Synesthesia

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

***Abstract*—This project aims to create a system capable of producing immersive light shows for novel audio inputs in real-time. Given the time and resource constraints of the course, we will build a program to control four types of light pairs - par, laser, derby, and strobe - both in unison and independently. To achieve this, we will build a signal processing system to process audio with a latency of less than 185 ms. We aim to develop scalable algorithms for audio processing, light show customization, automation, and communication that will allow small performers to reduce the need for expensive equipment and reduce setup time.**

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

1 INTRODUCTION

Stage lights are a paramount tool to engage the audience during musical performances. Performers use stage lighting that is controlled using audio interfaces, but they often have limited customizability [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 softwares such as QLC+ allow some stage setting and lighting customizability [3], performers are expected to spend a significant amount of time programming light behavior ahead of the performance. This capstone project proposes a dynamic lighting system that automates this process by analyzing audio inputs to control lights. This project will involve developing a system that can take in an audio input, decompose it into key components, identify features, and control a set of light pairs in close to real-time for multiple genres.

2 USE-CASE REQUIREMENTS

This project aims to control one set of light pairs that include 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. In order to achieve this, our system must be able to process the inputted audio and communicate with the lights in almost real-time. 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 amount of permissible latency for this system. Further, user testing and anecdotal evidence revealed that on average it takes a performer about an hour to set up the equipment for a performance. Hence, this system strives to add less than 5 minutes to the overall setup time, in order to minimize costs. Finally, without any significant increase 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

Decomposition of the inputted audio will determine the behavior of the lights at any given point in time. Hence, it is important that relevant features of the audio are being extracted. In order to achieve this, the signal processing subsystem should be able to detect and extract over 90% of the auditory features as compared to a hand-labeled waveform. This is done to ensure that there is enough data for the lights to reflect all the major and minor inflections in a song. Moreover, audio processing efficiency can be broken down into two major components: Latency for simple and complex audio tasks, and the sampling rate at which we can process them. Several state of the art real time audio processing systems can achieve latencies as low as a few milliseconds with processing times of 1-2 ms for simple audios, and 5-20 ms for complex audios. Thus, it is reasonable to expect that we can achieve processing latencies of 20-30 ms. Recent research shows that sampling rates of 40,000 can be processed at high speeds, [5] showing us that real-time audio processing of several thousand audio samples with minimal lag is possible using modern hardware.

2.3 Manual Adjustments

While automating the light shows can save significant amounts of time for performers, it is crucial for the system to provide some customizability so the performer may fine tune the lights. In order to accomplish this, the system allows the users to modify meta characteristics of the audio, as described in Table 2. However, given the selling point of

this project is automation, the system should not require more than 3 manual overrides per minute, which was determined as per the windowing of the audio chunks for signal processing. Further, these modifications are meant to modify the overall feel of the lights. The system aims to produce noticeable changes in light behavior when users adjust the slider, making it an important qualitative requirement.

3 ARCHITECTURE AND PRINCIPLE OF OPERATION

For the overall architecture of the project, we decided to divide our systems into five main subsystems. We decided that it would be easier to split the work and unit test functionality by doing so. Together these components will cover all the functionality that we require of the system as a whole.

3.1 Show

The Show component will function as the main shared system that interacts with each of the other subsystems. It will contain information regarding the audio source, and it will be streaming audio from either a microphone or file. It will also store the current musical parameters for each of the other subsystems to view. Additionally, it will also contain the Light Set, a software representation of the lighting rig we have available to us. For testing and demonstration purposes, it will be a Gigbar 2. Show will also contain functionality to communicate with the lights and execute lighting changes.

3.2 Feature Query

The Feature Query subsystem will allow us to identify a song and extract the musical parameters of the song. It will perform the song detection using the Shazam API and then perform a song name query using the Spotify API. From that we will be able to detect some global musical parameters such as danceability, valence, energy, tempo, loudness, and liveness. If we are able to detect the song, we can use these parameters. Otherwise, the user's inputted values will be used. The user will also have the option to manually override these parameters with their inputted ones.

3.3 User Interface and Manual Overrides

The UI subsystem will be how the user interfaces with the program. It is designed to be simple yet enable full functionality of the system without the user needing to touch any code. It will include a way to select an audio source, initialize the LightSet to match the available hardware, adjust and display the global musical parameters, and indicate when they want to override the Feature Query subsystem.

3.4 Signal Processing

The Signal Processing subsystem will be responsible for ingesting real-time audio and outputting the real-time musical features. It has two main components. The first is the sampling and windowing component which will be responsible for reading in chunks of live audio bits and feeding it to the next component for processing. The real-time feature extraction will analyze the chunks and generate features from these chunks that will be used as cues for the Lighting Engine.

3.5 Lighting Engine

The Lighting Engine overall will be ingesting the real-time audio features, the current LightSet state, and the global parameters. It will output lighting function calls. This will contain lighting logic that will make decisions of what functions to call and when to call them. It will then send these calls to the execution queue which will store an ordered list of all the function calls that are pending and handle the execution of these function calls.

4 DESIGN REQUIREMENTS

4.1 Latency

To calculate the latency of the system, we need to consider the entire workflow of how sound passes through the system, and how lighting calls are processed. Time through the feature query system is omitted as songs can be detected, features can be extracted from Spotify, and they can be passed to the UI as sliders in parallel along with the major chunk of audio processing, and lighting changes.

Chunk processing time (*chunk_time*) as calculated in the Design Trade Section, Audio Chunk Size: Latency vs Feature extraction. At any point in time for our lighting engine, we need two reference chunks - the current chunk and the previous chunk - to detect deflections for the lighting engine. Extraction of features (*feature_extraction_time*) adds to this time as we have to process three types of features for each chunk.

