COMPUTER GRAPHICS COURSE PROJECT: MUSIC SYNCHRONISED STAGE LIGHTS

Authored by

Divyanshu Solanki	160100070
Pragun Badhan	160100114
Rohit Sencha	160100068
Tanya Gupta	150040010

Guided by

Prof S.S Pande

Department of Mechanical Engineering

Indian Institute of Technology Bombay

Mumbai 400076

5 November 2019

Abstract

The work starts with a moto to cut the need of a light controlling person in any public concert event or indoor party. The task done by Lightman includes varying the colour, intensity, direction and on/off of these lights. To automate this process, the project takes on various algorithm which will be required to first extract characteristics of sound and then mapping them to relevant light controlling device.

The results of these algorithm can be watched in any rendering software. This project makes use of **Unreal Engine for rendering**. In rendering, sound is divided in packets of small-time intervals (0.1 sec) and then are given to an FFT spectrum analyser which outputs an array of amplitudes of different frequencies which comprises that sound.

Additionally, stage modelling is done in Unreal Engine and the Spectrum data is used to control the intensity of lights.

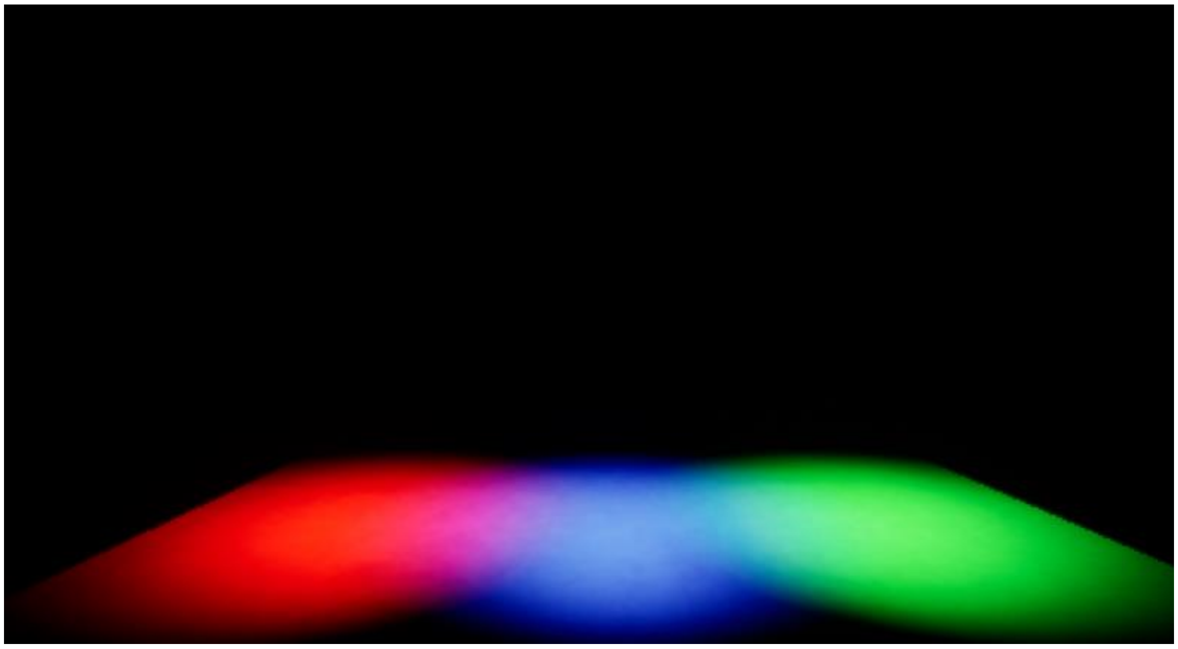
Introduction

It is important to apply and understand the theoretical knowledge learnt from the course and how it applies practically in real life. The Computer Graphics and Product Modelling course has been important for us to understand how a designer can interactively synthesize various product shapes, visualise them in different settings and analyse their functional performance.

