LightShow: Abstract Representations of Music Lighting In Python

by

Benton B. Wilson

B.S., Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author .....
Department of Electrical Engineering and Computer Science
August 5, 2022

Certified by .....
Joseph A. Paradiso
Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

LightShow: Abstract Representations of Music Lighting In Python

by

Benton B. Wilson

Submitted to the Department of Electrical Engineering and Computer Science
on August 5, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis explores music lighting and ways in which music lighting can be generated automatically. Using videos of prior concerts as training data for a machine learning model was explored, but ultimately proved unsuccessful. However, a useful abstraction for representing, designing, and implementing light shows based on audio is designed, implemented in Python, and used to generate lighting in a few contexts. The abstraction designed in this thesis ultimately focuses on allowing developers to easily expand on the package and reuse code, with the restriction that audio data must be known ahead of time (the abstraction does not support live audio streams, but the future work section outlines how this could be implemented).

Thesis Supervisor: Joseph A. Paradiso
Title: Professor of Media Arts and Sciences

Acknowledgments

Throughout the process of pursuing this project, I have received support from many different people. First and foremost, I would like to thank my advisor, Joe Paradiso, for providing guidance, inspiration, and encouragement. Joe has done a phenomenal job in helping me hone my idea, which initially started as "something to do with music and visuals," into something much more specific and useful. I am grateful for the time he has more than willingly spent helping me.

I would also like to thank the members of the Responsive Environments research group for being so open and willing to provide insight and to share their knowledge. Specifically, thank you to David Ramsay for numerous extended brainstorming sessions and for sharing knowledge on the entire thesis writing process. Also, thank you to Ishwarya Ananthabhotla for helping me figure out which questions might be the most interesting to try and answer.

I would also like to thank Zach Sternberg for being an amazing source of knowledge regarding industry level music lighting.

Finally, I would like to thank my family and friends for supporting me throughout this process, and for helping make my time at MIT so enriching.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Lighting

In music performances, lighting often plays an integral role in setting or complementing the mood and energy of a performance. Recently, light shows have become increasingly complex and often require many hours of manual labor to choose how to position light fixtures, time them to the music, as well as make sure that various aspects of the lighting, such as hue, saturation, and intensity properly supplement the performance. Work has been done to streamline these decisions, and there are various tools which help automate certain aspects of creation, but oftentimes these tools specialize in a specific aspect of lighting, and still require artists to spend extensive amounts of time working on choosing exactly how to design their light show.

While this may be reasonable for some artists, there are many groups on a tighter budget that cannot afford to spend so many hours designing and manipulating their lighting for a live show. Thus, it would be helpful if there was a simple way for an artist to have a lighting setup and be able to automatically generate a dynamic light show that responds to a musical performance, taking various contexts into account.

However, in order for this goal to be achieved, the barrier to entry for representing and coding light shows needs to be lowered. Thus, creating a framework that allows developers to easily explore and create light shows in order to marry emerging music information retrieval (MIR) technologies would be beneficial. In the following paper,

we present the LightShow package, implemented in Python, which allows developers to design light shows at a high level of abstraction, optionally taking into account where lights are located in space. The package is currently designed to handle audio that is known ahead of time, but the future work section outlines how this could be fixed to allow a stream of audio to generate a stream of lighting outputs. Full source code for the package can be viewed at `https://github.com/bentonw414/lightshow`.

Chapter 2

Background

In the world of music production lighting the common specification used by lighting systems for carrying information is known as Digital Multiplex, or DMX for short. In DMX, there are 512 "channels" and each channel contains a value in the range 0-255. Lighting fixtures are attached serially to each other, and channels are assigned to control various aspects of the light for each lighting fixture (a lighting fixture here is simply a group of lights). Then, a DMX controller sends packets of information through a cable, controlling values such as the amount of red, green, blue, brightness, or even position for each fixture.

However, this representation of light, while in a way abstract, does not really capture how we think about light shows. When we watch a light show, there are events that start and stop, and the mood may match the overall hue of the light. We don't see a light show as simply a stream of colors, but rather there are higher level associations with those values across time, and those high level associations are what are important. For instance, if a light is off for a period of time, then gets really bright, and over a short period time decreases in brightness until it is off again, we may perceive this as a "flash," which is a bit more abstract. While the end result is the same (there is a stream of color coming from a light over time), having a way to capture the higher level semantics of "a flash" is more natural for the designer of a show.

2.1 Previous Work

2.1.1 Relationship Between Music and Color

There has been extensive work done examining the relationship between music and color. One common theory is that music affects the emotion of a listener, and that there is a mapping between emotion and color. Since there are many different ways to classify emotion, various models have been used for finding such a mapping between music and emotion, but there have been some general trends that have been found across such mappings.

Across cultures, people tend to associate brighter, more saturated, yellower hues with music that is faster in tempo, or music that is in a major key, and they tend to associate darker, less saturated, bluer hues with music that is slower in tempo or in a minor key [11]. While this association is clear, it is relatively coarse, and does not necessarily paint a complete picture of how music and color are related.

2.1.2 Visualizing Music

Creating visualizations for music has been an area of research for some time, and much work has been done to make visualizations more expressive and easier to create. Earlier work involved simpler mappings from music to various shapes and expressions, like modulated spirals, and there have been relationships mapping darker and lighter colors to associated emotions for a viewer [3]. Work has also been done to help make visuals more easily, using programs like Imager with Sonnet, and later Jitter, which enable programmers to build and connect components to map audio inputs to visual outputs [2][6].

Work has also been done to help make creating light shows easier to do. For instance, software such as MagicQ [8] can help control lights from a laptop, while programs such as TouchDesigner [17] and Resolume [13] help abstract some of the lighting design away from a simple stream of values. However, each of these pieces of software specialize in a specific piece of the lighting pipeline, and often require time

to get up to speed on using the tools available within them.

2.1.3 Automatically Generated Music Lighting

MIR as a field has grown significantly in the past decade, and as a result, there are many libraries that have been created to make extracting information from an audio source much easier. One such library, implemented in Python, is librosa, which allows programmers to easily perform various analyses of audio signals [9]. On top of this, music emotion retrieval has also grown as a field, and many of the techniques employed in this field are extremely useful for extracting emotional information to help map music to visuals.

Work has also been done in the field of automatic music transcription, the process of converting a music signal into some sort of musical notation. One such package, named "Omnizart," was released in 2020 and is implemented as both a Python package and a command line tool, and allows the user to transcribe audio into midi files [18]. Other similar packages exist for annotation of music (such as aubio [1]), but another area where music data can be found is through public APIs. For instance, Spotify provides a extensive metadata for each song through the Spotify API [14], and there are packages in Python (namely Spotipy [15]) that make using said API extremely easy. All this metadata could be used to map to a light show, given that we define a clear mapping.

Attempts have been made to quantify the relationship between music, emotion, and color. In a 2016 study, a group of researchers quantified the emotion of a piece of music by using the Thayer model of human emotion, which places emotion as a point in a plane of 2 dimensions [5]. One dimension is valence, how positive or negative an emotion is. The other is arousal, how energetic a certain emotion is. In this study, researchers asked lighting engineers to create a simple lighting show for a short piece of music, adjusting only the properties of a single light in order to match the music. Then, the researchers split the music into its emotion as well as its intensity. They found that the lighting artists often associated certain colors with certain emotions under the Thayer model (depending on the music genre) [5]. They were then able to

automatically map music to a color by first mapping the music to the emotion space, and then mapping the emotion space to a color. However, they chose to associate the brightness of the lighting with the intensity of the music (so those two aspects of the light and music were separated during analysis).

2.1.4 Computer Vision and Lighting Detection

The world of computer vision has seen many advances. One popular open source package that has been around for a while is OpenCV, which is available in both Python and C++ and can be useful for its common helper functions and for running general purpose computer vision tasks.

Fast object recognition in images and videos is also an active area of research. One technique for processing images, known as YOLO (short for You Only Look Once), was invented in 2016, and greatly increased the efficiency with which computers are able to recognize objects by using a single pass through a neural network to recognize objects (whereas other techniques used multiple passes) [12]. This is now one of the most commonly used methods for live object recognition.

Furthermore, work has been done to automatically reconstruct a lighting environment for a scene, given only an image to work with (this has become increasingly important as augmented reality has grown as a field of research). Researchers have employed various techniques, from using neural networks to estimate the direction of lighting in an image [7], to estimating all sources of light in an indoor environment [4]. Since music performances are often shot from multiple cameras, this could be useful for determining the relative positions of different camera shots in a video.

Chapter 3

Initial Work to Arrive at LightShow Package

Initially, the goal of this thesis was more focused on automating the process of generating visuals. Particularly, since there are plenty of videos online of concerts and other performances with lighting that is matched to an audio signal, we had an idea that those online videos could be used as training data for a model.

3.1 Initial Attempts

The general pipeline that we used for automatic lights to audio was the following. First, the raw video frames processed to generate lighting features, and the audio was processed to generate audio features. Then a model could be trained to take in the audio features as the input, and use the video features as the target in a supervised learning model.

For audio features, it generally made sense to use what was already available (the common approach to audio classification is to use the audio to create a Mel Spectrogram). However, extracting the video features proved to be much more difficult, as many of the techniques mentioned in 2.1.4 for extracting the lighting information from a video are relatively coarse, requiring specific types of videos to work, since the techniques are still evolving.

3.2 Isolating the Stage

One problem with videos shot of performances is that the camera angle often moves, and when concerts are professionally filmed, there are often multiple cameras that a video cuts between. As such, one problem when extracting information from a concert video is in making sure that we can roughly identify where the stage is located in the video, in order to discard information that is not from the stage. After some experimenting, one algorithm that appeared to work was to use a sliding window average algorithm as follows:

1. For each frame of the video, find the largest area rectangle on the frame, such that 90% of the total energy in the frame is contained within the bounding box. Energy E is defined per pixel of a grayscale image in the following way, where $I(i, j)$ is the pixel value at location (i, j), and $\mu(I)$ is the mean of all pixel values in the image:

$$E(i,j) = \max \begin{cases} I(i,j) - \mu(I) \\ 0 \end{cases}$$

2. Once all the bounding boxes are labeled, we take the average bounding box (mean of each point from each of the last 10 frames). This helps to smooth out the data, since camera angles and cuts are much less common by frame than continuous shots.