With respect to the lighting engine, we need to factor in time to make lighting calls (*lighting_call_time*) which include eliminating incompatible function calls, and finding which calls work for a certain chunk of audio. Lighting execution (*lighting_execution_time*) is the communication facilitated with the DmxPy controller, and how we talk to the Gigbar lights.

We want to be able to subtract the time taken for audio chunks to reach the speakers we are looking to minimize the difference between when the lights show up compared to when the audio reaches the listener's ears.

$$\textbf{\textit{end-end latency}} = 2\times(\textit{chunk\_time} + \textit{feature\_extraction\_time}) + \textit{lighting\_call\_time} + \textit{lighting\_exeuction\_time} - \textit{audio\_speaker\_time}$$

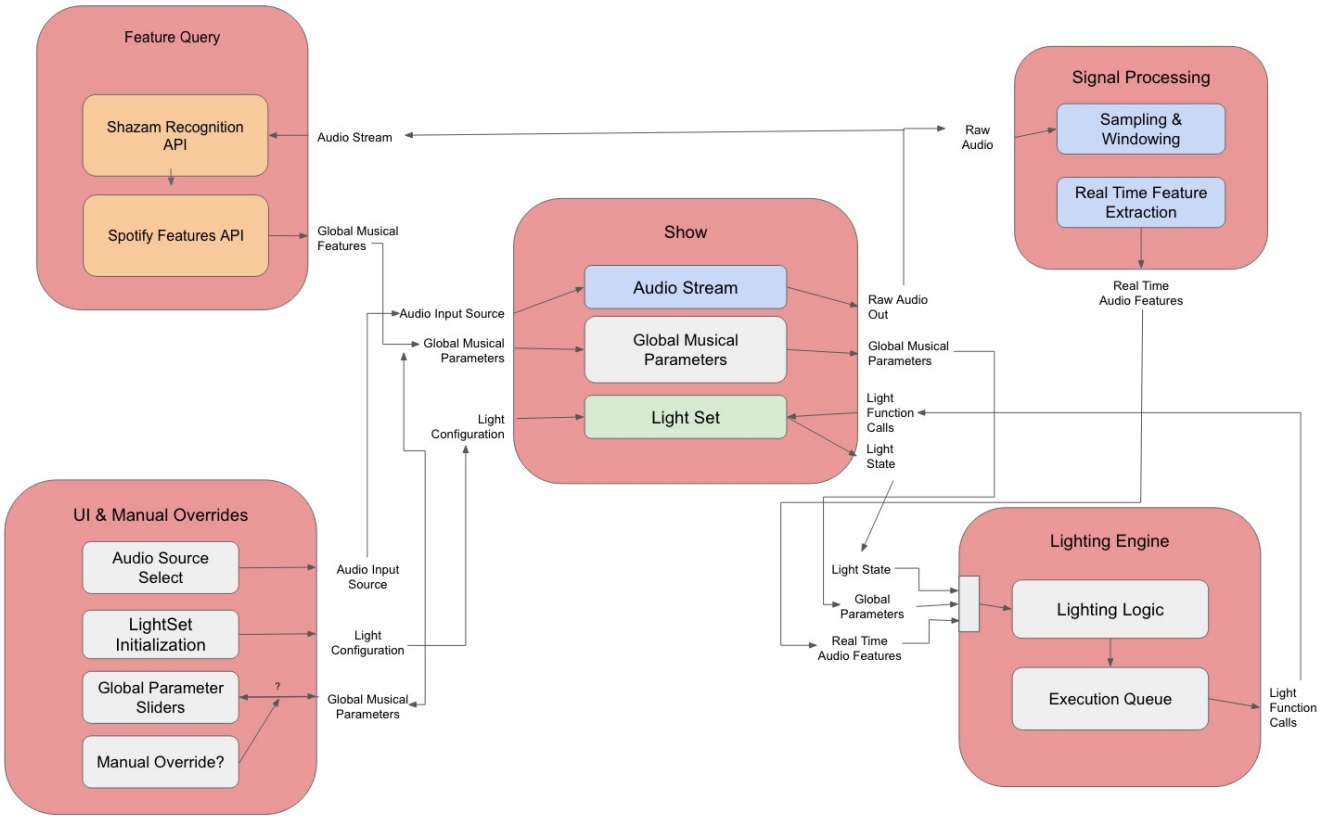

Figure 1: System Architecture

4.2 Signal Processing Attributes

We are breaking signal processing attributes into three different types of musical features: *Pitch*, *Timbral*, and *Rhythmic*. This is to ensure we are holistically analyzing the audio clip and extracting the maximum possible attributes from a clip.

Timbral features are used to characterize sound by properties. These properties relate to instrumentations or sound sources such as music, speech, or environmental signals. The audio features that we can extract to get a better representation of timbral features are not limited to Zero crossing, Spectral centroid, Spectral rolloff, and Mel-frequency cepstral coefficients (MFCC)

- Zero crossings: This gives us a sense of amplitude inflections, and we can adjust the intensity of light flashes accordingly
- Spectral centroid: This indicates where the center of the mass of the spectrum lies, and it has a robust connection with the brightness of the sound. We can correlate this with the brightness of lighting or our color choices.
- Spectral rolloff: This indicates the nth percentile in the power spectral distribution. This is useful in distinguishing voice, and unvoiced aspects of the sound where most of the energy for voiced speech is contained in the lower bands of the spectrum. This allows us to differentiate between the background music, and stanzas, giving us scope to adjust lighting for the score.
- Mel-frequency cepstral coefficients: This allows us to differentiate between certain voice features including true voice, tolerance voice, and false voice. These features will allow us to identify and act on melodic features within the voice itself

Rhythmic features provide the main beat that spans the track, and strength of the track. Beat extraction, and Spotify's energy attributes can help isolate these features for further processing.