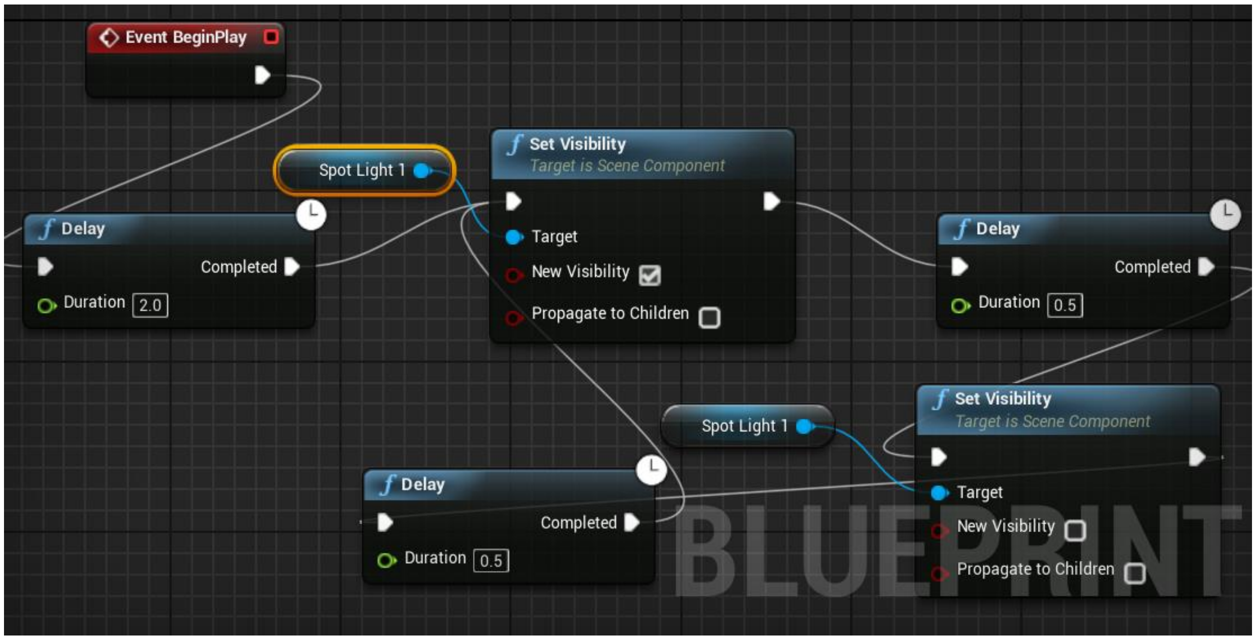
When it comes to making the lights dance in sync with the music, the current scenario is that there is a person who controls the lights, its movement and its intensity in association with the music. We would like to remove this human part and automate the process. This will help reduce costs and remove errors.

We opted for the use of **MATLAB**, which is extremely popular as it is a general-purpose programming language which uses efficient vector and matrix computations alongside **Unreal Engine**, a game engine featuring a high degree of portability used by many game developers today.

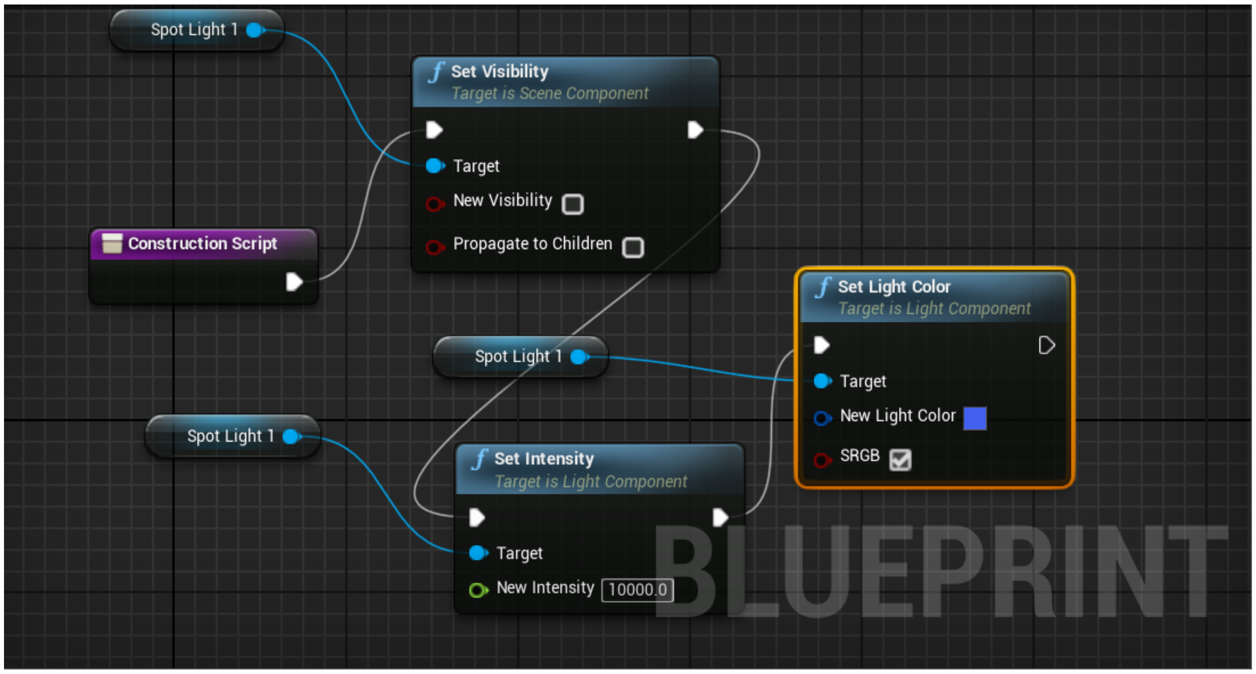
First, we started with a basic understanding of the software Unreal Engine itself and implemented a basic stage light control using time delay. This would have to be predefined for every audio file we play, and hence it is cumbersome and time consuming. Three spotlights of red, blue green colour are selected.



The blueprint for a single spotlight is shown below. We can see it starts playing 2 seconds after we start. The lights play at 0.3 second interval of each other in a loop. Each light remains on for 0.5 seconds and then off for 0.5 seconds. These values have been in respect to the audio file chosen, since the time characteristics will vary between slow/fast songs, etc.



In the construction script we define the characteristics of the spotlight, the one shown below is for setting the intensity of light and the colour of the light.



