Spring
2016

Senior Project Report

MUSIC REACTIVE LED LIGHTS

PETER CHU, ADVISED BY DR BRIDGET BENSON

Abstract

This project enhances the user's music listening experience by not only adding but also synchronizing visuals to the music. The project digitally analyzes music being played back and controls LEDs to give an audio-visual experience. This report describes the materials and knowledge used to create a proof of concept for a wireless and portable music reactive LED lighting system.

Acknowledgments

I would like to express my gratitude and appreciation for the suggestions and opinions I have received from my friends regarding this project. Most importantly, I would like to thank my advisor Dr. Bridget Benson for allowing me to turn what was originally a hobby project into my senior project when I approached her about my discontent for my former senior project.

Table of Contents

List of Figures

List of Tables

Introduction

The global electronic dance music (EDM) industry was worth $6.9 billion in 2014 with North America being about 29 percent, or $2 billion, of the global market. EDM concerts and festivals are prime examples of audio-visual entertainment and attracted 1.4 million attendees in the United States alone in 2015 [3,8]. Many of the music found on YouTube, especially of the EDM genre, have at least one music visualizer equivalent entry. With such a market and the fact the average American listens to over 4 hours of music a day [11], a music reactive lighting LED system is relevant to a substantial number of people.

There are a couple of similar systems already out on the market: ViVi – Music LED Controller from VisualVibes and the Lumazoid from Nootropic Design. ViVi aims to "…liven up any party, impress your friends, or wow a crowd all for a fraction of the cost of existing products…" [5]. Lumazoid aims to provide a programmable visualizer board that "lets you display an awesome lightshow that is synchronized to your music." [6]

Both the ViVi and the Lumazoid are real-time LED music visualizers just like this system. All three systems analyze audio in the frequency domain in real-time for information that is used for creating visualizations with the LEDs. ViVi and Lumazoid both support the WS2812 chipset LEDs while this system supports the APA102 chipset LEDs. This system is similar to ViVi in that both are meant to be plug and play and customizable with a mobile app, where the Lumazoid is sold as a board without an enclosure that is programmable with Arduino and requires some soldering for setup. This system in different and unique in that the music is being played back and processed on a remote computer, and the analysis of the music is streamed to the microcontroller via a server and client connection over Wi-Fi. As mentioned earlier, visualizations are customizable by a mobile app that is also connected to the microcontroller. The purpose of this project is to provide visual entertainment to users who are seeking to enhance their music listening experience.

Requirements

Overview

The project needs to have an LED strip react to audio in real-time and over Wi-Fi. The ESP8266 communicates with the LED strip via SPI with only the SDO (data) and SCK (clock) lines. There is a logic level shifter between the ESP8266 and the LED strip because the former is 3.3V device and the latter is a 5V device. The ESP8266 and the PC running the Processing app must be on the same network. The audio signal processing must be done by the Processing app using the Minim library and the Fast Fourier Transform (FFT) within it. The project works with any audio but really only makes sense with music. Because it reacts to audio in general, there is no limit to the number of songs it supports. There are four different visualization modes and some have the ability to be customized in terms of speed and color. The project also requires the use of Blynk, a platform on iOS and Android that is used to control devices, in this case the ESP8266, over the Internet. The system needs to run until it is turned off by the user. The system cannot be tethered to a PC or a power socket and needs to be convenient to place around a house.

Table 1 provides the justification for how the engineering requirements stem from the marketing requirements. The engineering requirements are described in the engineering column and the justification column describes why they are needed. The marketing column lists the specific marketing requirement(s) that the given engineering requirement stems from.

Engineering Requirements

***Table 1**: System Requirements and Specifications*

Specification	Marketing	Engineering	Justification
1	6	The system size excluding the LEDs is equal to or less than (191 x 110 x 57) cm^3 and weighs equal or less than 550 grams.	The system needs to be portable so it can conveniently fit into a backpack and be taken to various locations.
2	2, 6	The system needs to receive beat information via Wi-Fi from a device hosting a TCP server on the same network.	The lighting part of the system needs to be separate from the music playback part so the lighting part can be freely placed.
3	4	The system needs to sustain at least 4 hours of typical* use.	A typical adult social gathering the system will be used for typically lasts around 4 hours. [2]
4	1	The system requires real-time audio analysis.	Since it is not practical to program light shows per song, the system needs to adapt to each song.
5	3	The system requires at least 4 different visualizations modes.	Certain visualization modes provide more interesting results for certain type of music than others.
6	5	The system cost cannot exceed $100.	The system needs to be competitively priced [4,5].
7	7	The system uses off-the-shelf components from Adafruit and SparkFun.	The parts need to be easily acquired and replaceable.

*Typical use is described as 30% output at 1200 mA for the LEDs and 80 mA for the Thing [ref#]

Marketing Requirements

1. React to audio in real-time
2. Wireless
3. Customizable visualizations
4. Adequate Battery Life
5. Low Cost
6. Portable
7. Easily repairable/replaceable

Background

Understanding this project requires a basic understanding of two core concepts: Fourier transforms and musical beats. The Fourier transform plays an important role in analyzing the audio for beats. The results of the analysis determines whether there is a beat at any given time, and the purpose of this system is to use LEDs to visualize the beats that are found.

Fourier transform

Human audible sound consists of a frequency range roughly spanning 20 Hz to 20 kHz [12]. In order to analyze the music being played, it must be converted from the time domain to the frequency domain with the application of a Discrete Fourier Transform (DFT). This results in a representation of the same sound in the frequency domain and that is where the analysis takes place.