- Beat extraction: This helps us identify positions where dynamic inflections occur within the song with respect to the base sounds. This helps with flashing lights rhythmically as well.
- Feature Querying: Spotify's attributes and their relevance have been explored in the Design Trade Section

Pitch features explore the pitch histograms generated by the audio clip, and help analyze the overall melody of the song.

- Pitch Histograms: This helps us to represent the pitch content of music signals both in a symbolic and an audio form. Symbolic features are similar to musical scores where we get the start, duration, volume, and instrument type of notes of a musical piece. This distribution of pitches helps us get genre-specific attributes too through which we can tag lighting output.
- Melody extraction: This is done using pitch contour characteristics. Pitch contours represent a series of consecutive pitches which are continuous in both time and frequency. Using sinusoid extraction and salience attributes, this can help us get the pitch contours. Melodies help isolate lull points in a song to enact fades, and transition to different colors.

4.3 Light and Channel Selection

For the purposes of the project we wanted to build based around a specific lighting fixture even though our end goal is to have the system be dynamic and work with arbitrary sets of light fixtures. We determined that the most cost-effective fixture to purchase is the Gigbar 2 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 Gigbar 2 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 Gigbar 2 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 have decided to use.

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

Table 1: Light Channels

4.4 Lighting Call Rate

The Lighting Engine will need to make many lighting calls throughout a performance, and may end up being overloaded with requests. For this reason it is important to set targets regarding how many lighting calls our system should be able to handle in a given time frame. This will be measured by two metrics. The first will be average calls per second (ACPS) and the second will be peak calls per second (PCPS). ACPS will be measured as the 10 second moving average of the rate of calls that are performed. PCPS will be the 1 second moving average of the rate of calls that are performed.

$$ACPS_t = 1/10 \sum_{i=t-9}^{t} 1_{\{function\ call\}}$$

$$PCPS_t = \sum_{i=t-1}^{t} 1_{\{function\ call\}}$$

The goal is to be able to tolerate up to 150 BPM music with an ACPS of 2.5, in order to change the lights at every beat potentially. And for extremely exciting points in the music we should also be able to support a PCPS of 5 to support changing lights with respect to an eighth note during exciting points.

4.5 User Interface (UI)

The primary goal for the UI is to take in two types of inputs from the user: the audio file and the global parameter modifications. Hence, the guiding principle for the UI is to keep it as lightweight as possible while still being able to communicate between the other subsystems. For this purpose, we are using a locally hosted web application that is built using Python and Django.

The specification for the audio input dictates that user will be able to access local audio files and stream them from the webapp one at a time. To achieve this, the application will allow audio inputs in the form of .mp3 and .wav files as they are the most commonly used file formats for audio. A post-MVP reach goal would be to also allow a direct microphone input to control lights for a live performance.

As per our use-case requirements, the user is not expected to modify the global parameters more than 3 times in a minute. These modifications will be taken in using CSS sliders, and the changes in the slider values will be tracked using 5 second AJAX calls. Keeping the update time 5 seconds will allow us to still meet the use-case requirement, while still allowing the user to update the sliders more frequently if they wish. Also using a slower AJAX call will make the UI more lightweight, and hence improve the latency.

5 DESIGN TRADE STUDIES

5.1 Feature Querying Choices

Based on an R Studio Analysis we found Figure 2, it was seen that 6 out of the 12 spotify based audio features showed maximum variability across different genres.

We opted for Spotify as the application to poll our audio clips because it is a well-known, and well-used app with a large database of audio clips. From the graphs, we can make out that danceability, valence, energy, tempo, loudness, and liveness.

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.

Table 2: Spotify Audio Feature Descriptions

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. These are also features that an audience can relate to, so we are programming our sliders on the User Interface (UI) to first reflect the initial values for these features polled from Spotify, and then further allow the users to make changes to these sliders to change lighting as they wish.

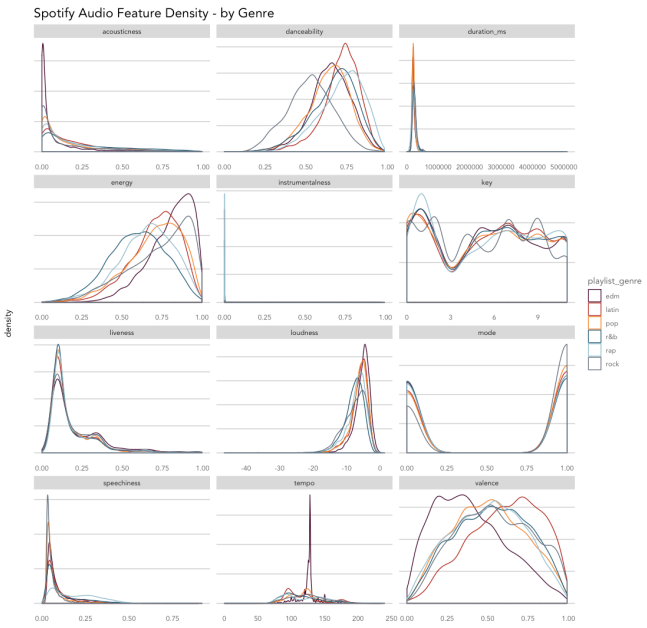

Figure 2: Spotify Audio Feature Density - by Genre

5.2 Genre Detection: ML Classifier vs Shazam-Spotify workflow

We initially explored an ML genre classifier to facilitate 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, and Convolution Neural Networks (CNN), and K-nearest neighbors (KNN), was around $60\% - 70\%$ making it less reliable, and increasing the uncertainty when we combine it with the rest of the workflow.

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. 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.