Blueprints are similarly designed for green and red spotlights and the resulting environment is run, but unfortunately as predicted we see that as soon as we change the audio file, the current settings become invalid and we need to redefine the time settings.

Thus, we see the need for music synchronised lights and have consequently done the required modelling.

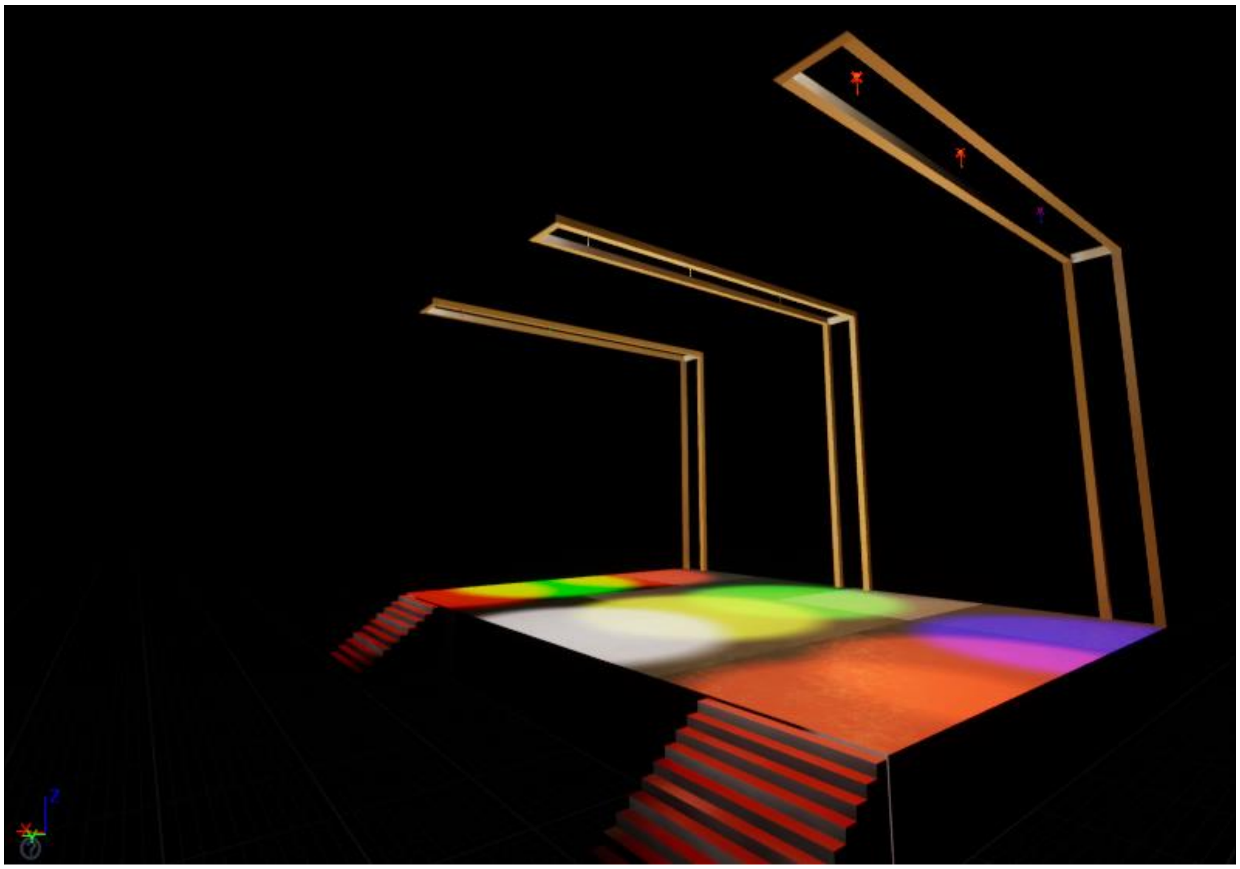
Project Design

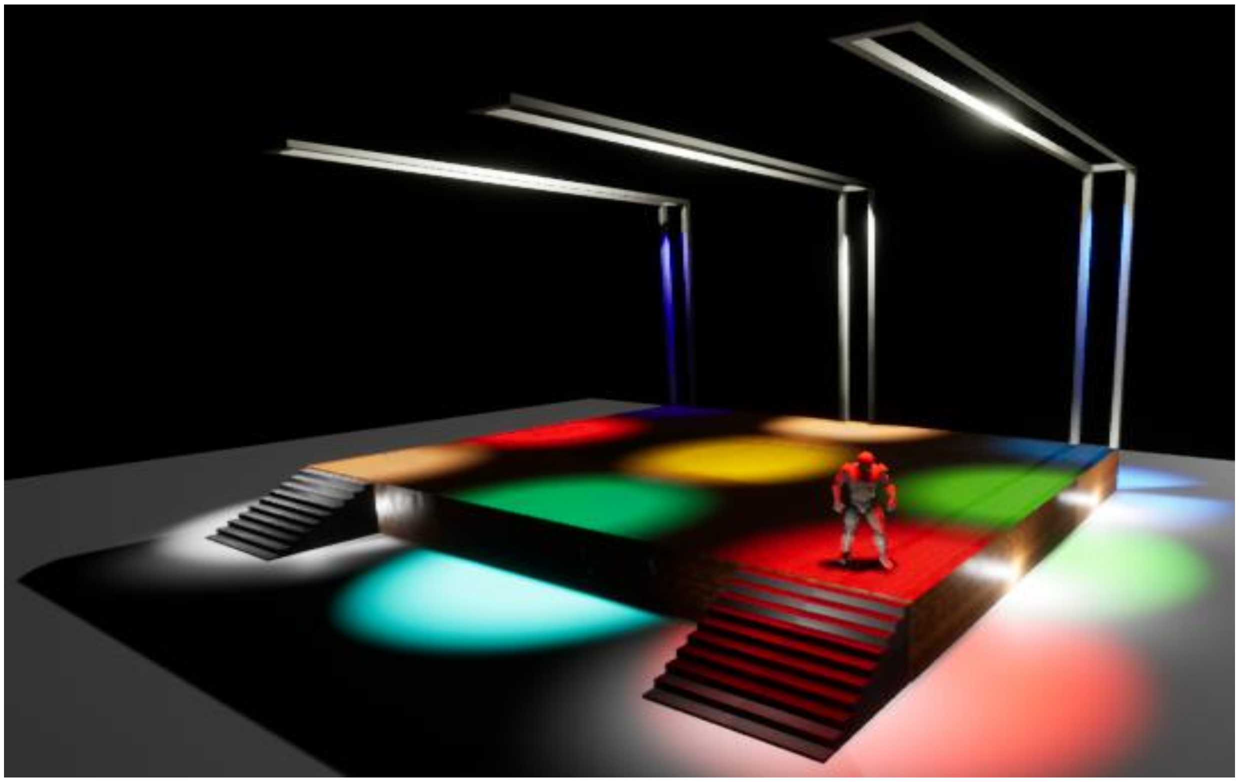
A. Stage Modelling

Environment (Level) was modelled in Unreal Engine 4.23. Level unit consists of a floor for audience, a raised stage for singer and 3*3 spotlight grid whose intensity changes in real time as per characteristics of music. A singer was also modelled and has basic movements like walking and jumping.