Results from one small clip can be viewed in Figure 3-1, and while the algorithm works okay in this clip, it struggles with videos that do not take place in a dark setting, or with camera angles that are not straight on the stage (for instance, when the stage is viewed from the side, a simple bounding box cannot really describe the location of the stage).

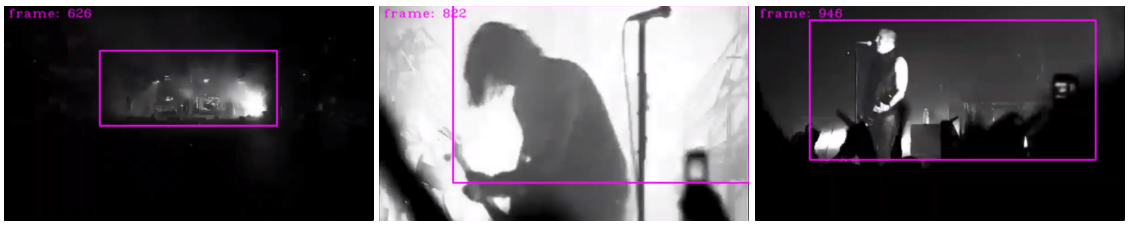


Figure 3-1: Example results from running the algorithm for isolating the stage on a video. Video can be viewed at `https://www.youtube.com/watch?v=hKmYnRQlg0g`

3.3 Manual Annotations

After working on isolating the stage in a camera shot, further extracting video features still proved unsuccessful, so instead of automatically extracting lighting data, a simple user interface was built to annotate videos manually. The user was given a simple representation of 36 stage lights, and then had the ability to add lighting events of either a fade from one color to another, a strobe between two colors, a constant color, or a flash of a single light color. Multiple events could happen at once (for instance, there could be strobing from above the stage, at the same time as lights on the side were fading from blue to green).

Manual annotation had the problem of being both slow and tedious. Oftentimes, the simple levels of abstraction of fade, constant, flash, strobe did not really capture what was happening (for instance, flashing with the beat), and it became clear that a richer level of abstraction for representing a light show would be much more useful. Hence, the LightShow package.

Chapter 4

LightShow Package

4.1 General Goals

Coming into this thesis, the high level goal was to use videos of performances as training data for developing a model to automatically generate a LightShow. However, it quickly became apparent that a direct mapping from audio to lighting, while intriguing, would not be as flexible as having a solid abstract representation of a light show. In other words, if a model can output something of the form "a 200ms green light on every bass note during the chorus," this is much more adjustable from an artist's viewpoint than a simple stream of color values. So while some attempts were made to extract information from a video to use as training data, it became clear that having a nice abstract representation of a light show would be immensely useful.

When coming up the core abstraction, the idea was to keep the flow of information as simple as possible, since that makes it easier to understand, use, and expand, and also helps make it easier to get started using the package. The overall flow of information in the package can be viewed in Figure 4-1, and the package itself is discussed in more detail in Section 4.3.

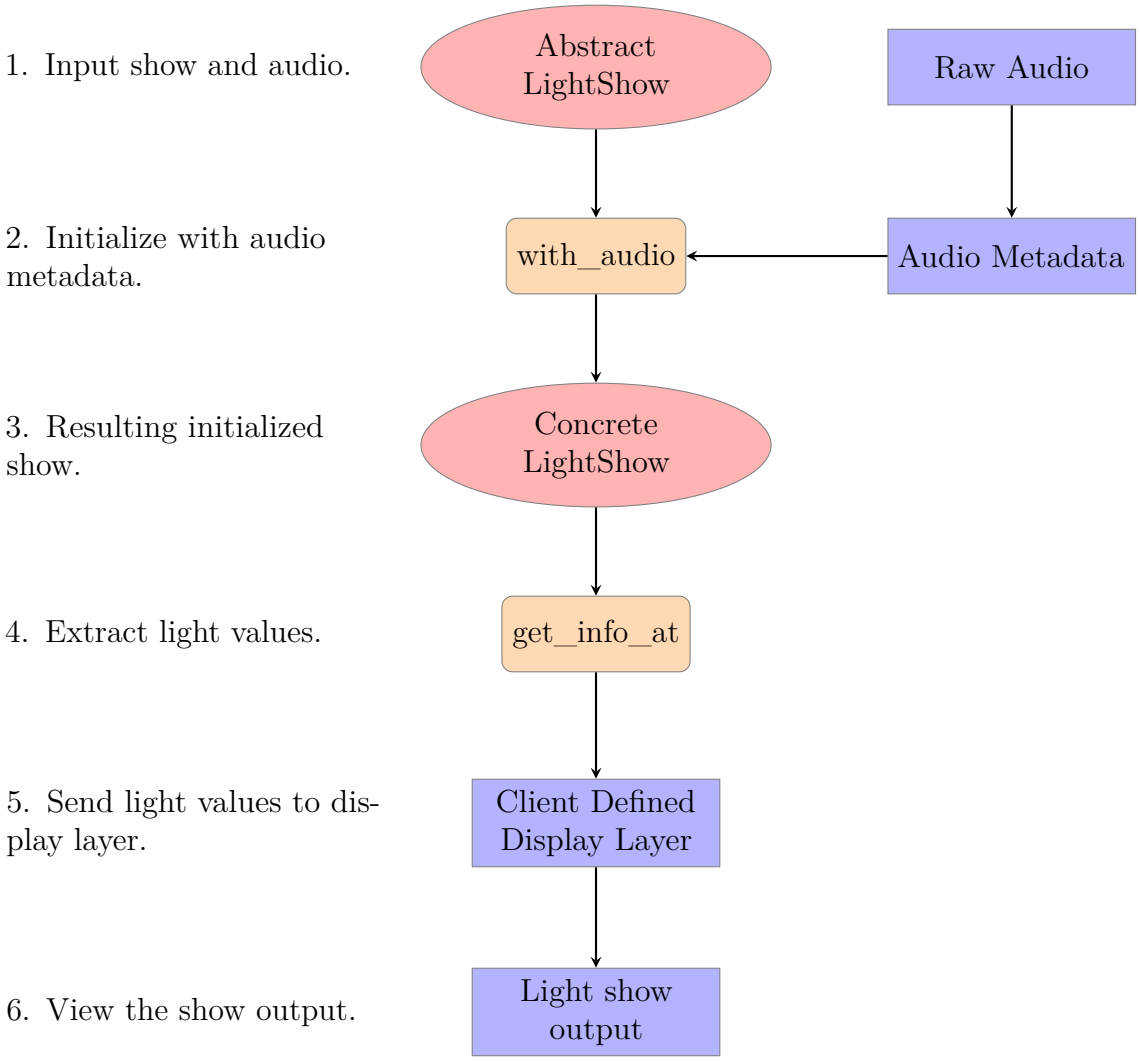


Figure 4-1: General Flow of LightShow package

4.2 Core Design Goals

The following section discusses motivating factors and goals and how they impacted the design of the LightShow package. Among these goals are: easy integration with old as well as new MIR technologies, an emphasis on re-usability, expand-ability, simplicity, as well as the ability to both encode, or not encode spatial geometry into the abstraction.

Easy to Integrate with MIR Technologies

The LightShow package in general should be easy to integrate with new MIR technologies. As such, the core levels of abstraction do not require any sort of specific audio metadata, and it is ultimately up to the implementer what metadata is used in a specific abstraction. For instance, there is extensive use of callback functions in the code, and the client can choose both the input to the callback function, and well as the implementation of the callback function. An example of this is the `DynamicAtEvents` class, which takes in a callback function of any metadata that must map to a list of timestamps to play a LightShow on. Then, later, the client can pass in whatever metadata is expected by the callback function. Since the user can choose the callback function, this means that any metadata could be passed in and used.

Thus, metadata could come as raw audio, midi files, algorithms using the librosa library, public API calls to Spotify, or any other information that the client wants to use in an abstraction.

Reusable

LightShow objects and abstractions should be as reusable as possible. One way to achieve this in a direct sense is by ensuring that all LightShows are immutable, meaning that they can be passed around and shared, and generally LightShow objects can be reused without needing to worry. For instance, if we have a show that we want to play on every beat, we can just reuse that show at every beat, without needing to worry about the underlying LightShow object changing.

Simple and Easy to Understand

While LightShows being immutable eliminates a large class of aliasing bugs, keeping the API simple and intuitive also allows for easier programming. LightShows only really have 2 core methods, namely `get_info_at` and `with_audio`, plus simple information such as `start`, `end`, and `length`, and all other abstractions are built around those two core functions.

Also, as shown in the examples (see `on_midi_beats_1` and later examples in 6.4), once an abstraction is created, it can be easily reused and combined with different LightShows to create more intricate LightShows. Also, once an abstraction is created, it is very easy to quickly represent various light shows using that abstraction (for instance, the code in the example for `on_midi_beats_5` in 6.4.5 is relatively compact).

Both Geometry Aware and Unaware

Initially, there was no notion of spatial geometry encoded into the LightShow package, but since dealing strictly with light indices can be cumbersome, it made sense to include support for geometry. For instance, what if there are a strip of lights, and we want to have color wash from the left to the right? Or, in a more complicated scenario, what if we have many strips, all oriented at different angles, and we still want color to wash over all of the strips from left to right? Or, if we don't care about the specific geometry, and we just want color to wash from the beginning of a strip of lights to the end, regardless of how we position them in space?

As a result, there are also `Shape`s and `LightingComponent`s, which can be used to optionally encode spatial geometry into a LightShow. For instance, we can use the `Sphere` shape to color a certain part of a `LightingComponent`, and since `Shapes` and `LightingComponent`s can be encoded in 1, 2, or 3D space easily, a sphere can also be used to light a circle on a 2D panel, or even a section of a single strip.

Expandable

The core of the package is the LightShow interface, which is just an abstract base class in Python. Since there are really only 2 core methods that need to be implemented (namely `get_info_at()` and `with_audio()`), the amount of code needed to implement a new LightShow class can be kept relatively small.