The specific usage of the DFT in this project includes a buffer size of 1024 bytes and a sample rate of 44,100 Hz. The resulting DFT object has logarithmically calculated averages because that is how humans hear sound. [9]

Below, as a series, Figures 1 through 4 show the relationship between the time domain and the frequency domain of a signal [1]. First in Figure, 1 an arbitrary signal is shown in the time domain in red. Then in Figure 2, the same signal is shown as a composition of different signals at different frequencies in blue. When these signals of different frequencies are added together, they result in the original signal shown in Figure 1. Figure 3 shows the same signal in the frequency domain in blue. Figure 4 shows the same signal in the time domain in red and frequency domain in blue side by side.

***Figure 1**: Signal in Time Domain (X-time, Y-amplitude)*

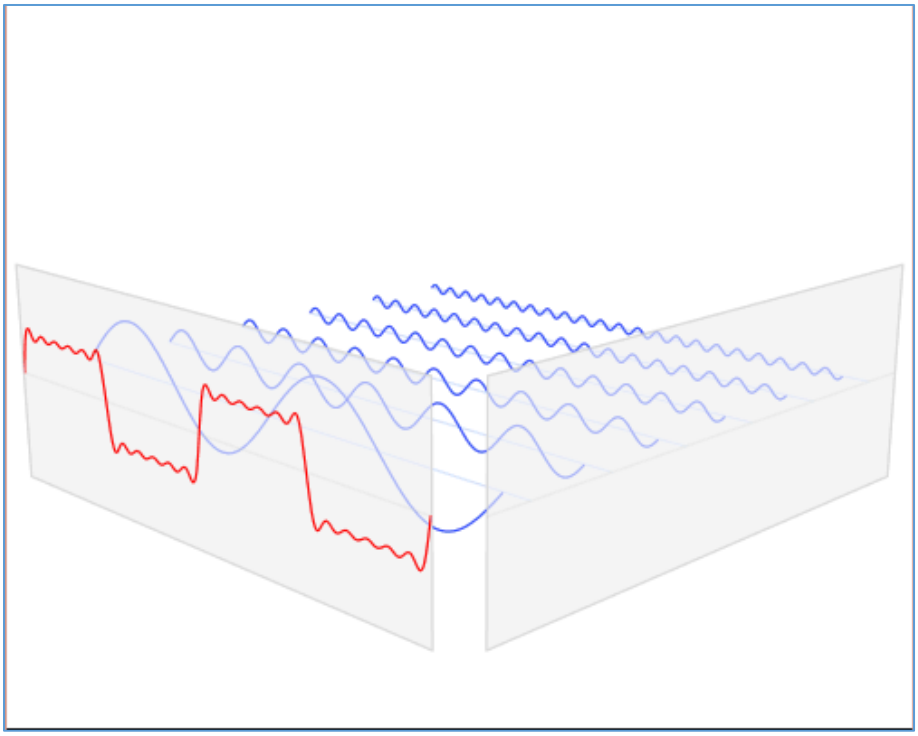

***Figure 2**: Signal in Time (X-time, Y- amplitude) and Frequency Domain (X-frequency, Y-amplitude) – Overlapped*

***Figure 3**: Signal in Frequency Domain (X-frequency, Y-amplitude)*

***Figure 4**: Signal in Time (X-time, Y-amplitude) and Frequency Domain (X-frequency, Y-amplitude) - Juxtaposed*

Beat detection

This project uses Frederic Patin's definition of a beat, "… a sound will be heard as a beat only if his [*sic*] energy is largely superior to the sound's energy history, that is to say if the brain detects a brutal variation in sound energy." Patin's algorithm detects energy variations by comparing the average sound energy of the signal and comparing it to the instant sound energy. If the instant sound energy is superior to a local average sound energy, a beat is detected. [7]

This algorithm can be applied to the entire spectrum of sound or just specific frequency bands. Through careful selection of particular frequency bands, such as the bass band (60-250Hz) or high mids (2000-6000 Hz), different beats made by different instruments can be individually represented.

Figure 5 below shows the energy levels of the sound spectrum for the song being played. Each vertical bar represents a frequency band. The frequency range spans 0 to 22050 Hz from left to right. The height of the vertical bar represents the average sound energy at that frequency band.

Green = local statistical maximum = set mean + 3 * standard deviation of set
Blue = local statistical mean = set mean
White = instant average sound energy

A beat is detected when the instant average sound energy is higher than the local mean average sound energy or, visually, when the white bars are higher than the blue bars. To reduce noisy beat detections, a sensitivity parameter controls how many beats can be found in a certain time frame, i.e. a maximum of one beat per 200 milliseconds.

The yellow and red bar on the right is the average sound energy of the whole spectrum. It ended up not being used in the project but may have future potential.