Other components of stage include:

- Three overhead ceiling bars to support spotlights
- Static illumination lights around the platform and on ceiling for environment illumination
- Two entry staircases



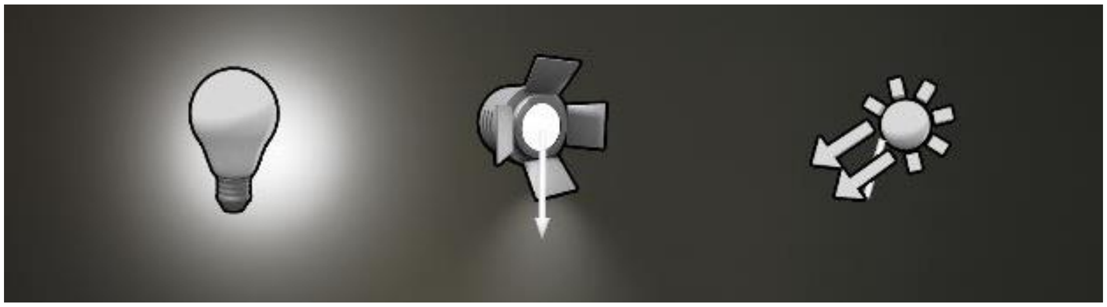


To reduce graphics rendering load on GPU, **LightMass Importance Volume** utility of Unreal was used. The Lightmass Importance Volume controls the volume which requires details lighting. Anything outside of this volume is poorly rendered. Good rendering means multiple light reflections are calculated and on the other hand, poor rendering means only one reflection is considered significant.

B. Lights

Unreal Engine allows us to use different lights according to our needs and control its position and settings accordingly.