Spotify's audio feature data frame consists of 12 features. From an R Studio analysis of genres on Spotify, and their audio features, we saw that there is maximum variability among 6 of these 12 features across different genres: Danceability, Valence, Liveness, Loudness, Tempo, and Energy. We went over a description of the features in the previous Feature Query section

We extract these features for a particular audio, and we want to allow users to adjust these slider values and accordingly visualize changes in the lights.

5.3 I/O Communication With Lights: PyDMX vs DmxPy

In order 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 need given that our logic will be 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 Audio Chunk Size: Latency vs Feature extraction

We are in the process of exploring an optimal chunk size. To choose an ideal chunk size, we are considering two different factors: the end-end latency to trigger lighting, and the maximum feature extraction possible from an audio clipping.

Our chunk size affects lighting changes because we want to be able to enact lighting calls in real-time. We want to process chunks in such a way that each chunk takes minimal time to process and pass to the Expressive Lighting Engine (ELE). Currently, we are testing with a 1000ms chunk size to even the Shazam API, and if we get the same song title with two chunks, we should be able to confirm the song easily. We might need to explore different chunk sizes for genre detection and signal processing engines as they have different use cases. Genre detection requires a quick way to confirm the song is the right one by comparing two chunks. Signal Processing requires us to extract timbral, pitch, and rhythmic features from each chunk and transform them into lighting changes.

$$\textbf{\textit{Chunk Latency}} = \frac{\textit{Buffer size}}{\textit{Sampling rate (Hz or cycles per second)}}$$
$$= \frac{256}{44100}$$
$$= 0.0058 \text{ or } 5.8 \text{ ms}$$

For simple calculation purposes, we are going to use a live sampling rate of 44.1KHz. Typical buffer or chunk sizes range from 128 to 256 for processing purposes. Low buffer sizes help us handle data quicker, which results in an increased demand for processing power. We can start

with a buffer size of 256, and see if we can scale up our compute power to handle a smaller buffer/chunk size. A smaller value for latency does not affect sound quality but might require quicker processing.

5.5 Input to Lighting Mapping: Deterministic or Probabilistic based on Elimination

One of the biggest challenges of this project is to map auditory inputs to specific lighting function calls. We decided that there were two routes we could've gone. The first would have been to discretize the input space and have each of the input partitions map directly to a specific lighting call. The second was to treat the entire set of function calls as an output space, and use the auditory input to narrow down the input space. We would then use random number generated inputs to select an output from the reasonable options.

We found the second approach a lot more interesting because it would leave a bit of variability in the actual output of the show allowing for more dynamic and interesting lighting. Also, we reasoned that there are clear types of lighting that would not be acceptable given certain audio variations, however, there is not really any specific lighting that is optimal given audio i.e., how optimal the lighting is depends on interpretation, and for the purpose of this project we will minimize subjective choices and rather focus on an accessible solution that avoids pre-programming lights.

6 SYSTEM IMPLEMENTATION

Our main subsystems are the Show class, Feature Query, User Interface (UI), Signal Processing, and Lighting Engine. Our current implementations with respect to the Lighting Engine, and Feature Query can be seen in our github repository: https://github.com/aasuper1/Synesthesia

6.1 Show

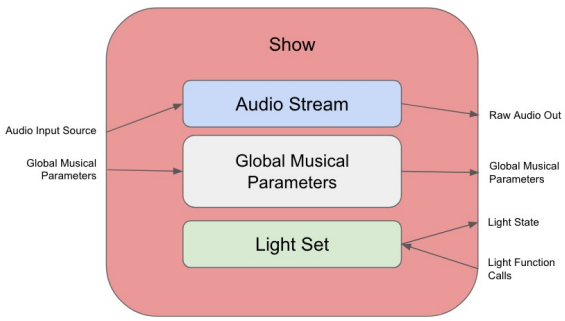

Figure 3: Show subsystem

The Show subsystem will house the Audio Stream, the Global Musical Parameters and the Light Set. The Audio Stream will have a variable storing the input source. It will then use Librosa to stream the audio. We will make sure, for concurrency reasons, that the audio stream is read only so that other subsystems can look at the values of the stream without causing hazards. The global musical parameters will be stored in an array of size 6 with the following values: valence, liveness, loudness, tempo, energy, and danceability.

The Light Set will be an object of the LightSet class. It will include a list of all the hardware lights that are available. These lights will be represented by the Light class. There are currently four types of Light subclasses. There is Par, Derby, Laser, and Strobe. There will be one Light object in the LightSet corresponding to a unique hardware light. From the Light classes that are available the user can create LightGroups. These are groups of lights that will function in a synchronized manner. There are three types of LightGroups: UnionGroup, MirrorGroup, and LineGroup. UnionGroup will enable all lights in the group to function in unison. MirrorGroup will enable lights to mirror each other's movements. A LineGroup will enable lights to create patterns across multiple lights. Each of these LightGroups will contain a function list which allows the Lighting Engine to determine a list of viable lighting calls.

6.2 Feature Query

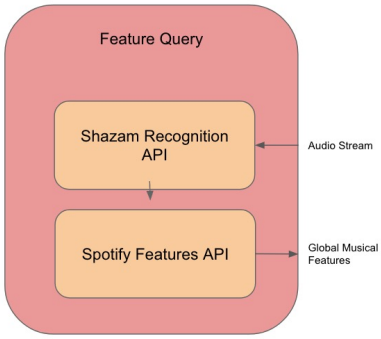

Figure 4: Feature Query Subsystem

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 is chunked into smaller bits to allow lower latency through the different engines. 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 a million parameters about the song that it identified. In a continuous loop, we compare chunk x to chunk x-1, and if we identified the same song in both chunks we can exit out of the song detection phase for the time being as we can confirm that we have received the right song.

However, we do need to send a heartbeat every now and then to the audio data to detect any musical changes. If there is a musical change detected, we call the song detection API once more, and get the current song we are in.