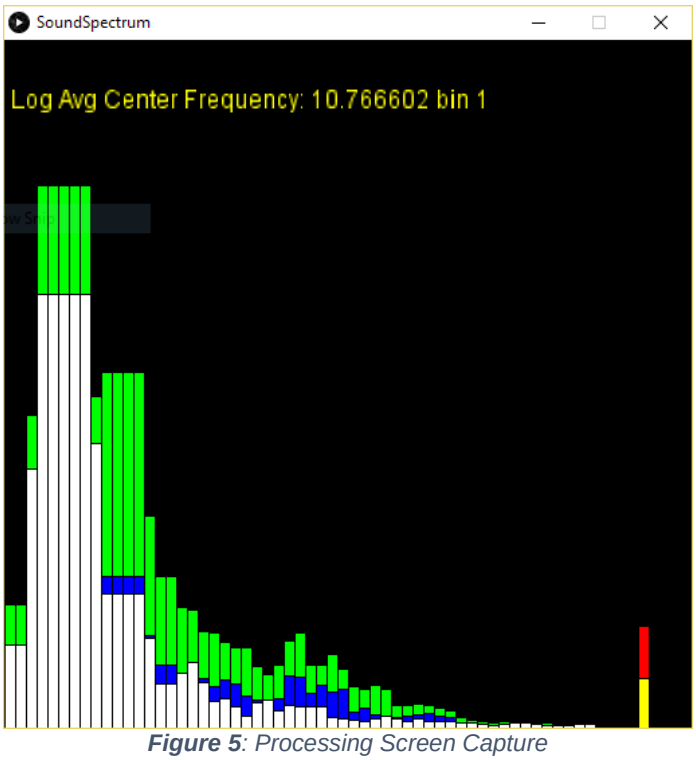

***Figure 5**: Processing Screen Capture*

Design

Hardware

Figure 6 below is the hardware block diagram and it shows the direction data and power flows throughout the system.

Music is played back on a remote PC running the music analysis software. The music analysis software streams beat information over Wi-Fi to the ESP8266 board. The board controls the visualization of the LEDs with the beat information.

The battery powers the ESP8266 board and the LEDs. It also provides the 5V reference voltage for the logic level shifter while the ESP8266 provides the 3.3V reference voltage.

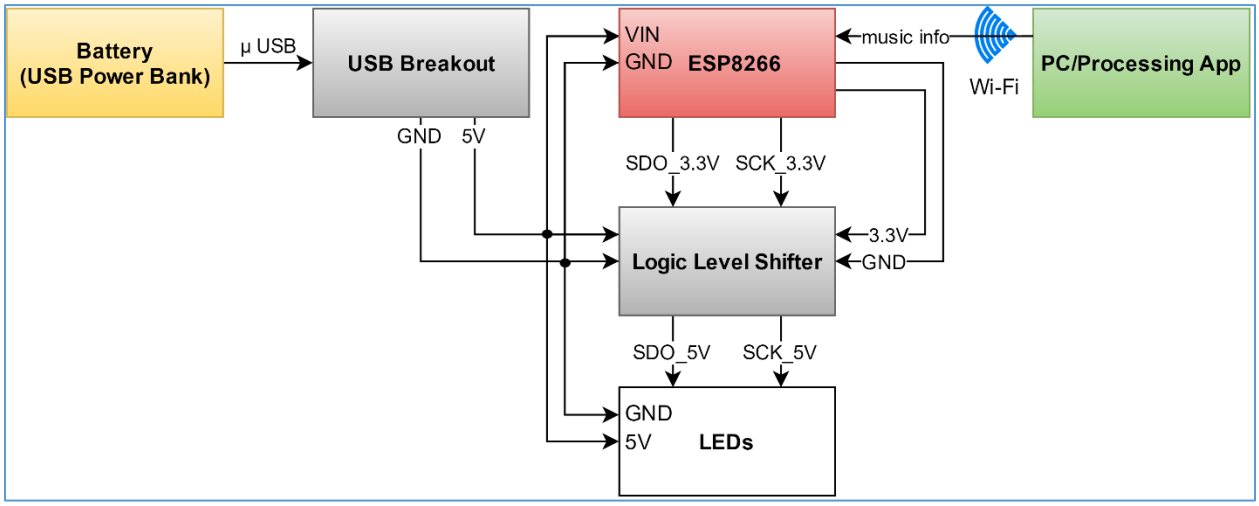

***Figure 6**: Hardware Block Diagram*

The ESP8266 was chosen for its Wi-Fi capability. The original idea was to use an Arduino Uno, but the limitation of the Uno being tethered to the PC running the signal analysis program was unfavorable. With the ESP8266, the LED strip is much more portable and can respond to the music as long as PC is on the same wireless network.

The digital LED strip chosen is the Adafruit Dotstar (APA102c). It was chosen against another popular digital LED, NeoPixel (WS2812b), because it is said to be easier to interface to a broader range of devices and there is no strict timing requirements.

The USB breakout board is the Adafruit USB Micro-B breakout board. It offers the 5V and GND pins needed to power the LEDs and ESP8266 and provide a reference high voltage for the Logic Level Shifter.

The battery is an Anker Astro E1 with 5200mAh. It was chosen for its small size and that it would fit into the project box.

In Figure 7 and Figure 8 below, the components are shown and put into a project box in a working state condition. The components are listed as follows:

1. Anker Astro E1 (battery)
2. SparkFun ESP8266 Thing
3. SparkFun Logic Level Converter
4. Adafruit USB Mini-B Breakout Board
5. Adafruit DotStar Digital LED Strip