Also, in general, note that care was taken to not constrain how expressive a LightShow can be. For instance, `Shape`s and `LightingComponent`s can optionally vary over time, and while many users may not have lighting components that move, disallowing lighting components from being able to vary over time may greatly constrain a user who is designing for a light show where the lights change positions over time.

Other Design Decisions

LightShow outputs map `Light`s instead of `LightingComponent`s to their HSV values. The reason for this is mainly simplicity. Ultimately, lights are associated with indices, and this is something that should be encoded in the LightShow itself. This allows the client of a Lightshow to just go from light index straight to the output instead of needing to worry about decoding what it means for a lighting component to be assigned an HSV value. Overall, this allows the client defined display layer step (in Figure 4-1) to be as basic as possible. If LightShows were to output `LightingComponent` values instead of just individual `Light` values, this would lead to the client potentially needing to do more work after the fact. For instance, if a panel component is red, but 2 of the lights inside of the panel are green, the client would need to resolve this. Instead, we resolve conflicts like this using the `with_importance` abstraction.

`Light`s are also allowed to be "generic" (more info on what this means in section 4.3.2). Most shows default output is on the generic light at index 0 with universe 0, and this is also what certain LightShows that take in the output of a LightShow as input should default to using. This allows simple code, such as `on_component(panel_component, fade(RED, BLUE, 400))`, to hide the abstraction of how `panel_component` and `fade` are connected (it all happens through generic

light 0). However, it still helps to have control over this, in case we want multiple generic inputs.

4.3 Basic Concepts

4.3.1 LightShows

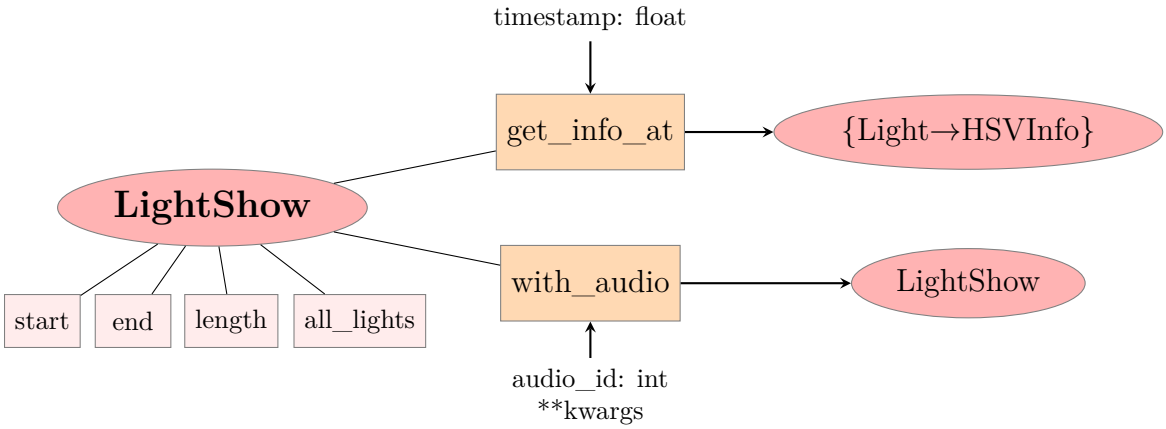


Figure 4-2: The LightShow Interface

LightShow objects are the highest level of abstraction in the LightShow package, and they represent the abstract notion of a LightShow. LightShows are immutable, which makes them highly reusable, but most importantly, they can be queried for information over time.

There are some simple properties that every LightShow has (see Figure 4-2), such as start time (`.start`), end time (`.end`), and length (`.length`) (all in milliseconds), as well as all the possible `Light`s that are ever used (`.all_lights`). However, the most important methods are `get_info_at)` and `with_audio`. Note that `LightingInfoType` (the output of `get_info_at`) is just a map of `Light` to `HSVInfo`.

The `get_info_at` method provides the output of the LightShow at a given time in milliseconds (that is, it maps each light that is on to an HSV value). Sometimes, multiple LightShows may try to control the same light at a time. In order to allow for behavior to be controlled in this situation, it is possible to designate certain LightShows as having higher importance, and a LightShow with higher importance

dominates the output over a LightShow that has lower importance. If two importance values are the same, then the chosen output is unspecified. See the implementation of `constant_lights_on_components` in 6.3.2 for an example.

Note that HSV values can optionally be undefined. For instance, if we have a LightShow that fades from bright to dark, and then another LightShow that controls the hue and saturation, it is possible to combine those two LightShows to get a complete HSV value in the end.

The other major instance method is the `with_audio` method. This method takes in a unique id of the audio (useful for caching to prevent generating extra Python objects, but not required), as well as a set of keyword arguments that provide the LightShow with metadata for contextual shows. For instance, a LightShow that takes a fade, and starts it at every beat would need to know which metadata to look at in order to figure out where the beats are located. Examples of this can be seen in Section 6.4.

4.3.2 Lights

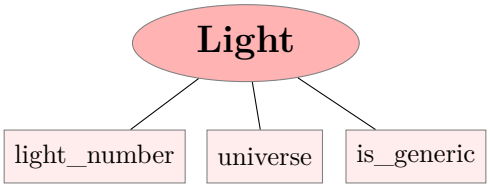


Figure 4-3: The Light Interface

Instead of simply working with indices directly, we use the `Light` abstraction (visualized in 4-3). `Light`s consist of an index, a universe (similar to the DMX concept of a universe, and useful for partitioning lights instead of needing to translate between indices later), and whether or not they are generic. They are also hashable for easy integration with Python dictionaries and sets.

Generic lights are used to represent outputs that are not meant to be the final output, and are instead meant to be inputs to some other modifying LightShow (i.e. we may have a generic light 0 that is green, and then a LightShow that takes whatever

the output for generic light 0 is, and projects it onto a lighting component).

Otherwise, the index is meant to be the unique light within each universe, and multiple universes can be useful in instances where lights may have the same index but not actually be the same light. For instance, if there are two different controllers connected to 2 different strips of 100 LEDs, it is easier to consider controller A to be in universe 0 controlling indices 0-99, and controller B in universe 1 also controlling indices 0-99. Without having universes, the client would be forced to represent this as something like lights 0-99 and 100-199, and then translate after the fact to partition which controller should handle each index.

4.3.3 Lighting Components

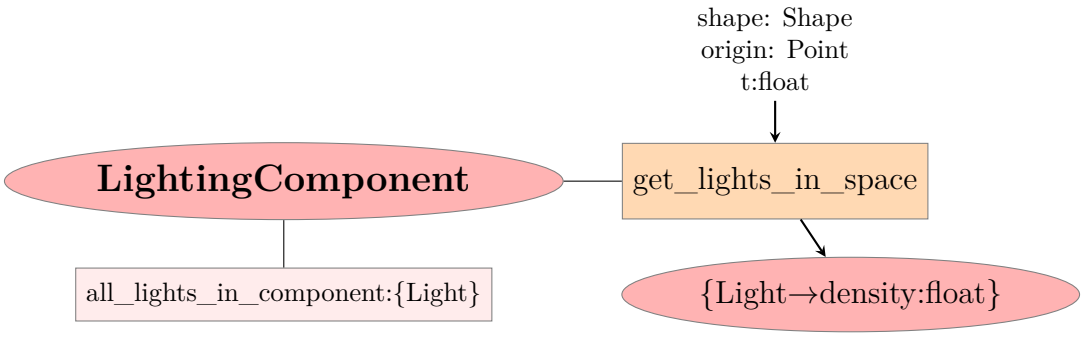


Figure 4-4: The LightingComponent Interface

Often, it is useful to not just think about LightShows strictly in terms of individual Lights. For instance, addressable LED light strips are relatively simple to wire up, but it would be much easier to represent them as an entire strip, and be able to optionally control sections of the strip.

`LightingComponent`s are created once by providing a set of `Light` objects in a certain type of component, and then optionally encoding geometry (i.e. the start and stop locations in space of a strip of lights).

Behind the scenes, `LightingComponent`s always have a geometry, but the default locations in space are chosen in a way that is useful. The default location of the start of a light strip is at the (x, y, z) origin $(0, 0, 0)$, and the end location is at (number of lights in strip, $0, 0$). This allows the client to still light up parts of the component

from the start to the end of the strip for instance, without really caring about where in space that object is.

Lighting components can also be combined to form composite components (for instance, a panel of lights might be created by encoding many `LightStrip`s stacked on top of each other, using the `LightingComponentGroup` class; see the example with `setup_light_components` in 6.3.1). Note that it is still possible to encode a single light with geometry, using the `SingleLight` class.

The two methods that a `LightingComponent` provides are `get_lights_in_space` and `all_lights_in_component` (visualized in 4-4). The method `get_lights_in_space` allows us to query all lights that are contained within a `Shape` (and also how dense the shape is at the location of a given light). The density ranges from 0 to 1 (1 being fully dense), and can be useful for making shapes look smoother when being lit up. Note that the time parameter `t` is available in case a client wants to encode a moving component into a show.

The `all_lights_in_component` method is useful since it just returns the set of all Lights that are in the `LightingComponent`.

4.3.4 Shapes

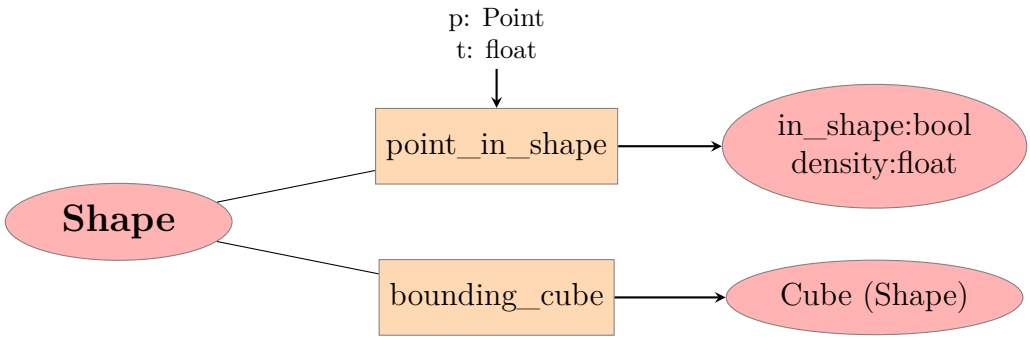


Figure 4-5: The Shape Interface

`Shape`s allow us to use the geometry encoded into `LightingComponent`s to actually manipulate the lights inside a component. The `Shape` interface represents regions in 3D space, and can be used to project lights into `LightingComponent`s.