Using the title of the song, we then poll the Spotify API with a fixed set of client credentials, and secret key. 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. If it errors in either one of these steps, we can alert the user to try a new song, or have certain fail safe defaults that they can listen to their audio clip with. We query Spotify with the song title using the track uri because that's a unique identifier, and we get back a dataframe with a bunch of audio features and data pertaining to the song. From this nested dictionary or dataframe, 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.

Additionally, we have also used the GeniusLyrics API to extract the lyrical content of the song we polled from Spotify using the title we got from the Shazam API. Using the NLTK library, we can do a simple sentiment analysis to determine positive, neutral or negative values the lyrics suggest, and this is an additional parameter that is passed into our engines to allow for the artistic flourish in lighting choices.

6.3 User Interface and Manual Overrides

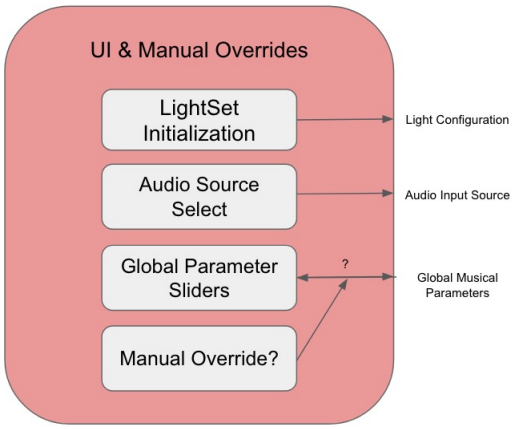

Figure 5: UI Subsystem

The UI is broadly divided into three major components: light identification and light set selection, audio source selection, and global parameter output and slider value modification. These components are implemented using a Python-Django web-app. Although in a real world system, the web-app will be hosted on a server or would use a native application, for the constraints of this project, it will be hosted locally. This will allow us to minimize latency in a small-scale environment as the application is designed to support individual performers. Further, the web application will follow a Model-View-Controller (MVC) design protocol.

The first component is the light identification. To achieve this the web application defines a model for the lights. This model communicates with the I/O using Show subsystem, and looks for a signal from the USB ports for a lighting accessory to be connected. At our scale, once the lights are connected, they will be recognized from an exhaustive subset. Here, users will be able to group the lights creating new Light Sets of their choice, specify the channels they would like to access, and determine the daisy-chaining of the lights. However, as we are only using a singular gigbar for the project, we will have a preset to save setup time. This initialization will be communicated to the ELE subsystem through Show and Show will send packets back to the web-app for the lighting queue.

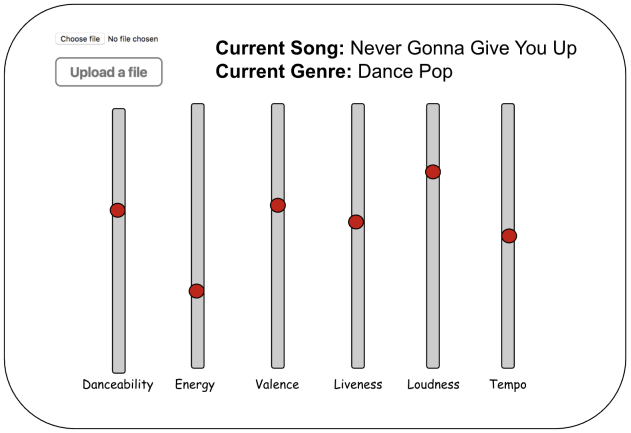

Figure 6: Mockup of the UI main page

Figure 6 (above) shows a mockup of the second and the third components of the UI. For the audio source selection, the UI will have access to the local directories on the computer. Here, the user will be able to select .mp3 or .wav files, and this constraint is enforced within the input model on the webapp. Once the audio file is selected, the user will be able to play/pause/replace the file using HTML form buttons on the interface. These buttons will send POST requests to the backend to trigger said actions. Once the file is uploaded, the data will be first relayed to the Feature Query subsystem via Show, which will respond back with the song title, genre, and values for the global parameters. These outputs will be processed in the web-app views to update the element reference tags on the interface. Simultaneously, the audio will also be sent to the Signal Processing subsystem via Show, which will start sending back a continuous data stream for the outputs.

Finally, the last component of the UI is the CSS sliders for global parameters. These parameters will default to zero values, until they are communicated from the previous component. Then using a reference tag again, these values will be updated to refer to the processed global parame-

ters. Further, the slider object will consist of a button, which will look for mouse inputs. If the button is dragged, these changes will be communicated to the backend 5 seconds later. This is to allow enough time for the user to change all the global parameters. These sliders will thus act both as an input source from the user, and an output from the Feature Query subsystem.

6.4 Signal Processing

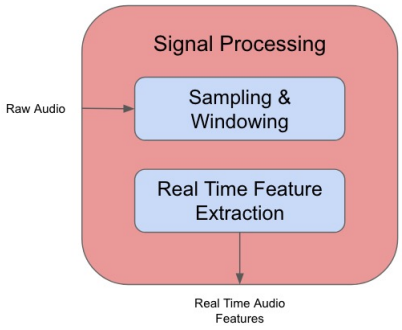

Figure 7: Signal Processing Subsystem

The Signal Processing subsystem consists of two major parts: Sampling, and Windowing, and Feature extraction. This process starts with normalizing the audio that helps us to remove background noises, and unwanted frequencies. This is followed by segmentation that allows us to break the audio signals into windows. These windows or chunks in our case help us continuously process audio signals and reduce lag. It allows us to make comparisons between two chunks, and make appropriate changes from the current state as well.