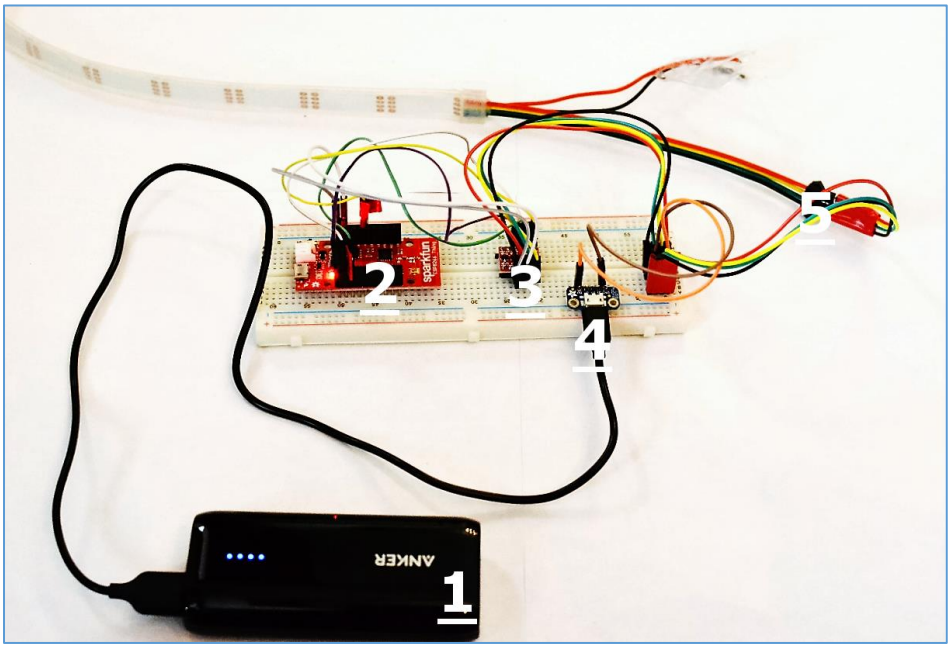

***Figure 7**: The Components Laid Out*

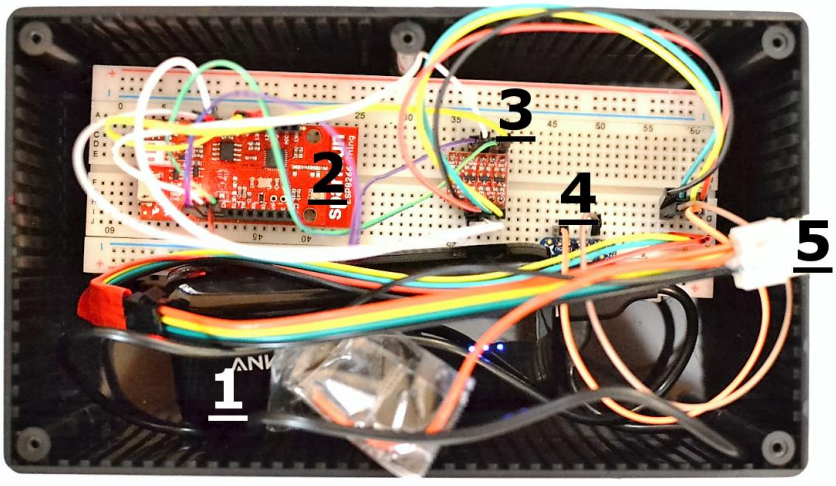

***Figure 8**: Components all inside a Project Box*

Figure 9 and Figure 10 below show the system in its final boxed form. Figure 10 captures a moment of the system in operation.