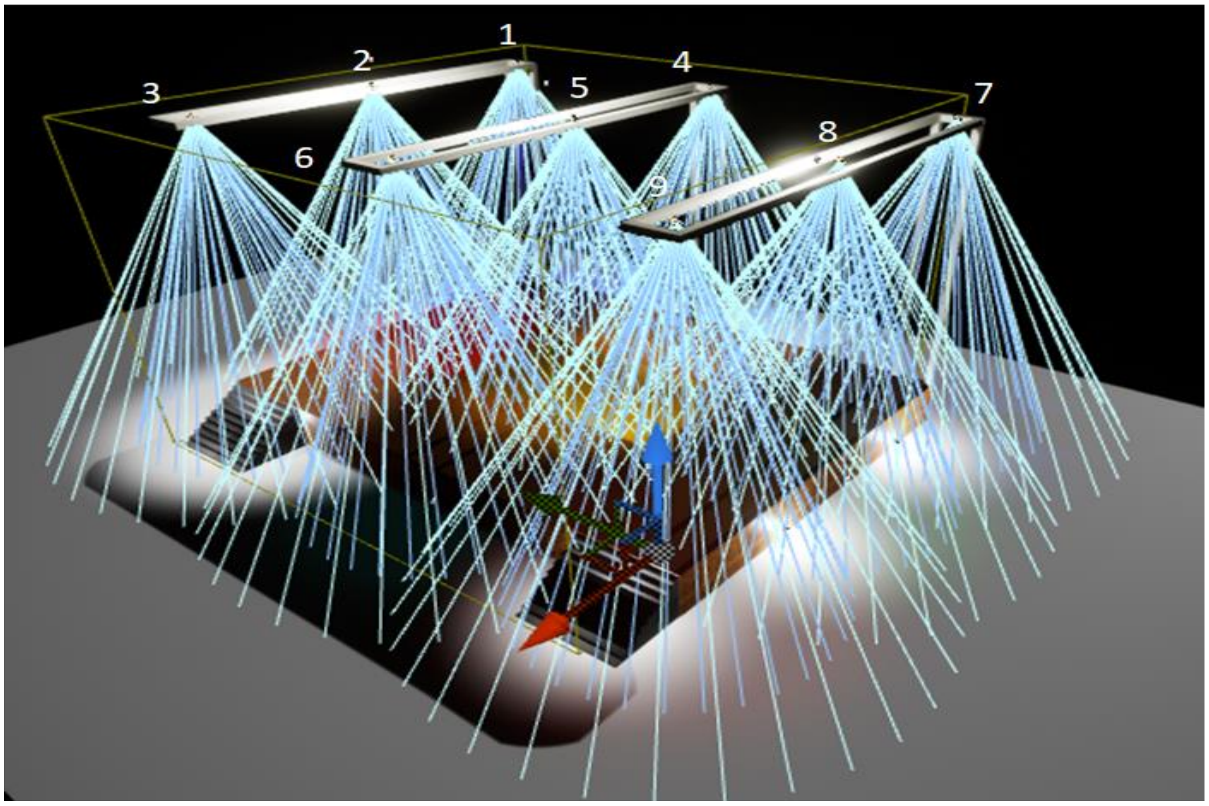
- It provides for 3 types of lights. From left to right they are: **Point Light, Spot Light,** and **Directional Light**



- **Intensity** determines how much energy the light outputs into the scene.
- **Light Color** will adjust the color of the light and the sprite that represents the light in the editor will change its color to match.
- **Attenuation Radius** of the light sets the reach of the light and defines what objects it will affect
- **Source Radius** and **Source Length** define the size of specular highlights on surfaces.

The word ‘static lights’ used before refers to those lights whose intensity remains constant throughout the simulation. Using static lights saves lots of computing load. When a light is declared static, the detailed lighting of environment is computed and then all that information of illumination is hard coded on texture of other materials giving a feel as if there is real light scattering.So calculation for static light rendering is done once only during simulation in unreal editor. So, when we load the program there is no need of real time light rendering calculation for static lights.

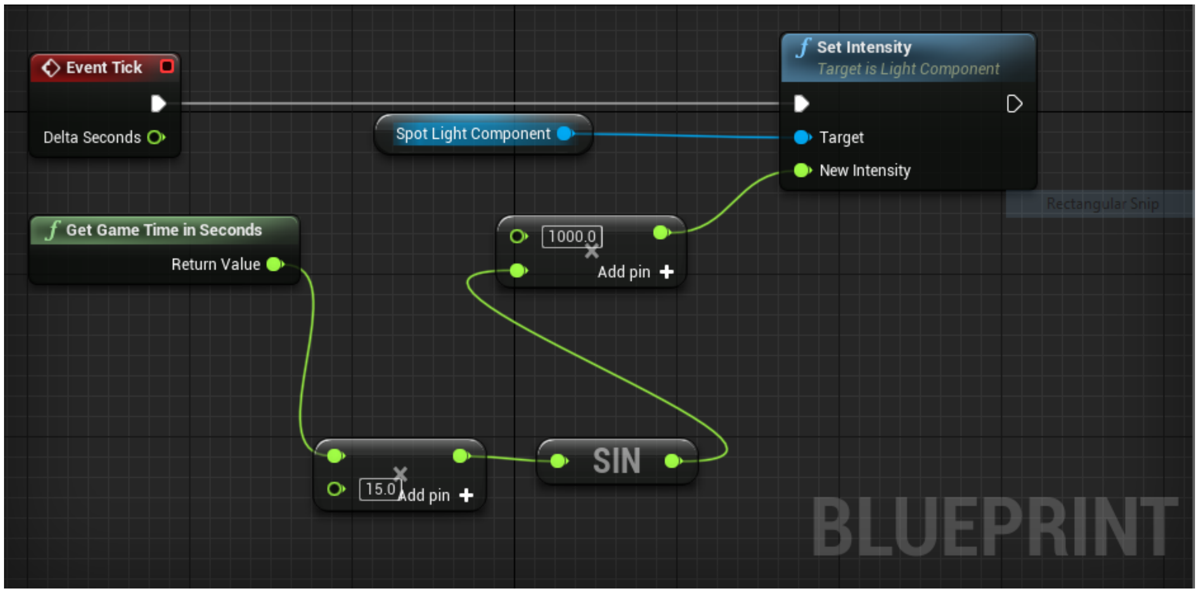
Below image shows the outer and inner cones of spotlight. The surrounding yellow coloured box is LightMass Importance Volume box. Outer cone and inner cone angles of spotlights refers to the solid angle at which they emit lights.