Using the Python library, Librosa, we first take a brief overview of the audio after loading the file in an mp3/wav format. We get the audio length, and the total samples. We use a default audio sampling rate of 44.1KHz. We separate the audio samples into harmonics, and percussive signals to allow us to see which parts of the audio include both aspects, a single aspect. This dictates how powerful the light is going to be for a certain chunk of time. Along with the percussive and harmonic split, we also want to isolate beats specifically. In Librosa, beat extraction uses a hop_length (number of samples between two windows) of 512, and it centers the frames so that the kth frame is centered around a sample k, making it easier to measure the peaks in a single onset strength (Onset strength is done by measuring the frame-based increase in energy in a window). This is relevant for determining the timing for flashes, and the intensity of light we want to trigger.

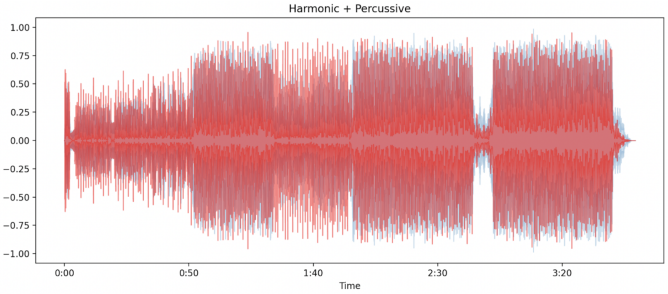

Figure 8: Separation of percussive and harmonic frames for a snippet of Teenage Dream

As explained in the Signal Processing Attributes section, we also want to take the Mel scale cepstral coefficients, the spectral centroid, spectral rolloff, and the zero-crossing rate features for each audio chunk. This way we compute coefficients from each of these metrics, and construct a dataframe for each chunk. This is sent into the lighting engine to determine what changes can be made to the current state.

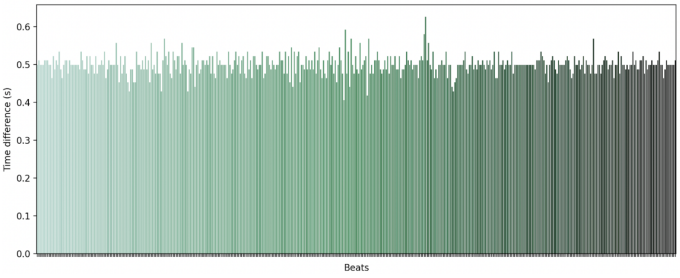

Figure 9: Beat extraction for a snippet of Teenage Dream

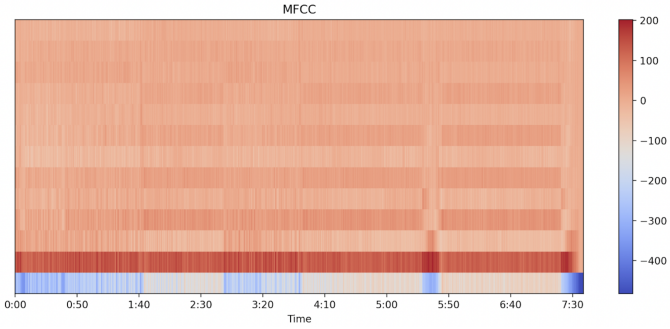

Figure 10: MFCC analysis for a snippet of Teenage Dream

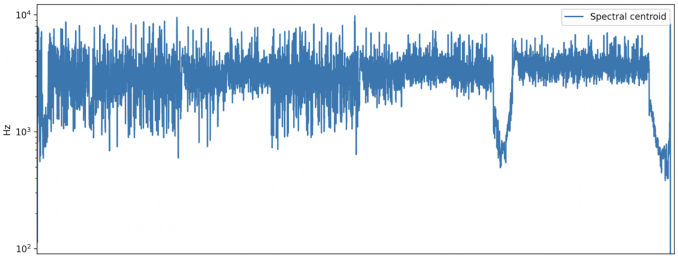

Figure 11: Zero crossings for a snippet of teenage dream

6.5 Lighting Engine

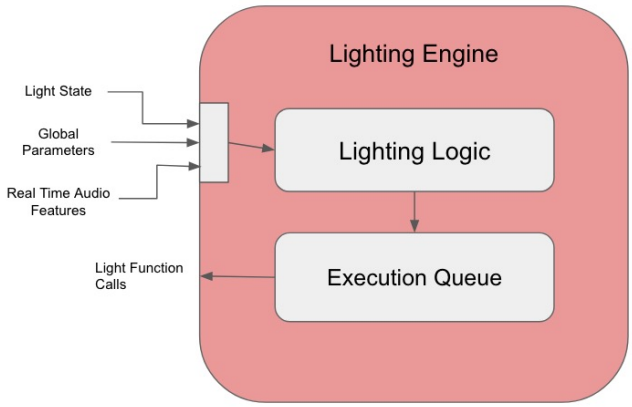

Figure 12: Lighting Engine Subsystem

The Lighting Engine is composed of two main parts: the Lighting Logic, and the Lighting Execution Queue. The Lighting Logic is responsible for determining the light call that needs to be made, and the Lighting Execution Queue is responsible for sending the requests to the LightSet to actually change channel values.

With regards to the logic, there are a couple main components: timing, and selection. Timing will determine a trigger for when to actually execute some lighting call. This will be determined mainly by the real time audio features that are coming from the signal processing unit. There will be beat detection, and based on beat detection and the energy of the piece we will call functions twice a beat, once a beat, every other beat or even less frequently. Once we have decided that we are going to change the lights, we have to decide what function to call. The first step of this process is to pick an available LightGroup. The LightGroups will be available if none of their member lights are currently executing any lighting commands. Once a LightGroup is chosen, a function will be chosen from the available LightGroup functions. Finally the parameters for that function will be set. Each of these decisions will be influenced by the global musical parameters and the real time audio processing values.

The Lighting Execution Queue will then take these function calls and place them in a queue that will handle any concurrency. This queue will speak directly with the LightSet and call the respective functions.