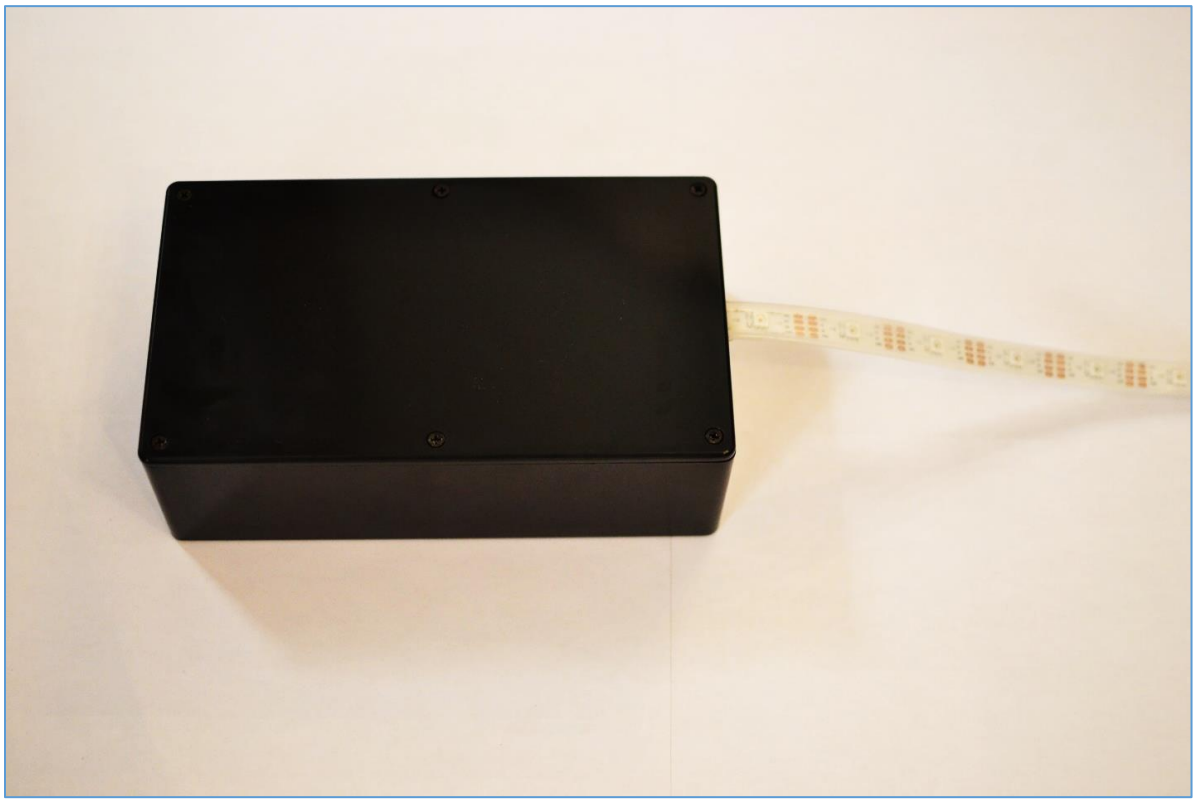

***Figure 9**: Final Boxed Form*

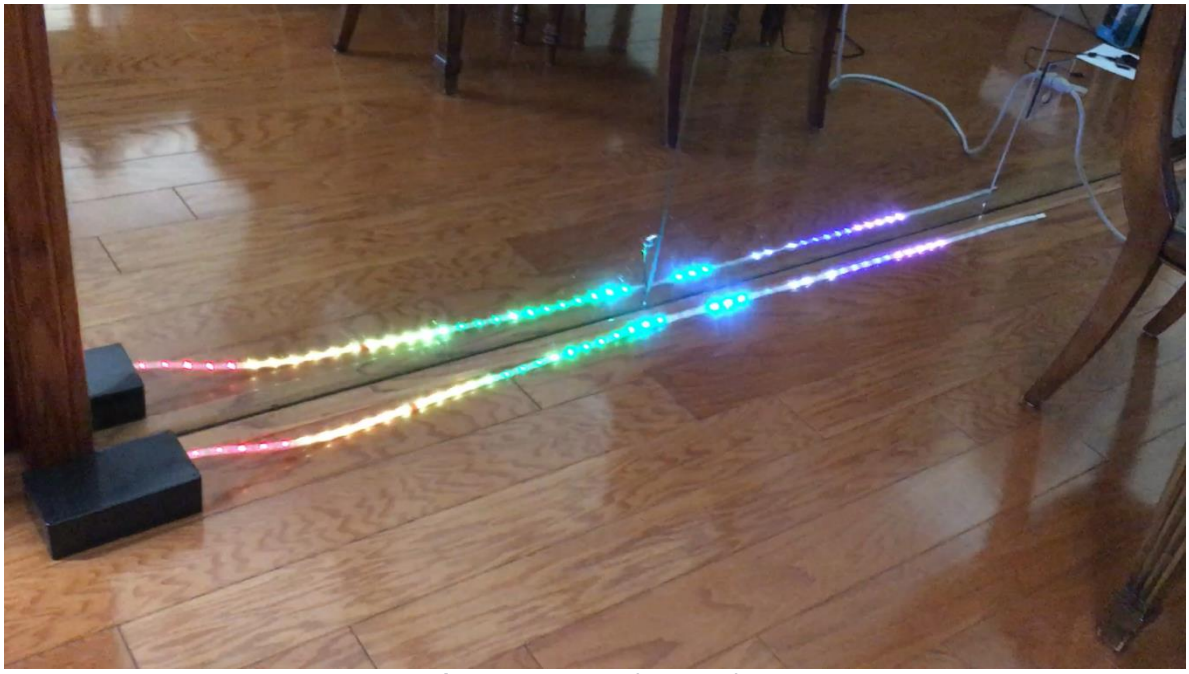

***Figure 10**: System in Operation*

Software

The software of this system consists of three interdependent parts: Processing, Blynk, and ESP8266. In short, Processing analyzes the music and tells the ESP8266 when beats happen, Blynk is used to control parameters in the ESP8266, and the ESP8266 controls how the LED strip is lit.

Processing

Processing is a flexible software sketchbook and a language for learning how to code in the context of visual arts. It is very useful for prototyping. [10]

Processing was chosen because there are similar projects that can be found online that also do audio signal processing using the Minim library. Learning from these other projects gave this project a jumpstart. Processing development is also relatively simple and similar to the development of Arduino projects and that is what also made it an attractive option. Processing also provided an easy way to visualize what the program was doing and that helped a lot of debugging and testing.

Figure 11 below is the software flowchart for the Processing program. Like Arduino, an initial set up function initializes the server and audio analysis tools before the endless main loop. Every iteration through the main loop has an FFT applied to the current 1024-byte sample of audio. A local sound energy average is calculated and if a beat is detected the ESP8266 is notified. If there is no beat the flow goes back to applying an FFT on the next sample of audio and the process repeats.

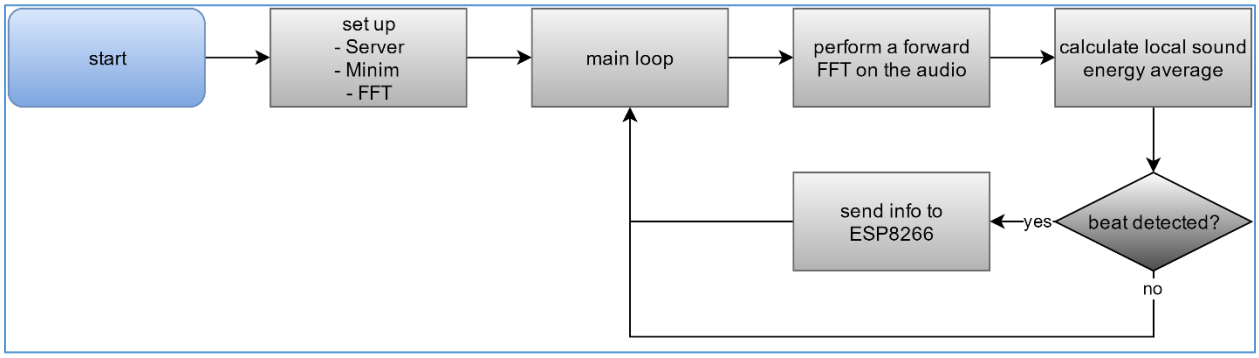

***Figure 11**: Software Flowchart*

Blynk

Blynk is a platform on iOS and Android for controlling devices over the Internet. Its purpose in this system is to provide a means to control the ESP8266 remotely and serve as a user interface.

Figure 12 below is a screen capture of the iOS Blynk dashboard for this project.

