faster than any manual light show. However, there are still significant improvements to be made that will allow the system to be able to listen to live music and make decisions on the fly. Further, while our system is scalable, it can be further tested on different types of lights and light rigs, and can potentially allow users to specify their own equipment.

11.2 Lessons Learned

If anyone wants to explore the domain of automated light shows for live music, they should be aware of the limited documentation and resources that are available for public use. Most of the signal-processing libraries are inaccurate and slow. These libraries often require a lot of bloatware to function and are not kept up to date, causing compatibility issues. Further, the signal parameters are not very transferable and may only work for a limited subset of musical genres.

Another roadblock to keep in mind while working with DMX is the scalability and power constraints. Many light rigs on the market often require a lot of power and may need a dedicated controller to communicate and daisy chain. This is not feasible for a mobile workstation such as a laptop. Furthermore, communicating with multiple lights in real-time requires knowledge of concurrency and may cause the system to break abruptly.

Finally, integrating signal processing with a web application and a lighting engine requires a lot of moving parts, and is a challenging task to accomplish on a strict timeline. It is essential to over-allocate time for integrating and testing various subsystems, as interactions between the different subsystems are very likely to produce unanticipated results.

Glossary of Acronyms

- API – Application Programming Interface
- ACPS – Average Calls Per Second
- AJAX – Asynchronous Javascript and XML
- CNN – Convolutional Neural Nets
- CSS – Cascading Style Sheets
- DMX – Digital Multiplex
- ELE – Expressive Lighting Engine
- KNN – K Nearest Neighbors
- MFCC – Mel-frequency cepstral coefficients
- MVP – Minimum Viable Product
- PCPS – Peak Calls Per Second
- QLC+ – Q Light Controller Plus
- UI – User Interface
- UV – Ultra Violet

References

1. Moody JL, Dexter P. Concert Lighting the Art and Business of Entertainment Lighting. New York: Routledge, Taylor amp; Francis Group; 2017.

2. Henrie J. How long does it take to set up a stage for a concert? Ennui Magazine. https://ennuimagazine.com/how-long-does-it-take-to-set-up-a-stage-for-a-concert/. Published March 1, 2020. Accessed May 5, 2023.

3. Wiebe DA. 8 best DMX stage light control software amp; hardware 2023; for clubs, churches, djs amp; more. Music Industry How To. https://www.musicindustryhowto.com/best-dmx-stage-lighting-control-software/. Published April 10, 2023. Accessed May 5, 2023.

4. Biamp. Video and network latency. Biamp Cornerstone.https://support.biamp.com/Tesira/Video /Video_and_network_latency. Published May 24, 2018. Accessed May 5, 2023.

5. Minotto VP, Jung CR, da Silveira LG, Lee B. GPU-based approaches for real-time sound source localization using the SRP-phat algorithm. The International Journal of High Performance Computing Applications. 2012;27(3):291-306. doi:10.1177/1094342012452166

6. Kaylin Pavlik. Classifying genres in R using Spotify Data. Kaylin Pavlik. https://www.kaylinpavlik.com/classifying-songs-genres/. Published January 3, 2020. Accessed May 5, 2023.

7. Kickstarter. https://www.kickstarter.com/projects/limbicmedia/maestrodmx-a-game-changing-technology-for-djs-and-musicians. Accessed May 6, 2023.

8. Events. ProtoPixel. https://www.protopixel.io/showcase/events Published April 1, 2020. Accessed May 5, 2023.

9. Moon CB, Kim HS, Lee DW, Kim BM. Mood lighting system reflecting music mood. Color Research amp; Application. 2013;40(2):201-212. doi:10.1002/col.21864

10. Game studies. Game Studies - Dynamic Lighting for Tension in Games. https://gamestudies.org/0701/articles /elnasr_niedenthal_knez_almeida_zupko.Accessed May 5, 2023.