7 TEST & VALIDATION

7.1 Test for Latency

We broke latency into many individual parts into the chunk processing time, audio feature processing, and time through the lighting engine in Design Requirements: B. Latency section. We want to first record the difference in timestamps as audio is uploaded into the UI, and as lighting functions are executed. According to our calculations, we can find the breakdown per system, and robustly test the latency of each subsystem. This way we can assess bottlenecks, and also find out where we can reduce time or be more flexible. We want to have an upper bound of 185ms for all of this processing.

7.2 Tests for Setup Time

We can test for setup time for the system with focus groups of performers. If the current setup time for performers is 60 minutes, we want to be able to reduce it to ¡5 mins of the setup time to significantly free up time for them. Setup time accounts for loading the system, and uploading the compiled tracks. A focus group of 5 diverse performers will help us evaluate whether our system is customizable and adaptive enough for different kinds of performers. If setup times for different performers are different, we should aim for reducing their time to less than 1/10th of the time.

7.3 Tests for Signal Processing Efficiency

We can test for signal processing efficiency by evaluating if we can get more than 90% of the auditory features as compared to hand labeling a file. The reference point is a hand-labeled file with a waveform and looking at all the features. We should also be able to handle a live sampling rate of 44.1KHz, and a chunk size of 256 through our signal processing systems.

7.4 Tests for Light Response

We have already achieved the metrics for light response tests. This includes being able to control the lights through a python library without the QLC+ interface. We were able to successfully achieve this using the DmxPy controller.

7.5 Tests for Multiple Outputs

We have already achieved the metrics for multiple outputs. This includes being able to control multiple lights using the DmxPy controller. Through the controller, we are able to control lights independently, and in unison by making calls to different lights using their unique address pins.

7.6 Tests for Manual Adjustments

We can achieve metrics for manual adjustments by limiting adjustments to less than 3 per minute per song. We use 3 as a benchmark because this is the most adjustments musical performers like DJs will have to make per minute as they shuffle between different song tracks. We can test for the manual adjustments metric on a sample audio with different genres, and measure the adjustments performers will have to make.

7.7 Tests for Overall Appeal

Using focus groups, we want to assess how dynamic our system is, and how interesting and accurate it is for an observer.

1. We can use a focus group of 5 people who visualize the light show without the audio. We should be able to check if the focus group is able to detect deflections in genres and recognize 'vibe' changes
2. The second focus group consists of performers. We want to evaluate their overall satisfaction levels with the system, and whether they are likely to use our new dynamic system as opposed to their less customizable lighting equipment.
3. Lastly, we want to assess the holistic appeal of the music from an audience's perspective. We want to see whether they are happy with the dynamic light changes, and nuanced reflections of the audio.

8 PROJECT MANAGEMENT

8.1 Schedule

Our project is ambitious and involves the development and integration of multiple subsystems that include audio processing, communicating with lights, a modifiable user interface, and feature extraction. In order to achieve this we have set up an aggressive timeline, and have tried to get a headstart on the development and testing, as seen in the Gantt Chart in Figure 13 at the end.

8.2 Team Member Responsibilities

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

- Abhishek is responsible for the lighting logic and communication with the hardware
- Parth is responsible for the modifiable user interface and communication between the audio and the lights
- Rachana is responsible for the audio processing and song feature extraction using various APIs All members will work on the integration of the subsystems and the overall testing of the 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 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 are borrowed from the CMU IDeATe lab. The only additional hardware is a personal computer to run the program on.

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

Table 3: Breakdown of Expenses

8.4 Risk Mitigation Plans

The primary risk for the project was efficient communication with the lights. However, given the current status of our project, we are already past the point of early risk evaluation. We have produced various proof of concept demos to show the viability of our different subsystems, which include demos for controlling different functionality of lights with our program, using signal processing to extract key audio features, and allowing user input and modification. As of the current state of the project, we have the following risks and risk mitigation plans in place.

8.4.1 Latency and Fail-Safe Defaults

Through our basic signal processing tests we have determined that windowing the audio signal is the only viable option to get meaningful data. However, finding the optimal window lengths comes with a set of challenges. If the windows are too long, we run into latency issues as there is a lag between when the audio is played as compared to when the lights react to it. On the other hand, if the windows are too small, there is a risk of misclassification of attributes, which can make the lights behave in non-deterministic ways.

To combat this, we have incorporated a lighting queue in our design that can store a set of lighting instructions. This allows us to look further ahead into the signal without any loss of data and with lesser impact on the latency. Further, we have established fail-safe defaults in case the audio processing fails. This means that the user may bypass the system and simply use the global parameters to control the lights, in the event that the signal processing system becomes unresponsive. These design choices have been made to minimize the need for user input and to prevent any abrupt or non-deterministic changes in the behavior of the lights.

8.4.2 Scaling and Independent Operation of Lights

Due to limited time and resources, we might need to further constrain the scope of our project. Given we are trying to build a system that is able to communicate with 4 different types of lights at any point in time, the probability of facing scaling issues is non-trivial. While we aim to develop a system that is able to communicate with the four pairs of lights both independently and in sync, this can pose a risk to the overall coherence and completion of the project.

To prevent this from happening, we have been following a top-down development approach, where we implement and test small deliverables every week. This is done to track our progress and work as an indicator of the need to rescope. Further, we are following an ambitious development timeline to ensure that we have enough time for thorough real-world testing.

9 RELATED WORK

There are several different systems developed to tackle the issue of light customizability for musical performers. There are dynamic lighting systems for fully reactive light-art installations by Protopixel [6]. Another type of system explores mood lighting that expresses emotions elicited through the audio. [7] Here the system examines the emotional impact of music or color, and this can be maximized if they are used together. The article presents a mood lighting system that automatically detects the mood of the piece of music and expresses the mood through synchronized lighting.