There are two methods which new shapes must implement: `bounding_cube` and `point_in_shape` (visualized in 4-5). The `bounding_cube` method is useful for optimizations, since it can quickly be used to constrict the region of space that the `Shape` might affect. Then, the `point_in_shape` method returns two values, whether the point is inside of the region controlled by the shape, and then also the density of the shape at that point.

Both of these methods are relatively simple to implement, but note that new shapes can be implemented which vary over time (see the `GrowingAndShrinkingSphere` code in 6.3.5).

4.4 Language

Generally, working with LightShow classes (described in 5) should be avoided directly. For instance, to make a LightShow object that represents a single fading light, we could directly grab an instance of the `Fade` class, but in order to allow for more reusability, wrappers methods should be created (such as `fade()`) to get the LightShow object that we want. For instance, `constant()` is currently implemented as a `Fade` between two of the same colors, instead of adding an entire new class called `Constant`. By putting this method behind a wrapper, the client does not need to care about the difference. A full list of the abstractions currently implemented in the language is available in A.1 and A.2.

Chapter 5

Classes and Operators

5.1 Core Operators

5.1.1 Primitives

LightShows can generally be built up from small primitives, and the package provides some primitive classes, which can be used as the building blocks of more complicated light shows. Among the primitive classes that implement the LightShow interface are:

- `Fade` objects, which have a starting HSV value, and an ending HSV value, as well as a length (and they control a specified set of lights). This class is used to implement the `fade()` and `constant()` language methods (since a constant show can just be a `Fade` between two identical HSV values).

- `Strobe` objects, which alternate between two HSV values (the time hi and time low are inputs to creating the LightShow object). This is wrapped by the `strobe()` method, which provides a useful abstraction by also allowing the frequency of the strobing to be an input (and then it converts to create the correct `Strobe` object). Similar to the `Fade` object, this takes in a `length` parameter, and starts at time zero.

5.1.2 Combiners

There are also a few combiner classes, which allow us to combine and build up LightShow objects to create more complicated light shows. Among these are:

- `Together` (and its corresponding wrapper `together()`), which takes multiple LightShows and returns the LightShow representing all of the input LightShows put together. It resolves conflicts when multiple LightShows try to control the same `Light`.

- `RepeatAt` (`repeat_at()` wrapper), which takes in a single LightShow as well as a set of times, which represents the set of times that the input LightShow should be played at. This is useful when there are shows that need to be scheduled many times over (since `RepeatAt` provides a more efficient implementation than just a large `Together` of shows).

5.1.3 Modifiers

Modifiers are shows that take in a LightShow, and wrap that LightShow in a way that changes the output of the show. Some modifiers might modify the input time (such as `At` and `During`), whereas others may modify the values after the fact.

`PostModifier` is the most general implementation of modifying the output of a LightShow directly, and it takes in a LightShow, as well as two callback functions. The first callback function takes in a `LightInfoType`, and outputs a new `Lighting InfoType` (representing how we modify the result from the input Lightshow in the new `PostModifier` LightShow). The second callback represents how to modify calls to `all_lights`, so that the new output is still consistent with the old LightShow. `PostModifier` is used to implement many of the following useful wrapper abstractions:

- `with_importance()`, which takes in a LightShow, and changes all output from that LightShow to have a new importance.

- `new_controls()`, which takes in a LightShow, and a callback function that is

used to modify which lights an old LightShow controls (i.e. if a LightShow controls light 1, if could be modified to control light 5 with the same outputs).

There are a few other modifiers, namely `At` and `During`, which can be used to set the time that a LightShow begins or is active. `At` is useful for scheduling when a LightShow should begin, since it delays the start by given amount of time. On the other hand, `During` just inactivates a LightShow outside of a range (for instance, if the LightShow should only output values during the chorus, this could be encoded using a `During`, with the start and end times being when the chorus starts and ends).

5.2 High Level Classes

There are also a few high level built in classes that can be used to create abstractions. These classes are by no means a complete set of what might be useful, but some serve as a good starting point to work with.

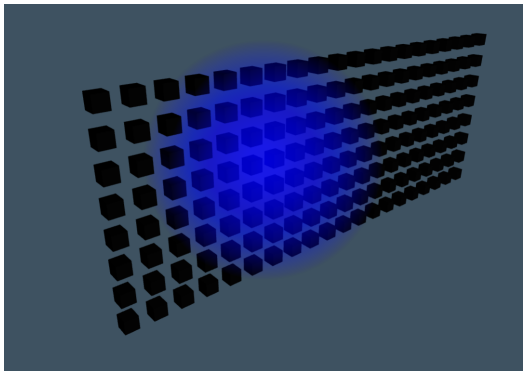
The `DynamicAtEvents` class is useful for scheduling shows to start based on some sort of audio metadata. For instance, the `on_midi()` wrapper uses metadata to schedule a LightShow to start on every note in a midi file that matches the pitch specified in the input, and is implemented using the `DynamicAtEvents` class. Also, since the implementation of `DynamicAtEvents` does not use a specific algorithm, the client could implement a wrapper function that uses any algorithm to schedule LightShow to be played (for instance, the client could use their own beat tracking algorithm to determine where beats should be, independent of the LightShow package).

Another example of a high level abstraction is the `StateChanger` class, which switches between various LightShow outputs based on time, and can be useful for having different LightShows for different sections of music.

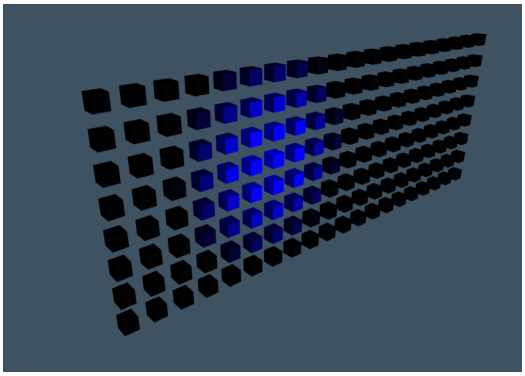
5.3 Geometry Aware Classes

Some classes are useful for projecting onto `LightingComponent`s using `Shape`s. For instance, the `Mover` class takes in a LightShow, a `Shape`, a `LightingComponent`, as

well as a callback that determines position over time. It then moves a shape around a lighting component over time, coloring the shape based on the output from the given LightShow. See `moving_spheres` example for usage in 6.3.4. A visualization of how an abstract shape can be projected onto a lighting component is given in 5-1.



An abstract sphere, lit up blue, around a lighting component panel defined in space.



After using `on_shape` to project a constant blue LightShow onto this panel using the sphere, this is the result.

Figure 5-1: Example of how abstract geometries can be used to generate output.

Chapter 6

Applications and Examples

Since the output of a LightShow is really just HSV values on lights over time, a user of the LightShow package is not constrained to any specific output medium. Raspberry Pi computers or microcontrollers work great for controlling addressable lights in a simple home environment, but a user could also use public APIs to control smart home lighting, or even connect the output to a DMX controller to control stage lights. The user would only need to write code to take the output of a LightShow and actually display it on some medium (the client defined display layer in 4-1).

It is worth noting that lights do not necessarily need to be lights in the physical world. For the rest of this section, lights are represented as colored cubes in a three dimensional rendered environment implemented in three.js [16], running in a web browser. This is implemented by compiling a LightShow into a CSV file of the form as follows, where each line contains information for the lights at one timestamp:

```
timestamp in millis,light index,h,s,v,light index,h,s,v...newline
```

This CSV file is compiled on a server, and is then sent to the web browser, which uses the CSV file to control the colors of the cubes in the web browser. For the following shows, the full code containing all of the shows is available at `https://gist.github.com/bentonw414/5c06612863aa151b27180c25852e7dff`.

6.1 Examples Setup

For all examples, we will use a virtual panel of lights to display a LightShow. The bottom left corner is the light at index 0, and the lights have increasing indices left to right, then bottom to top. Note that all lights are assumed to be in universe zero (see 4.3.2 on lights for explanation of universes). Also, for the examples, we use the following constant values:

```
RED = HSV(0, 1, 1)
BLUE = HSV(.67, 1, 1)
GREEN = HSV(.33, 1, 1)
ONE_SECOND = 1000  # milliseconds
```

6.2 Simple Examples

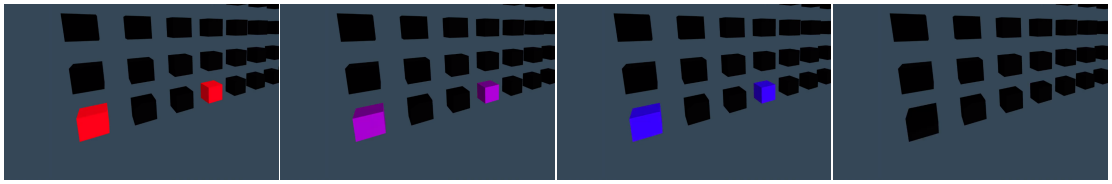
6.2.1 A Simple Fade

For this show, we can just make a simple light show that fades from red to blue as follows:

```
def a_simple_fade() -> LightShow:
    """
    Simple Fade Over 2 seconds from Red to Blue on lights 0 and 3
    """
    lights = {Light(light_number=0), Light(3)}
    return fade(start_value=RED,
                end_value=BLUE,
                length=2 * ONE_SECOND,
                lights=lights)
```

First, we have to pick which lights we want to control. For this show, we will choose lights 0 and 3 (note that by default lights are in universe zero and are not generic). Then, we return a LightShow that is a fade over 2 seconds, from red to blue.