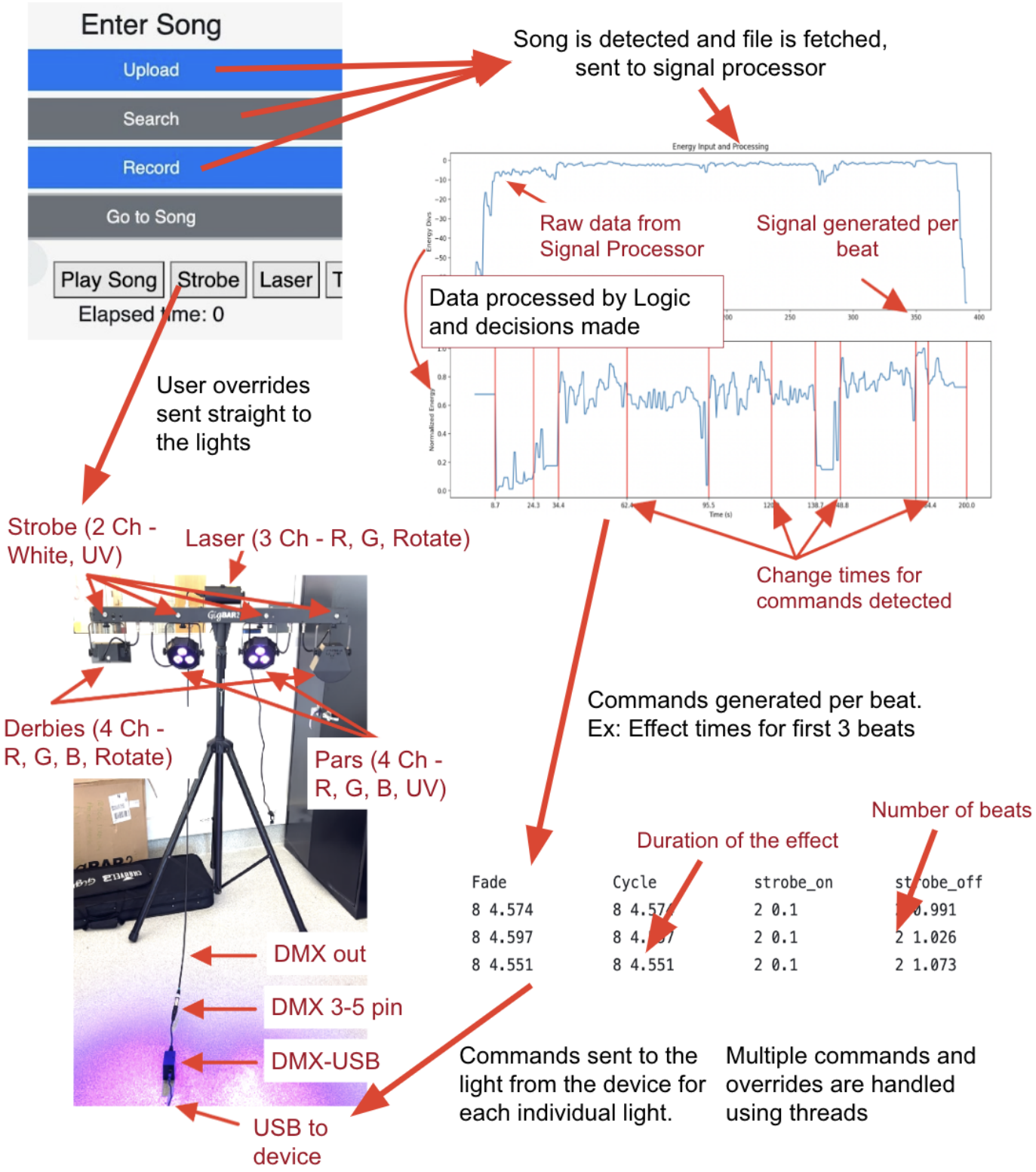

Figure 12: Complete Setup and Example Execution Workflow for a Song