We explored the real-time element through the reference for Dynamic lighting to reflect tension in gaming [8]. We were inspired to look at a feasible Expressive Lighting Engine (ELE) through this since games similar to performances simulate elements of traditional media such as plot, characters, sound, and music, lighting. The interactive experience aspect of games also presents a host of player challenges similar to audience-performer interactions.

10 SUMMARY

We hope to create a program that is capable of producing automated light shows in real-time. We will do so by processing audio and extracting features that can be used to control the behavior of multiple lights. Using an interface to allow for user inputs and modifications, this project aims to enable small performers to create seamless light shows with minimal setup time and equipment.

By creating classes for lights, and translating auditory features to lighting logic, we are able to create systems that are modular, testable, and scalable. However, standardizing lighting equipment is beyond our scope. This project will thus demonstrate the capabilities of such a system at a smaller scale, with four light pairs.

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. Concert Lighting: The Art and Business of Entertainment Lighting - James L Moody, Paul Dexter
2. https://ennuimagazine.com/how-long-does-it-take-to-set-up-a-stage-for-a-concert/
3. https://www.musicindustryhowto.com/best-dmx-stage-lighting-control-software/
4. https://support.biamp.com/Tesira/Video/Video_and_network_latency
5. "Real-Time Audio Effects on Mobile Devices using GPUs" (published in the Journal of Signal Processing Systems in 2018)
6. https://www.protopixel.io/showcase/events/chagall-unlocked-lighting-performance
7. Mood lighting system reflecting music mood : Chang Bae Moon, HyunSoo Kim, Dong Won Lee, Byeong Man Kim
8. Dynamic Lighting for Tension in Games: Seif El-Nasr, Magy, Niedenthal, Simon, Kenz, Igor, Almeida, Priya, Zupko, Joseph

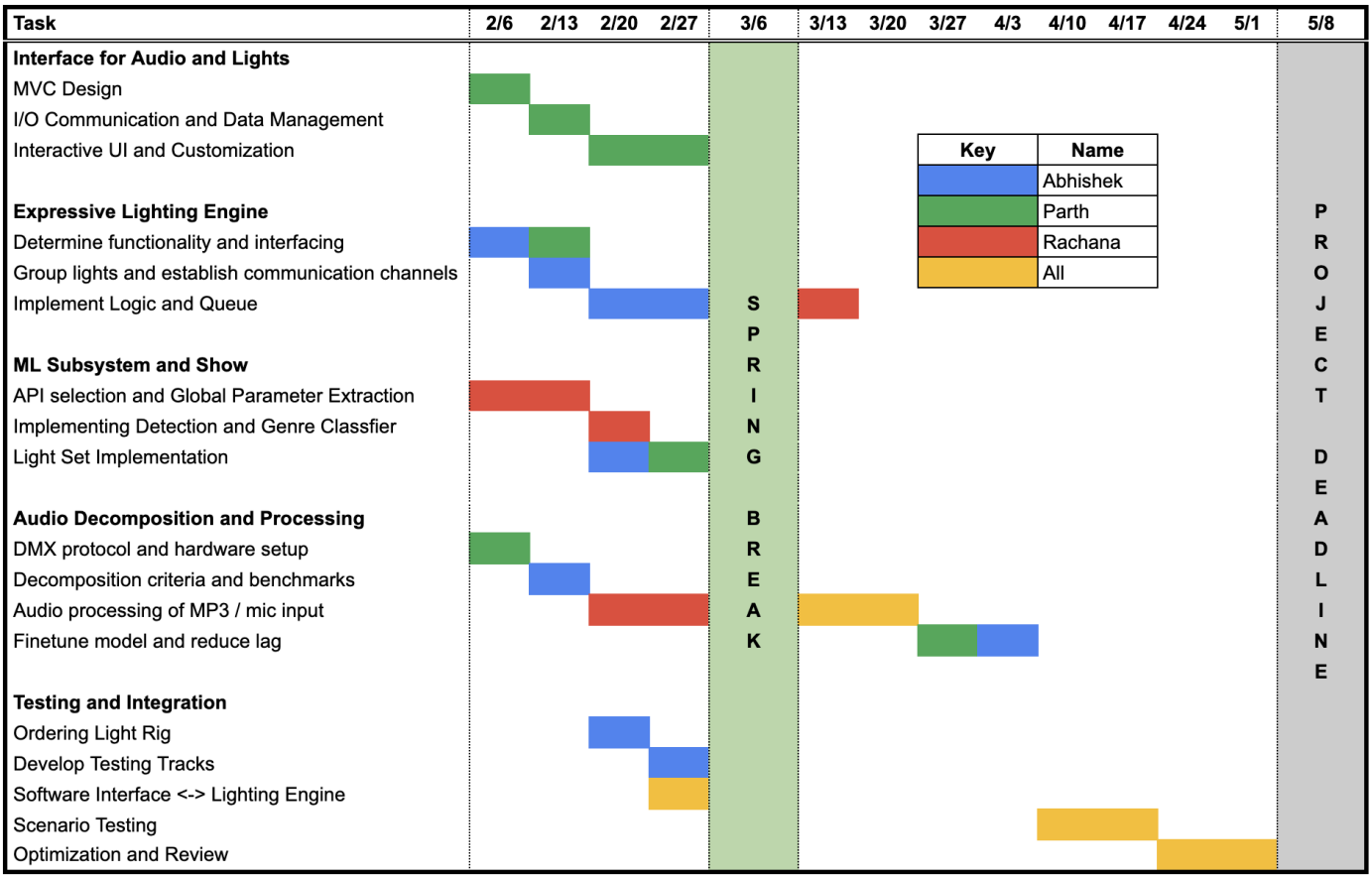

Figure 13: Tentative Task Breakdown and Schedule