C. Spotlight Animation Design

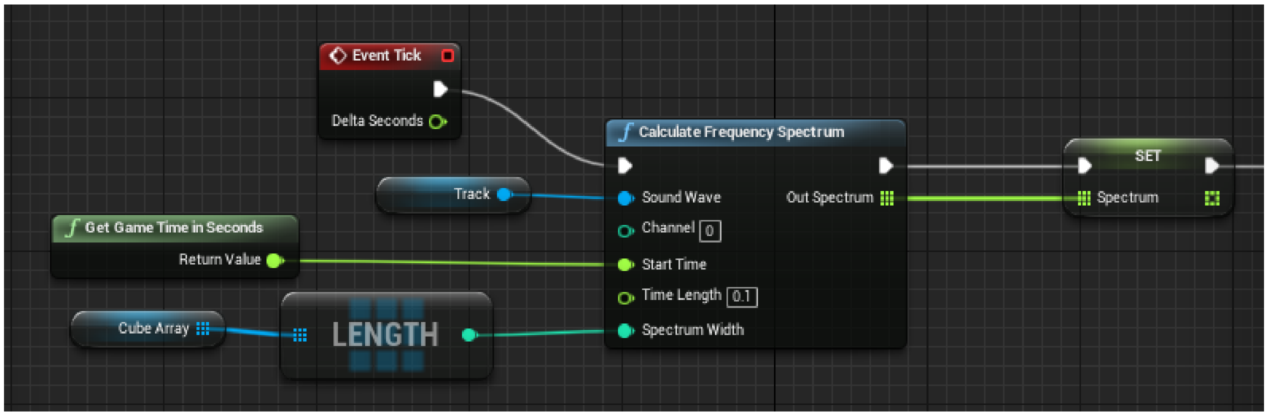
Blueprint allows for visual scripting of simulation events.

Image represents blueprint for a blinking spotlight.

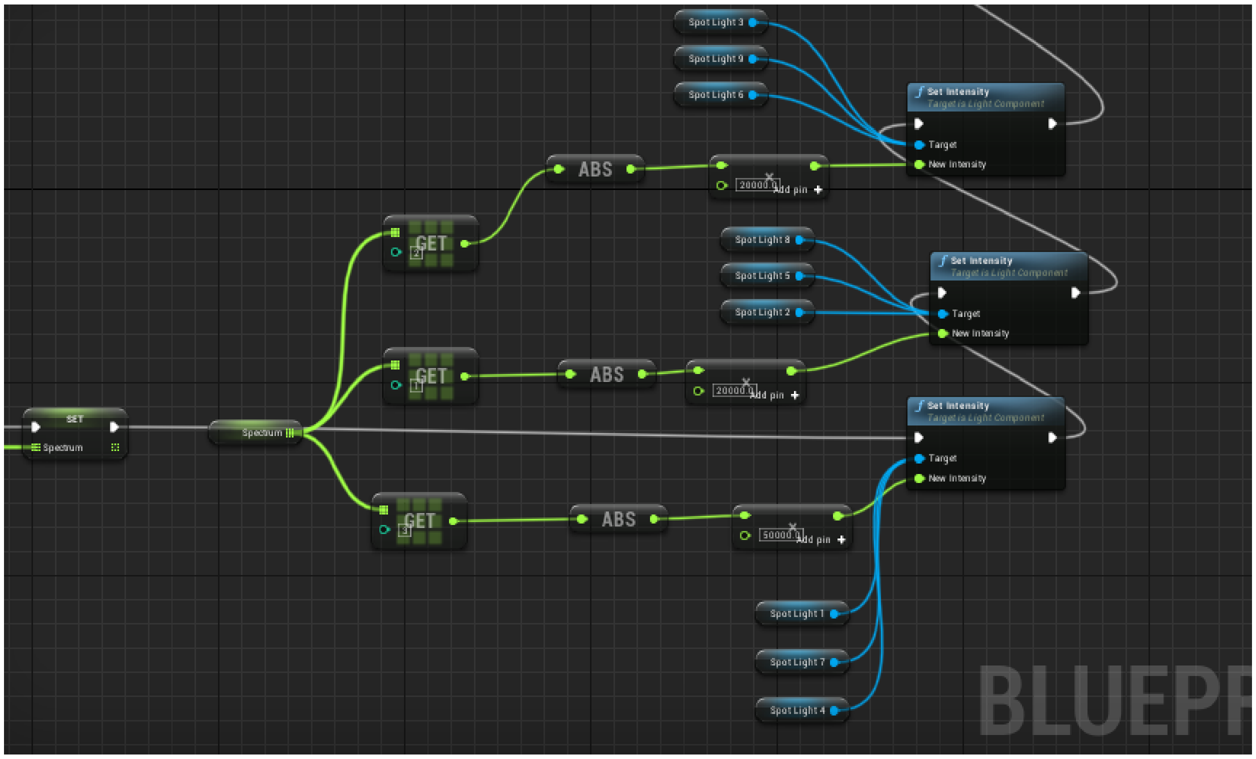


At each instant, **Event Tick** module asks **Set Intensity** module to set intensity of spotlight based on inputs to **New Intensity** value. **Get Game Time in Seconds** sends game time as input to a **multiplier** module. The output is then fed to **sin** function and then to another **multiplier**. **Sin** function enables fluctuating nature of light and multipliers alters frequency of fluctuations and boost intensity of light.