The three sliders specify how much of red, green, and blue is to be shown in the lights when the customize option is selected. There are four buttons for selecting between modes 0 through 3. These modes will be discussed in the next section. There is a customize switch button that can be toggled on and off for allowing the user to specify the RGB values. The small beam prop slider customizes the speed of the beam propagation in mode 2. The mode view showing a 1 in this example shows what the device is currently visualizing and in this case it is mode 1.

***Figure 12**: iOS Blynk Screen Capture*

The Blynk app provides an authentication token that is specific to this project and needs to be specified in the firmware.

ESP8266

The ESP8266 has an optional Arduino IDE, which is a familiar environment, and that was the development environment used in this project. Like most embedded systems, there is no end to the firmware flow. It is always checking if data was received from the server, Processing, and if there is, the visualization being showed depends on the mode selection flag set by Blynk.

The description of each mode goes as follows:

Mode 0 – bar flashes: The entire LED strip flashes a random or specified color on a bass beat. It provides an interesting visualization if the song being played has very pronounced and distinct “thumps” in the bass part. Since it only looks at the lower end of the sound spectrum, it is inactive when there is a lack of “thumps”.

Mode 1 – spectrum: The entire LED strip is used to represent activity in the different parts of the spectrum. The colors span from red to violet representing low to high frequency ranges respectively. It is appropriate for all kinds of music and sound in general. It looks at the entire spectrum and visualizes any beat detected.

Mode 2 – beams: A beam is propagated across the LED strip with a random or specified color on a bass beat. It interesting visualization because it propagates a beam on “thumps” but the beam remains visible as it travels. It is especially interesting if the LED strip is not in a straight line.

Mode 3 – mini bar flashes: This is a derivative of Mode 0. Parts of the LED strip flashes a random or specified color on a bass beat, mid beat, and high beat. It solves the inactivity problem of Mode 0 thus making it appropriate for a broader range of music.

Figure 12 below is the firmware flowchart for the ESP8266. An initial setup function is run only once and it initializes the Blynk app connection, the client, and the LED strip. Inside the main loop, the first thing it does is check whether the Wi-Fi client successfully established a connection with the server. If not, the program does nothing. If successful, the program checks if data is available from the server. If not, the loop ends and it polls until data becomes available. Once data is available, beats are visualized according to the currently selected Mode. The default Mode is 0 and that can be changed with Blynk. Mode 2, the beam propagation visualization, is the only mode that does not require any dimming. The flow repeats and awaits the next data available from the server.

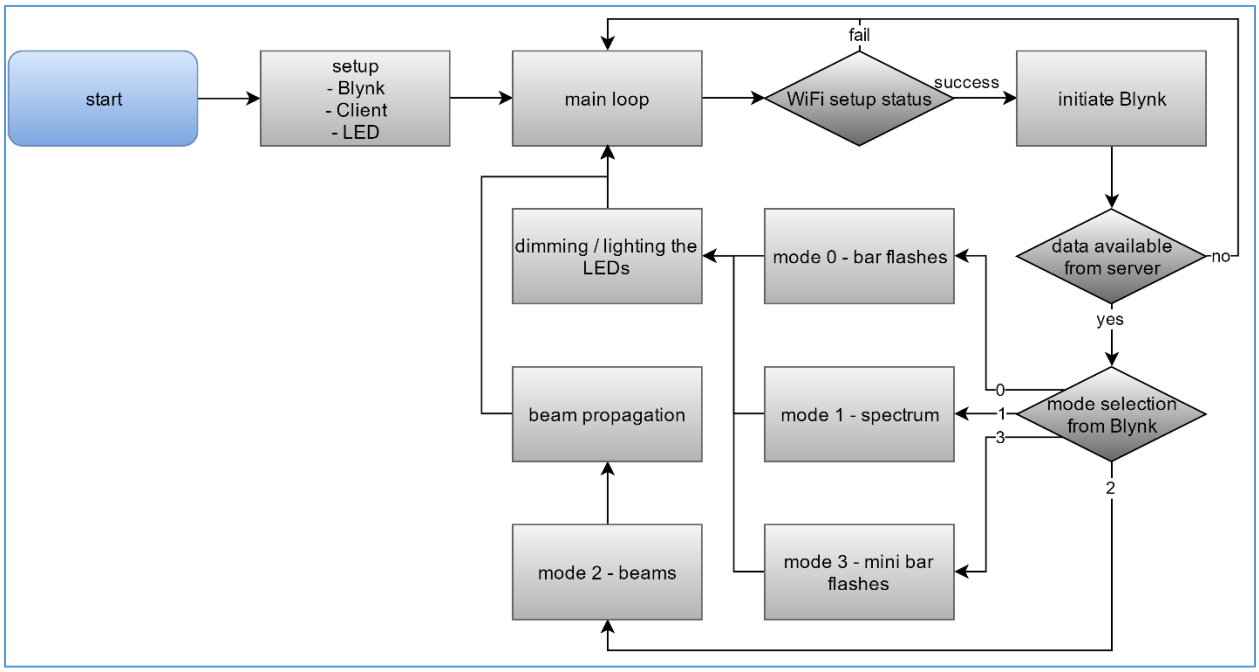

***Figure 13**: Firmware Flowchart*

Testing

To test if the ESP8266 was functional, there is an example provided by the Internet that simply blinks the onboard LED on and off. The test consisted of uploading the code onto the board and visually inspecting for the blinking LED. It was successful.

To test if the LED strip was functional, there is an example provided by the Internet that simply propagated a beam through the LED strip. The test consisted of uploading the code onto the board and visually inspecting for the light beam going across the strip. It was successful.