When displayed on the 3D renderer, this lightshow is the following:



$t = 0$ms $t = 1000$ms $t = 2000$ms $t > 2000$ms

Figure 6-1: Output of `a_simple_fade()` when played in the visualizer. Video can be viewed at `https://youtu.be/jtnOrSjO6p4`

6.2.2 Simple Fade Repeated at Times

Instead of just having our simple fade play once, we can reuse that fade, and have it play every 3 seconds, 5 times. This can be done using the `repeat_at` method, reusing the fade from the previous example, and then specifying which timestamps we want to repeat the fade at. The result of playing the `repeated_fade` LightShow is viewable at `https://youtu.be/586Rx5WioEI`(some snapshots are in 6-1).

```
def repeated_fade() -> LightShow:
    """
    Same as a simple_fade, but repeated every 3 seconds, 5 times
    """
    fade_show = a_simple_fade()
    timestamps = [3*ONE_SECOND*i for i in range(5)]

    return repeat_at(timestamps, fade_show)
```

6.2.3 Together With Another Fade Delayed By 4 Seconds

We can also create another longer fade on lights 1 and 2, and delay it by 4 seconds using the `at` method. Then, we can make a new show that combines this show with the `repeated_fade` example using `together`. The result of playing this show in the web visualizer can be viewed at `https://youtu.be/EDmzI53f6TI`.

```
def together_and_delayed() -> LightShow:
    """
    Make lights 1 and 2 do another longer fade together with
    repeated_fade, and delay the start by 4 seconds
    """
    repeated_fade_show = repeated_fade()

    new_fade = fade(start_value=GREEN,
                    end_value=HSV(
                        h=GREEN.h,
                        s=GREEN.s,
                        v=0),  # fade to black
                    length=10 * ONE_SECOND,
                    lights={Light(1), Light(2)})

    delayed_new_fade = at(4 * ONE_SECOND, new_fade)

    return together([repeated_fade_show, delayed_new_fade])
```

6.3 Geometry Aware Examples

6.3.1 Setting up the Geometry of the Scene

In order to use LightShows that depend on the geometry of the lights, we can create LightingComponents in the function below. Note that for this, we represent the panel of lights as a group of strips from the point $(-5, 0, -2)$ at the lower left to $(5, 4, -2)$ at the upper right, and we also manually encode the locations of a few extra lights in the corners.

```
def setup_light_components() -> tuple[
    List[LightingComponent],
    LightingComponent,
    List[LightingComponent],
    LightingComponent]:
    """
    Returns all_strips, panel, extra_lights, all_lights
    """

    # Set up all the strips in the panel
    all_strips: List[LightingComponent] = []

    for row in range(8):
        all_strips.append(
            LightStrip([Light(col + row * 20) for col in range(20)],
```

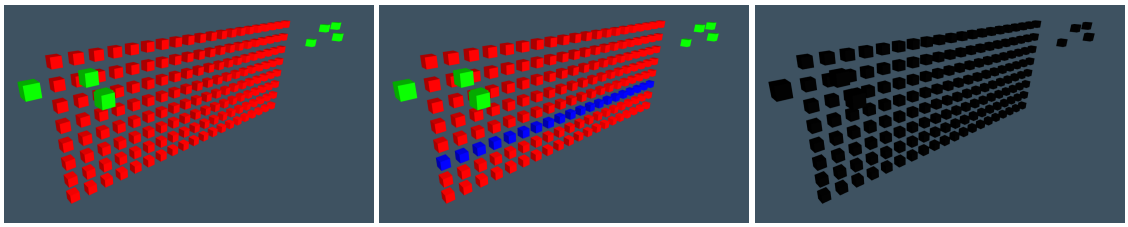
```
                    start_location=Point(-5, row * .5, -2),
                    end_location=Point(5, row * .5, -2)))

    # The panel is just made up of all of the strips
    panel = LightingComponentGroup(all_strips)

    single_cube_positions = [
        [-4.75, 3.5, -.5],
        [-4.35, 3, -.7],
        [-4.75, 3.25, 0],
        [-5.75, 3.5, -.5],
        [4.75, 3.5, -.5],
        [4.35, 3, -.7],
        [4.75, 3.25, 0],
        [5.75, 3.5, -.5],
    ]

    extra_lights = []
    for i, [x, y, z] in enumerate(single_cube_positions):
        extra_lights.append(
            SingleLight(
                Light(i+len(panel.all_lights_in_component())), Point(x, y, z))
        )

    all_lights = LightingComponentGroup(
        [panel, LightingComponentGroup(extra_lights)])

    return all_strips, panel, extra_lights, all_lights
```

6.3.2 Lighting Entire Component

One way to use `LightComponent`s is by using the output from one LightShow to control an entire component. This can easily be done using the `on_component` function. For this LightShow, the panel and the extra lights are lit up to be a constant red and green, respectively. Then, after 5 seconds, the 3rd strip is lit up blue. Note the use of the `with_importance` method, which ensures that the blue strip overrides the red light of the panel (by default, importance is zero, so setting the output of a LightShow to a higher importance ensures that it overrides any other conflicting output).

Also, note that by default, the generic light zero is used by the `on_component` method, which is also the default output from the LightShows created by methods like `constant` and `fade` (this practice makes the code a bit shorter for most LightShow definitions).

```
def constant_lights_on_components() -> LightShow:
    """
    Lights up the whole panel red, the extra lights green,
    but overrides the 3rd strip to be blue after 5 seconds
    """
    all_strips, panel, extra_lights, all_lights = setup_light_components()

    panel_red = on_component(component=panel,
                             lightshow=constant(RED, 10 * ONE_SECOND))

    extra_lights_green = on_component(LightingComponentGroup(
        extra_lights), constant(GREEN, 10 * ONE_SECOND))

    third_strip_blue_after_5_seconds = at(
        5 * ONE_SECOND,
        on_component(all_strips[2], constant(BLUE, 5 * ONE_SECOND)))

    return together([
        panel_red,
        extra_lights_green,
        with_importance(1, third_strip_blue_after_5_seconds)
    ])
```



$0\text{ms} \leq t < 5000\text{ms}$ $\quad$ $5000\text{ms} \leq t < 10000\text{ms}$ $\quad$ $t \geq 10000\text{ms}$

Figure 6-2: Output of `constant_lights_on_components()` when played in the visualizer. Video can be viewed at `https://youtu.be/rT-VYyzvpXg`

6.3.3 Lighting a Shape

Since we have defined the geometry of our scene, we can use that geometry with Shapes to light up intersections of our lights with shapes we define. For instance, we can define a sphere of radius 5 centered around the point $(0, 1, -2)$. Then, we can use the output of a blue fade LightShow to control the color of that sphere. We also specify that we want the sphere to affect the panel component.

After the blue light sphere is done, we can do the same thing in red, but note that

we make a new LightingComponent based off of every other light strip in the panel, so the sphere in red is only shown on those strips. See Figure 5-1 for a visualization of how methods like `on_shape` work.

```
def lights_on_spheres() -> LightShow:
    """
    Lights up a sphere on the panel to be blue,
    followed up by a sphere on only every other strip to be red
    """
    all_strips, panel, extra_lights, all_lights = setup_light_components()
    sphere = Sphere(radius=5, origin=Point(0, 1, -2))

    # make a sphere
    blue_sphere_on_panel = on_shape(
        shape=sphere,
        lighting_component=panel,
        lightshow=fade(BLUE, HSV(BLUE.h, BLUE.s, 0), 4000),
    )

    red_sphere_on_every_other_strip = on_shape(
        shape=sphere,
        # every other strip should be affected by the red sphere
        lighting_component=LightingComponentGroup(all_strips[::2]),
        lightshow=fade(RED, HSV(RED.h, RED.s, 0), 4000)
    )

    return concat([blue_sphere_on_panel, red_sphere_on_every_other_strip])
```

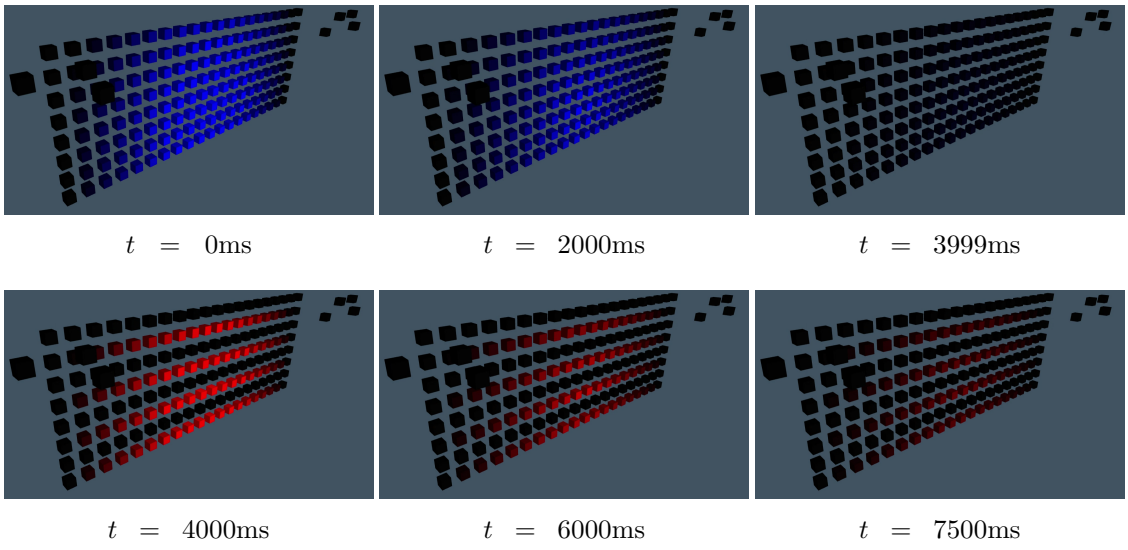


Figure 6-3: Output of `lights_on_spheres()` when played in the visualizer. Video can be viewed at `https://youtu.be/62v4m5V7otw`