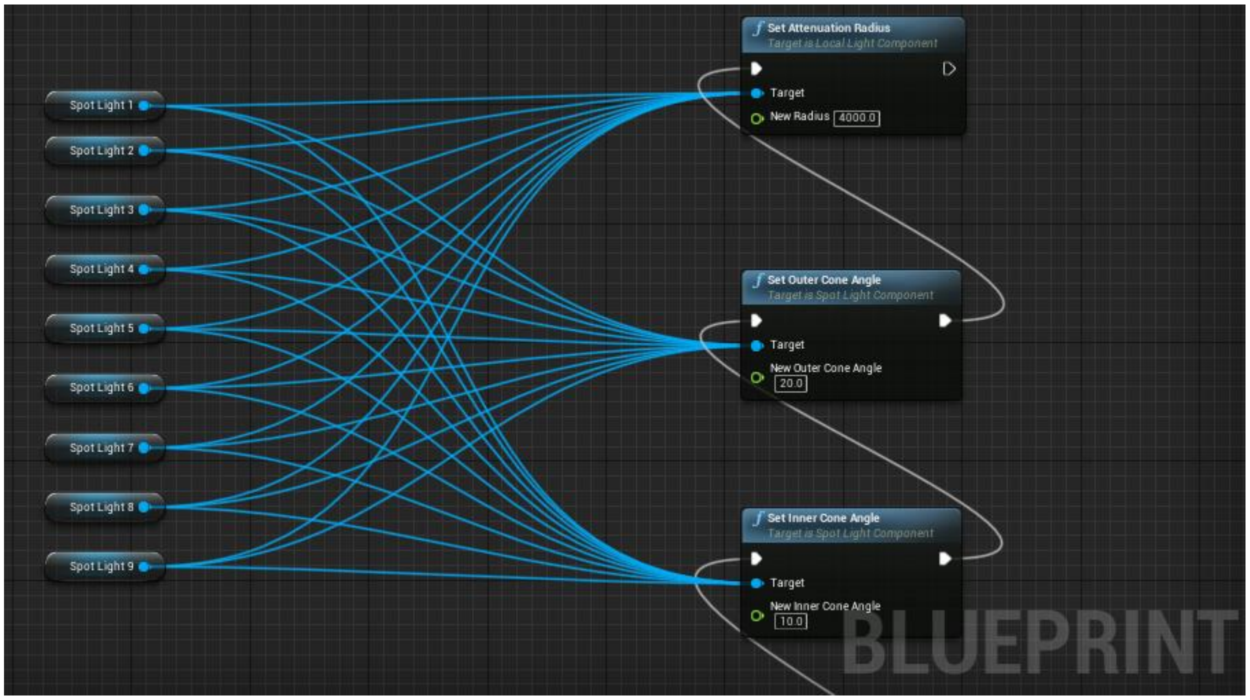
To achieve their intensity control based on song, following scripting was done



The **SET** command stores the calculated FFT spectrum data in an 2d array. We then send this data to intensity of different spotlights in real time. The scripting shown below is in continuation with above image.



The **Outer cone angle** and **Inner cone angle** of spotlights is also controlled with below scripting using spotlight properties. Below shown image is continuation of above image.



The white line running through most of modules shows the functions which run during each Event Tick. We call them primary functions. Event Tick can be understood as series of pulsating input which asks other function to execute their task periodically. Other functions attached to these primary functions are called secondary function and gets called as per requirement of primary functions

Audio Analysis/ Interpretation

The analysis of audio was performed in MATLAB and then, due to limited functionalities of MATLAB and open-source nature of Python, using the Librosa library of Python. The audio analysis was done using the various functionalities of Librosa and established techniques of sound/audio processing. Details of each of these is given below.

Mel Spectrogram

A spectrogram is a visual representation of the spectrum of frequencies of a signal as it varies with time. A Mel scale is a widely used scale in music which converts the frequencies (Hz) to mels as:

$$m = 2595 * \log(1 + \left(\frac{f}{700}\right))$$

A Mel Spectrogram shows which frequencies occur in the audio and at which time with what power/intensity. Hence it is a 3D graph and Librosa uses colour gradients to show the power values. The code below plots the Mel Spectrogram for an audio file with Librosa and Matplotlib libraries.

```
# Path of audio file
audio_path = 'sample.wav'
y, sr = librosa.load(audio_path)

# Creating mel spectrogram of the audio
S = librosa.feature.melspectrogram(y, sr=sr, n_mels=128)

# Converting to log scale (dB)
log_S = librosa.power_to_db(S, ref=np.max)

# Making a new matplotlib figure
plt.figure(figsize=(12,4))

# Displaying the spectrogram on a mel scale
librosa.display.specshow(log_S, sr=sr, x_axis='time', y_axis='mel')
```

```
# Customizing the figure
plt.title('Mel Spectrogram')
plt.colorbar(format='%+02.0f dB')
plt.tight_layout()
plt.savefig('MelSpectrogram.png')
```