To test if the Wi-Fi connection was properly established between the ESP8266, PC, and Blynk, there are examples for setting up the server and client between the board the PC provided by Processing and Arduino, respectively. The board would print to the serial port a message saying whether a successful connection was made to the server and the Processing application would print in the system output whether a successful connection was made with the client. Blynk provides an example to toggle the onboard LED with a virtual button on the dashboard. These tests were successful.

To test if the system reacts to music in real-time over Wi-Fi, music was played on the server side and the LEDs were visually inspected personally and judged by peers. The peer reviews were mostly positive but some had trouble following the beat. Objectively, in the broadest sense the system certainly reacts to music in real-time over Wi-Fi, but subjectively, how visually entertaining and impressive the system does so is to be determined by the user.

To test if the system runs until the user turns it off, the system was simply left running for an extended period of time. This requirement is decidedly not met because the ESP8266 Wi-Fi connection with the network is only up for an indefinite amount of time. One hypothesis is that the board requires more time to run background maintenance routines, such as Wi-Fi upkeep, than the current firmware allows. The problem is that the more time is allowed for background routines, the less real-time the system becomes and for a music related application, the real-time property is paramount.

To determine if the system is portable, the system needs to fulfill the requirement that it is not tethered to a PC or a power socket. The system works wirelessly over Wi-Fi and is powered by a USB power bank. The limitation is that the board and the PC must be on the same network.

Below is a table that summarizes the tests and requirement results.

Table 2 *- Design Requirements*

Specification	Design Requirement	Met / Not Met
2	The system needs to receive beat information via WiFi from a device hosting a TCP server on the same network.	Met
2,4	The system reacts to music in real-time over Wi-Fi.	Met
3	The system needs to sustain at least 4 hours of typical use.	Not Met*
1	The system size excluding the LEDs is equal to or less than (191 x 110 x 57) cm^3 and weighs equal to or less than 550 grams.	Met
5	The system requires at least 4 different visualization modes.	Met
6	The system cost cannot exceed $100.	Met
7	The system uses off-the-shelf components from Adafruit and SparkFun.	Met

*The ESP8266 cannot last 4 hours without having to restart. It is suspected that the board requires more time to run background maintenance routines, such as Wi-Fi upkeep, than the current firmware allows. The problem is that if more time is allowed for background routines, the system becomes less real-time, and for a music related application the real-time property is paramount.

Conclusion and Future Work

The design works rather well. According to the specifications and requirements that were set out, it only failed to meet one and subjectively meets two others. Because the goal of the project is to entertain, which makes the goal inherently subjective, it ultimately depends on the end user whether they like the results or not. Personally, the results are satisfactory given the nine weeks of time that was spent developing the project.

However, it should be noted that many improvements can still be made. The current system requires two separate parts working together: the client (board) and the server (PC). It would be ideal for the two parts to be in one comprehensive system. It depends on the specific application, but many use cases could benefit from the project being just one comprehensive system. The client part is also not currently one hundred percent reliable because of its indefinite runtime as discussed earlier. One caveat of this two part system is that it requires a shared network. However, nowadays wireless networks are widely available and even if one cannot be found, a typical smartphone can provide a personal hotspot as well.

Although not currently in use, there is some logic and ideas for a future implementation dealing with the average spectrum sound energy and the max spectrum sound energy. This would add more modes of visualization. The project could also benefit from having its own mobile app instead of using Blynk. If the system becomes one comprehensive system, it could be taken on-the-go to music festivals or friends' houses without the need to install the signal processing program on a PC. The beat detection algorithm is currently rather basic too. Another improvement would be creating an algorithm that could adapt to different kinds of songs.

References

[1] Barbosa, Lucas. (2016). Time and Frequency Domains of a Function. Retrieved from https://en.wikipedia.org/wiki/File:Fourier_transform_time_and_frequency_domains.gif

[2] Evite. (2016). How Long Should a Party Run? Retrieved from https://ideas.evite.com/etiquette/how-long-should-a-party-run/

[3] Godard, T. (2015.08.12). The Economics of Electronic Dance Music Festivals. Retrieved from https://smartasset.com/insights/the-economics-of-electronic-dance-music-festivals

[4] Jimbo. (2016). ESP8266 Thing Hookup Guide. Retrieved from https://learn.sparkfun.com/tutorials/esp8266-thing-hookup-guide/

[5] Kachur, M. (2016). ViVi – Music LED Controller. Retrieved from https://visualvibes.io/

[6] Krumpus, M. (2016). Lumazoid – Realtime Music Visualizer Board for LED strips. Retrieved from https://nootropicdesign.com/lumazoid/

[7] Patin, F. (2003). Beat Detection Algorithms. Retrieved from http://www.flipcode.com/misc/BeatDetectionAlgorithms.pdf

[8] Peoples, G. (2015.05.22). Global EDM Market Hits $6.9 Billion. Retrieved from http://www.billboard.com/articles/business/6575901/global-edm-market-hits-69-billion

[9] Pigeon, S. (2016). The Nonlinearities of the Human Ear. Retrieved from http://www.audiocheck.net/soundtests_nonlinear.php

[10] Processing (2016) Home Page. Retrieved from https://processing.org

[11] Stuts, C. (2014.06.19). The Average American Listens to Four Hours of Music Each Day. Retrieved from http://www.spin.com/2014/06/average-american-listening-habits-four-hours-audio-day/

[12] Wikipedia (2016). Hearing Range. Retrieved from https://en.wikipedia.org/wiki/Hearing_range

Appendix A: Senior Project Analysis

Summary of Functional Requirements

The project takes audio, specifically music, as an input and visualizes patterns on a strip of LEDs as the output. The project is a two-part system: visualizer and signal processor. It provides for entertainment and enhances festive atmospheres in an indoor environment.