6.3.4 Moving Spheres

To demonstrate more clearly how shapes can be projected, we can define a shape that is two spheres near each other, and then pass that to the `Mover` class, which takes in a function of time to generate an offset for projecting the spheres. In this instance, the spheres move left and right, and up and down in a sinusoidal manner. Also, note that while the `Mover` class can be useful directly, it generally should be used in a wrapper that is a bit higher level (for instance the `back_and_forth` method uses the `Mover` class to control shapes going between two points). The output of the `moving_spheres` LightShow can be viewed at `https://youtu.be/maUxdks5x7g`.

```
def moving_spheres() -> LightShow:
    """
    Two spheres that are red, moving left and right and
    up and down in a sinusoidal manner
    """
    all_strips, panel, extra_lights, all_lights = setup_light_components()
    spheres = CompositeShape([
        Sphere(radius=2, origin=Point(0, 1, -2)),
        Sphere(radius=1, origin=Point(3, 2.5, -2))
    ])

    def position_controller(t: float) -> Point:
        """x goes from +4 to -4, y from -1 to 1"""
        return Point(4 * math.sin(t * 2 * math.pi / 2250),
                     math.cos(t * 2 * math.pi / 500),
                     0)

    return Mover(all_lights,
                 shape=spheres,
                 lightshow=constant(RED, 10000),
                 position_controller=position_controller)
```

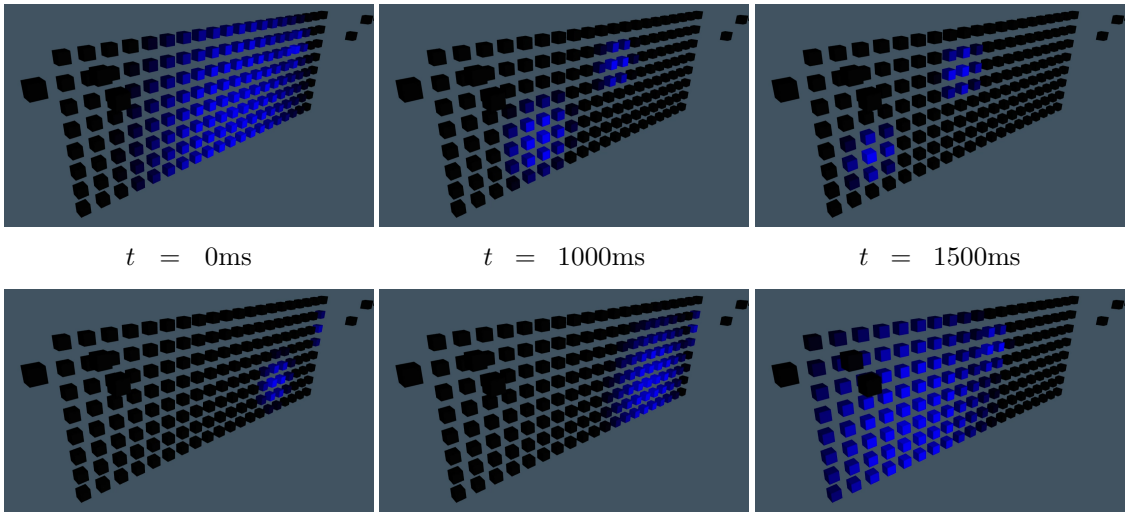
6.3.5 Shapes that Change Over Time

Shapes implement the `point_in_shape` method, but note that `point_in_shape` takes in a time parameter, which allows us to implement shapes that change over time. For instance, if we wanted to implement a sphere that grows and shrinks over time, we could implement it as follows:

```python
class GrowingAndShrinkingSphere(Shape):
    """
    A sphere that changes in size from max_radius to min_radius to
    max_radius repeatedly, taking <cycle_length> time to happen
    Also, the sphere is centered around <origin> in space.
    """

    def __init__(self, max_radius: float, min_radius: float,
                 cycle_length: float,
                 origin: Point = Point(0, 0, 0)):
        self._max_radius = max_radius
        self._min_radius = min_radius
        self._cycle_length = cycle_length
        self._origin = origin
        self._bounding_cube = Cube(
            origin.minus(Point(max_radius, max_radius, max_radius)),
            origin.minus(Point(-max_radius, -max_radius, -max_radius))
        )

    def point_in_shape(self, p: Point, t: float = 0) -> tuple[bool, float]:
        delta_to_point = p.distance_to(self._origin)
        t = t + self._cycle_length/2

        current_radius = self._min_radius + \
            (utils.linear_on_zero_one(t*2* math.pi/self._cycle_length))*\
            (self._max_radius - self._min_radius)

        if delta_to_point > current_radius:
            return False, 0

        solid_radius = current_radius * 0

        if delta_to_point < solid_radius:
            return True, 1

        density = 1-(delta_to_point-solid_radius)/(current_radius-solid_radius)

        return True, density

    def bounding_cube(self) -> Cube:
        return self._bounding_cube
```

Then, this growing and shrinking sphere could be used in a manner as follows (this is similar to the `moving_spheres` example, but the spheres only move side to side, and the left sphere grows and shrinks over time):

```python
def growing_and_shrinking_moving_sphere() -> LightShow:
    """
    Two spheres moving left and right, but one sphere is growing and shrinking
    """
    all_strips, panel, extra_lights, all_lights = setup_light_components()
    spheres = CompositeShape([
        GrowingAndShrinkingSphere(max_radius=4.5, min_radius=0,
                                  cycle_length=4000, origin=Point(0, 1, -2)),
        Sphere(radius=1, origin=Point(3, 2.5, -2))
    ])

    def position_controller(t: float) -> Point:
        """x goes from +4 to -4"""
        return Point(4 * math.sin(t * 2 * math.pi / 2250))

    return Mover(all_lights,
                 shape=spheres,
                 lightshow=constant(BLUE, 10000),
                 position_controller=position_controller)
```



$t = 0$ms $t = 1000$ms $t = 1500$ms

$t = 2500$ms $t = 3000$ms $t = 4000$ms

Figure 6-4: Output of `growing_and_shrinking_moving_sphere()` when played in the visualizer. Video can be viewed at `https://youtu.be/88IZ7bmQMa4`here

6.4 Music Aware Examples

So far, all of the example LightShows have not depended on any sort of audio or metadata. While this may be useful for certain applications, or manually coded single light shows, it is much more useful to have a LightShow that can abstractly

use audio data or metadata to generate lighting. This is what `with_audio` is used for, which generates a LightShow that actually uses the metadata passed into it. In general, the values that are passed into `with_audio` are not specified, but rather up to the implementer of high level abstractions. For this example, we use the `on_midi` abstraction, which expects the location of a midi file to be passed in, and then schedules the given LightShow to be played on every note that matches the pitch in the LightShow. Note that the resulting LightShow is abstract, and by calling `with_audio` on it, we can get a LightShow that actually uses some audio metadata. For the rest of the examples, we will use the song I Turn My Camera On by Spoon.

6.4.1 Simple Usage of `on_midi()`

`On_midi` is an abstraction that relies on the `pretty_midi` library in Python and expects a midi file to open, but under the hood it just generates a `DynamicAtEvents` object, so if an implementer wanted to make their own `on_midi` implementation, that would be completely doable.

In the LightShow below, note that the final output has nothing to do with a specific audio track, so before playing it back we must call `with_audio` like so:

```
lightshow_to_play = on_midi_beats_1().with_audio(0,
    drum_midi_location="./SpoonITurnMyCameraOnDrums.mid")
```

This returns a new LightShow, initialized to play the LightShow with the proper metadata. Below is a basic LightShow using `on_midi`, which just controls lights 0, 1, and 2 to fade on every bass, snare, and hi-hat note (midi transcription was done using the Omnizart library [18]). The result of playing `on_midi_beats_1` with the midi metadata can be viewed at `https://youtu.be/CtdBkH4Vo5E`.

```
def on_midi_beats_1():
    """
    Simple single color fades on lights 0,1,2 that go with the beat
    """
    fade_time = 400

    bass_drum = fade(GREEN, HSV(GREEN.h, GREEN.s, 0),
                     fade_time, lights={Light(0)})
    snare_drum = fade(BLUE, HSV(BLUE.h, BLUE.s, 0),
                      fade_time, lights={Light(1)})
    hihat_drum = fade(RED, HSV(RED.h, RED.s, 0), fade_time, lights={Light(2)})

    midi_file_kwarg = "drum_midi_location"

    return together([on_midi(midi_file_kwarg=midi_file_kwarg,
                             light_show_on_midi=bass_drum,
                             pitch=35),
                     on_midi(midi_file_kwarg,
                             snare_drum,
                             38),
                     on_midi(midi_file_kwarg,
                             hihat_drum,
                             42)])
```

6.4.2 `On_midi()` Combined with Shrinking Spheres

We can expand on this to use Shapes. In the following LightShow, we use two shrinking spheres, and schedule them on the bass and snare drums, and then we also use the hi-hats to control the bottom light strip. The results of playing this on_midi_beats_2 LightShow after calling with_audio can be viewed at `https://youtu.be/1I7P5Y30hS4`.

```
def on_midi_beats_2():
    """
    Spheres on left and right go with the bass and snare,
    then bottom strip is with hihat
    """
    fade_time = 400
    max_sphere_radius = 3
    min_sphere_radius = 0

    green_fade = fade(GREEN, HSV(GREEN.h, GREEN.s, 0), fade_time)
    blue_fade = fade(BLUE, HSV(BLUE.h, BLUE.s, 0), fade_time)
    red_fade = fade(RED, HSV(RED.h, RED.s, 0), fade_time)

    all_strips, panel, _, _ = setup_light_components()
```

```
    bass_drum = on_shape(GrowingAndShrinkingSphere(
                            max_sphere_radius, min_sphere_radius,
                            fade_time * 2,
                            Point(-3, 3, -2)),
                         panel,
                         green_fade
                         )

    snare_drum = on_shape(GrowingAndShrinkingSphere(
                            max_sphere_radius, min_sphere_radius,
                            fade_time * 2,
                            Point(3, 3, -2)),
                          panel,
                          blue_fade
                          )

    hihat_drum = on_component(all_strips[0], red_fade)

    midi_file_kwarg = "drum_midi_location"

    return together([on_midi(midi_file_kwarg=midi_file_kwarg,
                             light_show_on_midi=bass_drum,
                             pitch=35),
                     on_midi(midi_file_kwarg,
                             snare_drum,
                             38),
                     on_midi(midi_file_kwarg,
                             with_importance(1, hihat_drum),
                             42)])
```