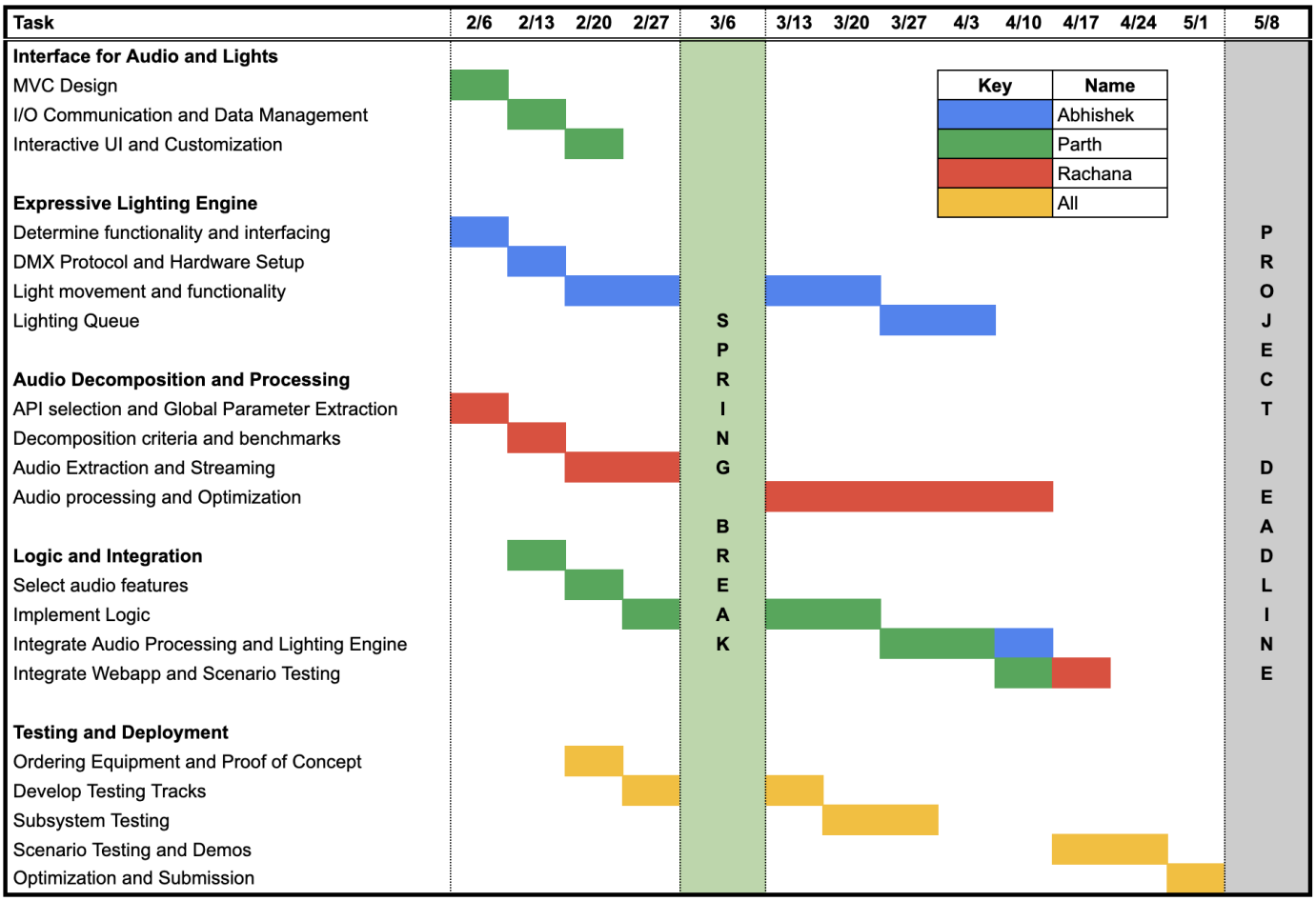

Figure 13: Completed Task Breakdown and Schedule