<u>**Separating Harmonic and Percussion components of audio**</u>

The total spectrogram of the audio can be divided into two components, namely harmonic and percussive components. The percussive components of the audio correspond to the sound from two colliding objects, for example, drums. Percussive sounds do not have a pitch but a very clear localization in time. Harmonic sounds are perceived to have a certain pitch, for example a violin sound. Usually, a note played on an instrument (say piano) has onset of percussion (hammer strung on piano strings) followed by a harmonic tone (from the vibration of the strings)

In a spectrogram, harmonic sounds form horizontal structures (in time direction) while percussive sounds form vertical structures (in frequency direction). The analysis of these decomposed components of the audio can be effectively utilised for determining the beats, tempo and when a certain pitch (or instrument) starts. The library Librosa provides for decomposition of the audio in the two components. The code for plotting spectrograms for the two is given below.

```
# Separating harmonic and percussive components of audio
y_harmonic, y_percussive = librosa.effects.hpss(y)

# Creating a mel spectrogram of both components
S_harmonic   = librosa.feature.melspectrogram(y_harmonic, sr=sr)
S_percussive = librosa.feature.melspectrogram(y_percussive, sr=sr)

# Converting both to log scale (dB)
log_Sh = librosa.power_to_db(S_harmonic, ref=np.max)
log_Sp = librosa.power_to_db(S_percussive, ref=np.max)

# Making a new matplotlib figure
```

```
plt.figure(figsize=(12,6))

# Displaying the spectrogram on a mel scale for harmonic components
plt.subplot(2,1,1)
librosa.display.specshow(log_Sh, sr=sr, y_axis='mel')
plt.title('Mel Spectrogram (Harmonic)')
plt.colorbar(format='%+02.0f dB')

# Displaying the spectrogram on a mel scale for percussive components
plt.subplot(2,1,2)
librosa.display.specshow(log_Sp, sr=sr, y_axis='mel')
plt.title('Mel Spectrogram (Percussive)')
plt.colorbar(format='%+02.0f dB')
plt.tight_layout()
plt.savefig('Spectro_HarmonicandPercussive.png')
```

Beat tracking and Tempo

Beat tracking simply means finding the time stamps where a listener "taps feet", which is either the onset from a musical instrument which is regularly spaced as a part of the musical rhythm. The library Librosa used in our code detects the beat and tempo by i.) measuring the onset strength ii.) estimating tempo from correlating the onset strength iii.) pick peaks approximately in the tempo and checking the strength of peak. Since the onset determines the beat, the percussive components are used for tracking the beat.

The code for beat tracking and tempo is given below.

```
# Beat tracking use the percussive components
tempo, beats = librosa.beat.beat_track(y=y_percussive, sr=sr)

# Displaying the spectrogram, with the detected beats
plt.figure(figsize=(12,4))
librosa.display.specshow(log_S, sr=sr, x_axis='time', y_axis='mel')
```

```
# Draws transparent lines over the beat frames in the plot
plt.vlines(librosa.frames_to_time(beats),
           1, 0.5 * sr,
           colors='w', linestyles='-', linewidth=2, alpha=0.5)
plt.axis('tight')
plt.title('Mel Spectrogram with beats (Vertical Transparent lines
denote the beats)')
plt.colorbar(format='%+02.0f dB')
plt.tight_layout();
plt.savefig('/Users/pragunbadhan/Desktop/CGPM/sample_Graphs/Percussive
BeatsSpectro.png')


# Printing the beats frames and times
print('Estimated tempo:      %.2f BPM' % tempo)
print('First 5 beat frames:   ', beats[:5])
print('First 5 beat times:   ', librosa.frames_to_time(beats[:5],
sr=sr))
```

<u>Chromagram</u>

A chromagram represents what pitch classes belong in a sound and its various components. A pitch class is the set of all pitches which are a whole number of octaves apart. A chromagram is a 3-D graph showing the power/intensity of each pitch class at a time instant in the audio sample. This is particularly useful for us as the onset of an instrument can be tracked in the chromogram and correspondingly the light can be moved and the duration for which it is lit.

In the code the following command is used for creating a chromogram.

```
C=librosa.feature.chroma_cqt(y=y_harmonic,sr=sr, bins_per_octave=36)
```

Spotlight colour randomization

To give the concert environment a dynamic colour feel, we have chosen to randomize the colours of each spotlight, along with change in flickering speed

Results

- The visuals of controlling stage lights as per its frequency spectrum are appreciable.
- Other inputs such as beats per seconds and length of beats will give a more intuitive visual of lights.
- Doing an FFT for sound packets of 0.1 second extracted from lengthy song is easy on computation load.
- Taking output from frequency spectrum in forms of amplitudes and randomly distributing to different lights gives a more immersive experience. This step ensures that light spectrum do not look like an equalizer pattern which may be monotonic.

Conclusion and Future Work

Building on our understanding of the mathematical basis for product modelling and aim of implementing an interactive stage design for controlling lights on the beats of music, we have designed such a stage in Unreal Engine. Alongside this, audio analysis has been done in Matlab and Python.

Future Work:

- Post-processing effects to make the lights match the music even more accurately. These include strobes, black out, bright white flashes, hue shifts.
- Currently, we are limited to music files already stored in system. We would like to expand the capability to take in upper input of audio tracks in real time.
- If possible, implement a live demo of the working of our algorithm by designing an electrical circuit and using LED lights.