6.4.3 `On_midi()` with Moving Shrinking Spheres

The following LightShow is very similar to `on_midi_beats_2`, except in this one, the shrinking spheres move downwards, creating a neat effect. Note that his makes use of the `back_and_forth()` wrapper, which is an abstraction around the `Mover` class. The result of `on_midi_beats_3` can be viewed at `https://youtu.be/jQ15oUVABm4w`.

```
def on_midi_beats_3():
    """
    Same as on_midi_2, but now the spheres are moving
    """
    fade_time = 400

    green_fade = fade(GREEN, HSV(GREEN.h, GREEN.s, 0), fade_time)
    blue_fade = fade(BLUE, HSV(BLUE.h, BLUE.s, 0), fade_time)
    red_fade = fade(RED, HSV(RED.h, RED.s, 0), fade_time)
```

```
    all_strips, panel, _, _ = setup_light_components()

    bass_drum = back_and_forth(start_location=Point(0, 0),
                               end_location=Point(0, -6),
                               time_to_move=fade_time * 2,
                               shape=GrowingAndShrinkingSphere(
                                   3, 0, fade_time * 2, Point(-3, 3, -2)),
                               lighting_component=panel,
                               lightshow=green_fade
                               )

    snare_drum = back_and_forth(start_location=Point(0, 0),
                                end_location=Point(0, -6),
                                time_to_move=fade_time * 2,
                                shape=GrowingAndShrinkingSphere(
                                    3, 0, fade_time * 2, Point(3, 3, -2)),
                                lighting_component=panel,
                                lightshow=blue_fade
                                )

    hihat_drum = on_component(all_strips[0], red_fade)

    midi_file_kwarg = "drum_midi_location"

    return together([on_midi(midi_file_kwarg=midi_file_kwarg,
                             light_show_on_midi=bass_drum,
                             pitch=35),
                     on_midi(midi_file_kwarg,
                             snare_drum,
                             38),
                     on_midi(midi_file_kwarg,
                             with_importance(1, hihat_drum),
                             42)])
```

6.4.4 Matching Album Art Colors

Instead of the colors in the LightShow just being red, green, and blue, wouldn't it be cool if they could automatically match the colors from some album art? We can implement this in a class called `WithAlbumArtColors`, which when initialized with metadata, changes any output from the LightShows passed in, to match the colors extracted from the album art. The implementation of the class is shown in Appendix B, and expects new metadata, namely `album_art_url`, so when we play this LightShow, we have to call `with_audio` in the following way (note that the real Spotify album art url can easily be extracted using the Spotipy library [15]):

```
lightshow_to_play = on_midi_beats_4().with_audio(
    0,
    drum_midi_location="./SpoonITurnMyCameraOnDrums.mid",
    album_art_url="http://spotifyalbumwebsite.com/fakeexample")
```

The result of this on_midi_beats_4 LightShow can be viewed at `https://youtu.be/hIkC3jGkFEY`, and since the following image is the album cover for the song, the resulting show ends up as red and white:


```
def on_midi_beats_4():
    """
    Same as before, but now the colors of the shows match the album art colors
    """
    fade_time = 400

    # doesn't matter what color, since it will be overwritten
    # by the album art color
    generic_fade = fade(GREEN, HSV(GREEN.h, GREEN.s, 0), fade_time)

    all_strips, panel, _, _ = setup_light_components()

    bass_drum = back_and_forth(start_location=Point(0, 0),
                               end_location=Point(0, -6),
                               time_to_move=fade_time * 2,
                               shape=GrowingAndShrinkingSphere(
                                   3, 0, fade_time * 2, Point(-3, 3, -2)),
                               lighting_component=panel,
                               lightshow=generic_fade
                               )

    snare_drum = back_and_forth(start_location=Point(0, 0),
                                end_location=Point(0, -6),
                                time_to_move=fade_time * 2,
                                shape=GrowingAndShrinkingSphere(
                                    3, 0, fade_time * 2, Point(3, 3, -2)),
                                lighting_component=panel,
                                lightshow=generic_fade
```

```
                                )

    hihat_drum = on_component(all_strips[0], generic_fade)

    midi_file_kwarg = "drum_midi_location"

    return WithAlbumArtColors(album_url_kwarg="album_art_url",
                              lightshows=[
                                  on_midi(midi_file_kwarg=midi_file_kwarg,
                                          light_show_on_midi=bass_drum,
                                          pitch=35),
                                  on_midi(midi_file_kwarg,
                                          snare_drum, 38),
                                  on_midi(midi_file_kwarg,
                                          with_importance(1, hihat_drum),
                                          42)])
```

6.4.5 Slightly Changing How We Define Lighting Components

This LightShow is very similar to the previous example of `on_midi_beats_4`, except the spheres move towards each other, and only control every other light strip. Also, the hi-hats control the `extra_lights` component instead of the bottom strip. Overall, this produces a relatively different LightShow, even though a very small amount of code was changed. The results of this LightShow are available at `https://youtu.be/xT2iRABbvn8`.

```
def on_midi_beats_5():
    """
    Similar to midi_beats_4, but now two spheres move towards each other
    and affect different strips. The hihat also only affects extra lights
    """
    fade_time = 400
    generic_fade = fade(GREEN, HSV(GREEN.h, GREEN.s, 0), fade_time)

    all_strips, _, extra_lights, _ = setup_light_components()

    midi_file_kwarg = "drum_midi_location"

    common_shape = GrowingAndShrinkingSphere(
        5, 0, fade_time * 2, Point(0, 1.5, -2))

    bass_drum = back_and_forth(start_location=Point(4, 0),
                               end_location=Point(-8, 0),
                               time_to_move=fade_time * 2,
                               shape=common_shape,
```

```
                            lighting_component=LightingComponentGroup(
                                all_strips[::2]),
                            lightshow=generic_fade
                            )

    snare_drum = back_and_forth(start_location=Point(-4, 0),
                             end_location=Point(8, 0),
                             time_to_move=fade_time * 2,
                             shape=common_shape,
                             lighting_component=LightingComponentGroup(
                                 all_strips[1::2]),
                             lightshow=generic_fade
                             )

    hihat_drum = on_component(
        LightingComponentGroup(extra_lights), generic_fade)

    return WithAlbumArtColors(album_url_kwarg="album_art_url",
                              lightshows=[
                                  on_midi(midi_file_kwarg=midi_file_kwarg,
                                          light_show_on_midi=bass_drum,
                                          pitch=35),
                                  on_midi(midi_file_kwarg,
                                          snare_drum,
                                          38),
                                  on_midi(midi_file_kwarg,
                                          with_importance(1, hihat_drum),
                                          42)])
```

Chapter 7

Conclusion

7.1 Evaluation

For the most part, evaluating this thesis was done by iterating and testing how difficult it was to build new abstractions. For instance, testing how difficult it was to create an abstraction like `om_midi` helped to validate how easily the library could be expanded.

While this thesis ended up being a small package for lighting, much of the initial design was focused around just coming up with a good abstract representation of a light show. Thus, the initial design and general workflow was not necessarily designed around the standard way of doing lighting abstractions. However, after discussing with a lighting professional, it seems like the general way of abstracting light shows is consistent with how lighting is done in a professional setting. While the LightShow package does not implement all of the abstractions that are available in industry level software, it does do a decent job in providing an interface that could implement many abstractions which might typically need to be handled in separate pieces of software. Also, since LightShow is a relatively small and simple Python package, it could make creating small light shows in the home environment much easier to approach since the lighting representation can all be done within the LightShow package.

Typically, different steps of the pipeline require different pieces of software, and while those pieces of software may individually be much more advanced than what LightShow can do, LightShow can represent many of these abstractions in different

ways. For instance, TouchDesigner [17] is a node based visual programming language that can be used to create live, interactive media, and there is other software, such as Resolume [13] which can help map videos to lighting setups. There is even software that can be used to control lights from a laptop (like MagicQ [8]), and oftentimes different software works well for plugging into each other. However, many of these pieces of software specialize in handling one piece of the pipeline. There are also products for home lighting, but most of those provide a small set of preset effects that can be used. LightShow makes it easy to design simple shows without needing to download, learn, and program multiple pieces of software in order to do so. Instead, everything can be done within Python, in a relatively basic package. Also, the package is open source, so as it expands, or as users come up with their own implementations of the LightShow interface, these implementations could be shared and used by anyone else, quickly within their own systems.

7.2 Limitations and Future Work

7.2.1 Flood lighting is not handled directly

In the LightShow package, `LightingComponents` can move, but there is not a great way of encoding how directional lighting affects a scene. For instance, spotlights may change where they point, but the LightShow object cannot encode this information well. Furthermore, the LightShow package does not really handle encoding how the light source may differ from where the light is viewed (i.e. a spotlight which lights up the back of the stage, positioned on the front). For now, we would need to treat the spotlight as having a location somewhere on the back of the stage. Further work is needed to better encode the effects of something like a spotlight or even a flood light.

7.2.2 Representing Shows in Real Time

For now, LightShows are immutable, and in order to initialize a LightShow to play, all the metadata must be known before playing the LightShow. In other words, the

entire audio needs to be known before playing back any audio, so the package is not super usable in live shows.

A better solution may be to use an abstraction similar to a Python stream. Instead of `get_info_at` taking in a timestamp, it could take in a stream of audio data, and return a stream of lighting values. The method signature for this function could be something along the lines of:

```
def get_info_stream(audio_stream: Stream[Any]) -> Stream(LightingInfoType):
    """
    audio_stream is a stream of audio metadata which allows the LightShow
    to generate new lighting values.
    """
```

7.2.3 Exploration of mediums for displaying a lightshow

We are currently familiar with live light shows in person at performances, and this package makes it easier for someone at home with a few lights to design and implement their own light shows. Also, as demonstrated earlier in the examples, LightShows can be rendered in a 3D space. However, as virtual reality and augmented reality grow in popularity, and as more exploration takes place in the world of virtual reality concerts [10], exploring how show shows can be displayed in these virtual or semi-virtual worlds provides a huge area of exploration.

7.2.4 Expansion of the package

While the framework laid out in this thesis may be useful, it does not even begin to exhaust what could be done with MIR in Python. To name a few abstractions that I did not have the time to build:

- For now, the shapes in the language are cubes, spheres, and growing and shrinking spheres, but cylinders, ellipses, and triangles, could be implemented easily.
- An abstraction using some of the work done by [5] to automatically adjust hue colors based on the energy and valence of a piece of audio could be built.

- An abstraction to continually adjust the size of a shape to match something in the audio (for instance, matching the size of a sphere to the intensity of a piece of music).

- An abstraction that automatically detects the verse, chorus, and bridge of a song, and uses different LightShows to generate lighting based on which type of section we are in.

7.2.5 Working with Machine Learning

Some early attempts were made to label songs with simple, high level abstractions using videos to map to abstract LightShows. However, this was done manually, and was quite slow, part in due to a lack of a proper way of abstracting shows.

However, if a system was made that could automatically map from a video to a LightShow, this could allow us to collect more training data for automatically generating LightShows from concert videos. A system like this would also have the immediate use of being able to show a concert video to a model, and immediately be able to replay that lighting on any medium (since we would get a matching LightShow object, which can be used any output medium).

Appendix A

Implemented Abstractions

Table A.1: Implemented Simple Abstractions

Abstraction Name	**Parameters**	**General Description**
at	*time_offset: float* *lightshow: LightShow*	Shifts the input *lightshow* by *time_offset* milliseconds
together	*lightshows: Iterable[LightShow]*	Combines all of *lightshows* into one LightShow.
fade	*start_value: HSV* *end_value: HSV* *length: float* *lights?: Set[Light]*	A LightShow that fades from *start_value* to *end_value* over *length* milliseconds (starting at time zero), controlling *lights* if given, otherwise controlling only generic Light zero.
constant	*color: HSV* *length: float* *lights?: Set[Light]*	A LightShow that is the color *color* over the time [0, *length*] in milliseconds, controlling *lights* if given, otherwise, just generic light zero.
new_controls	*control_definition: (Light)→Set[Light]* *lightshow: LightShow*	Changes the input *lightshow* to control outputs of *control_definition*. That is, if Light `x` is in the output of *lightshow*, those values are instead given to *control_definition(x)*
strobe	*high_value: HSV* *low_value: HSV* *length: float* *lights?: Set[Light]* *time_high?: float* *time_low?: float* *frequency?: float*	LightShow that strobes from time zero to *length* between *high_value* and *low_value*. Controls *lights* (default generic light zero). If *frequency* is given, uses that value for cycles per 1000 milliseconds. Otherwise, *time_high* and *time_low* must be given, and define how long, in each high-low cycle the values should be used.
with_importance	*importance: int* *lightshow: LightShow*	This is the same as just the input *lightshow*, but all outputs have importance of *importance*.
repeat_at	*timestamps:Iterable[float]* *lightshow: LightShow*	Repeats the given *lightshow* at *timestamps*. Equivalent to using *together* and *at* semantically, though this method can be more efficient.
concat	*lightshows: Iterable[LightShow]*	Plays the LightShows in *lightshows* one after the other.
during	*start: float* *end: float* *lightshow: LightShow*	Modifies the input *lightshow* to not have any outputs outside the time range *[start, end]* (in milliseconds)

Table A.2: Implemented Higher Level Abstractions

Abstraction Name	**Parameters**	**General Description**
on_component	*component: LightingComponent* *lightshow: LightShow* *control_light?: Light*	A LightShow that uses the value of *control_light* (default generic light zero) from *lightshow* as the value for setting all the lights in *component*.
on_midi	*midi_file_kwarg: str* *light_show_on_midi: LightShow* *pitch: int*	A LightShow that schedules *light_show_on_midi* to happen at the start of every note that has pitch equal to *pitch*, and expects a kwarg named *midi_file_kwarg* pointing to the midi file to use when with_audio is called.
on_shape	*shape: Shape* *lighting_component: LightingComponent* *lightshow: LightShow* *control_light?: Light*	A LightShow that uses the value of *control_light* (default generic light zero) from *lightshow* to light *shape* on the given *lighting_component*.
back_and_forth	*start_location: Point* *end_location: Point* *time_to_move: float* *shape: Shape* *lightshow: LightShow* *lighting_component: LightingComponent* *control_light?: Light*	Uses the output from *lightshow* on *control_light* (default generic light zero) to color *shape* onto *lighting_component*. However, the origin for the shape is shifted from *start_location* to *end_location* and back (one cycle takes *time_to_move* milliseconds).

Appendix B

Reference Code

```
class WithAlbumArtColors(LightShow):
    """
    Represents a new LightShow that modifies the output of each LightShow
    in <lightshows> to have colors given by 3 most dominant colors in the
    album art at the url <album_url_kwarg>
    """

    def __init__(self, lightshows: List[LightShow], album_url_kwarg: str):
        super(WithAlbumArtColors, self).__init__()
        self._lightshows = lightshows.copy()
        self._album_url_kwarg = album_url_kwarg
        self._with_audio_cache: Dict[int, LightShow] = {}

    def get_info_at(self, timestamp: float) -> LightingInfoType:
        return self._post_modifier.get_info_at(timestamp)

    @property
    def length(self) -> float:
        return max(map(lambda lightshow: lightshow.length, self._lightshows))

    @property
    def start(self) -> float:
        return min(map(lambda lightshow: lightshow.start, self._lightshows))

    @property
    def end(self) -> float:
        return max(map(lambda lightshow: lightshow.end, self._lightshows))

    @property
    def all_lights(self) -> Set[Light]:
        output = set()
        for lightshow in self._lightshows:
            output |= lightshow.all_lights
        return output
```

```
    def with_audio(self,audio_id:Optional[int]=None,**kwargs)->LightShow:
        if audio_id in self._with_audio_cache:
            return self._with_audio_cache[audio_id]

        import requests
        from PIL import Image

        album_cover_url = kwargs.get(self._album_url_kwarg)
        album_cover = Image.open(requests.get(
                                    album_cover_url,
                                    stream=True).raw)
        palette_img = album_cover.quantize(3, kmeans=3)

        # Find the colors that occurs most often
        palette = palette_img.getpalette()
        color_counts = sorted(palette_img.getcolors(), reverse=True)
        colors_hsv = []

        for i in range(3):
            palette_index = color_counts[i][1]
            dominant_color = palette[palette_index * 3:palette_index * 3 + 3]
            colors_hsv.append(
                colorsys.rgb_to_hsv(
                    dominant_color[0] / 255,
                    dominant_color[1] / 255,
                    dominant_color[2] / 255))

        album_color_covers = list(map(lambda hsv:
                                    HSV(hsv[0],
                                    hsv[1], hsv[2]),
                                    colors_hsv))

        # Function for creating the various PostModifier functions
        def info_modifier_factory(index: int) -> Callable[[LightingInfoType],
                                                          LightingInfoType]:
            def info_modifier(old_info: LightingInfoType) -> LightingInfoType:
                new_info: LightingInfoType = {}
                index = index % len(album_color_covers)
                for light, info in old_info.items():
                    new_info[light] = HSVInfo(
                        hsv=HSV(album_color_covers[index].h,
                                album_color_covers[index].s,
                                info.hsv.v),
                        h_importance=info.h_importance,
                        s_importance=info.s_importance,
                        v_importance=info.v_importance
                    )
                return new_info

            return info_modifier

        # Make the various PostModifier objects
```

```
        together_list = []
        for i, lightshow in enumerate(self._lightshows):
            together_list.append(
                PostModifier(
                    lightshow=lightshow.with_audio(audio_id, **kwargs),
                    info_modifier=info_modifier_factory(i),
                    all_lights_modifier=lambda x: x
                )
            )

        new_output = Together(together_list)

        self._with_audio_cache[audio_id] = new_output
        return new_output
```

Bibliography

[1] Aubio: a collection of algorithms and tools to label and transform music and sounds. `https://aubio.org/`. Accessed: 13-June-2022.

[2] Fred Collopy and Robert M. Fuhrer. A visual programming language for expressing rhythmic visuals. *Journal of Visual Languages & Computing*, 12(3):283–297, 2001.

[3] Tom DeWitt. Visual music: Searching for an aesthetic. *Leonardo*, 20(2):115–122, 1987.

[4] Marc-André Gardner, Yannick Hold-Geoffroy, Kalyan Sunkavalli, Christian Gagné, and Jean-François Lalonde. Deep parametric indoor lighting estimation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7175–7183, 2019.

[5] Shih-Wen Hsiao, Shih-Kai Chen, and Chu-Hsuan Lee. Methodology for stage lighting control based on music emotions. *Information sciences*, 412:14–35, 2017.

[6] Randy Jones and Ben Nevile. Creating visual music in jitter: Approaches and techniques. *Computer Music Journal*, 29(4):55–70, 2005.

[7] Peter Kán and Hannes Kafumann. Deeplight: light source estimation for augmented reality using deep learning. *The Visual Computer*, 35(6):873–883, 2019.

[8] Magicq. `https://chamsyslighting.com/products/magicq`. Accessed: 20-June-2022.

[9] Brian McFee, Colin Raffel, Dawen Liang, Daniel P. W. Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th python in science conference*, pages 18–25, 2015.

[10] Kelsey E. Onderdijk, Dana Swarbrick, Bavo Van Kerrebroeck, Maximillian Mantei, Jonna K. Vuoskoski, Pieter-Jan Maes, and Marc Leman. Livestream experiments: The role of covid-19, agency, presence, and social context in facilitating social connectedness. *Frontiers in Psychology*, 12, 2021.

[11] Stephen E Palmer, Karen B Schloss, Zoe Xu, and Lilia R Prado-León. Music–color associations are mediated by emotion. *Proceedings of the National Academy of Sciences*, 110(22):8836–8841, 2013.

[12] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2015.

[13] Resolume. https://resolume.com/. Accessed: 20-June-2022.

[14] Spotify web API. https://developer.spotify.com/documentation/web-api/reference/#/. Accessed: 13-June-2022.

[15] Spotipy Python library. https://spotipy.readthedocs.io/en/2.19.0/. Accessed: 13-June-2022.

[16] Three.js: Javascript 3D library. https://threejs.org/. Accessed: 13-June-2022.

[17] Touchdesigner. https://derivative.ca/. Accessed: 20-June-2022.

[18] Yu-Te Wu, Yin-Jyun Luo, Tsung-Ping Chen, I-Chieh Wei, Jui-Yang Hsu, Yi-Chin Chuang, and Li Su. Omnizart: A general toolbox for automatic music transcription. *Journal of Open Source Software*, 6(68):3391, 2021.