Primary Constraints

- There are many different types of music, and different visualizations suit some better than others. A challenge was to create a generic visualization that would be acceptable for all types of music.
- One constraint is that the project requires signal processing to be done on a separate machine. This limits the portability of the system.
- Communication between the development board and the signal processing machine is limited to one byte per message. This requires a protocol to convey more complicated messages such as the the sound intensity per frequency range.

Economic

Bill of Materials

***Table 3**: Bill of Materials*

Item	Price
Adafruit DotStar LED strip (30 LEDs)	$20
Sparkfun Logic Level Converter	$3
Sparkfun FTDI Basic Breakout 3.3V	$15
Sparkfun ESP8266 Thing	$16
Stackable Pin Headers 10 (2-count)	$1
Breadboard Jumper wires	$12
Breadboards	$11
Adafruit USB micro-B breakout board	$5.83
Anker battery 5200 mAh	$10
Original Estimated Cost	$78
Actual Final Cost (original + added items)	$93.83

Equipment and tools needed for development include a computer running Arduino IDE, iOS or Android Blynk app, and a soldering iron for some assembly.

***Table 4**: Development Time*

Original Estimated Development Time	17 weeks
Actual Development Time	10 weeks

Manufacturing

***Table 5**: Manufacturing*

Estimated number of devices to be sold per year	50
Estimated manufacturing cost for each device	$100
Estimated purchase price for each device	$120
Estimated profit per year	$1000
Estimated cost for user to operate device, per unit time	$0.00108 per hour

The cost of electricity is calculated by noting that the each LED on the strip draws 60mA. 30 LEDs would draw 1.8A. The LEDs are powered at 5V so this makes 9W. According to NPR, the average price people in the U.S. pay for electricity is $0.12 per kilowatt-hour.

$$\frac{\$0.12}{1000\,W*hr} * \frac{60mA}{LED} * 5V * 30LED = \frac{\$0.00108}{hr}$$

Manufacturability

The project is as manufacturable as any relatively simple custom PCB as all components are sourced as off-the-shelf products.

Environmental

Any disposal of this project would contribute to electronic waste.

Sustainability

- The battery needs to be recharged when depleted.
- The projects draws ~60mA per LED so 30 LEDs draw ~1800mA.
- The current design requires all purchasable parts. The project can be turned into a custom PCB.
- PCB design and fabrication knowledge is necessary.

Ethical

One goal of the project is to enhance festive atmospheres but it does not encourage or condone irresponsible or illegal activities.

Health and Safety

The project involves a little bit of soldering so all necessary precautions associated with soldering should be taken. The LEDs can be really bright so it is advised that users do not look directly at the lights for a prolonged amount of time or within close proximity.

Social and Political

The project is anticipated to be well received by individuals who enjoy music and the various associated cultures.

Development

Knowledge regarding how humans hear music and music analysis in the frequency domain was learned independently. Signal analysis techniques such as moving average and moving maximum were introduced in STATS-350/EE-228/CPE-464 and implemented in this project for the first time.

Appendix B: Source Code

Firmware

```
/*
 * Student: Peter Chu
 * Advisor: Dr Bridget Benson
 *
 * Senior Project: musicLED
 * Description:
 *  This project visualizes a strip of 60 RGB LEDs (APA102C)
 *  to music information delivered by the accompanying
 *  Processing app. It uses Blynk on iOS/Andorid to adjust
 *  the various modes and settings of the application.
 */

/*************************************************************
 * Blynk is a platform with iOS and Android apps to control
 * Arduino, Raspberry Pi and the likes over the Internet.
 * You can easily build graphic interfaces for all your
 * projects by simply dragging and dropping widgets.
 *
 *   Downloads, docs, tutorials: http://www.blynk.cc
 *   Blynk community:            http://community.blynk.cc
 *   Social networks:            http://www.fb.com/blynkapp
 *                               http://twitter.com/blynk_app
 *
 * Blynk library is licensed under MIT license
 * This example code is in public domain.
 *
 *************************************************************
 * You need to install this for ESP8266 development:
 *   https://github.com/esp8266/Arduino
 *
 * Please be sure to select the right ESP8266 module
 * in the Tools -> Board menu!
 *
 * Change WiFi ssid, pass, and Blynk auth token to run :)
 *
 *************************************************************/

#include <Adafruit_DotStar.h>
#include <SPI.h>
#include <ESP8266WiFi.h>
#include <BlynkSimpleEsp8266.h>
#include <ESP8266WiFiMulti.h>

// constants
#define NUMPIXELS 60        // number LEDs on the strip
#define ESP8266_LED 5       // ESP8266 on-board LED on port 5
#define MINIBAR_LEN_TREB 7  // len of small bar in mode 3
#define MINIBAR_LEN_BASS 30 // len of mid bar in mode 3
#define MINIBAR_LEN_MIDL 15 // len of mid bar in mode 3

#define DATAPIN   2 // GPIO2 - MOSI
#define CLOCKPIN  4 // GPIO4 - CLK

#define IP_ADDR "192.168.1.130"       // IP address of pc running Processing app
#define SSID_NAME "peter"           // SSID of the network in use
#define SSID_PW "peteriscool"       // password to the network in use

// WiFi
ESP8266WiFiMulti WiFiMulti;

// Use above defined pins for DATA and CLK
Adafruit_DotStar strip = Adafruit_DotStar(NUMPIXELS, DATAPIN, CLOCKPIN);

// Use hardware SPI (DATA-13, SCLK-SCL)
//Adafruit_DotStar strip = Adafruit_DotStar(NUMPIXELS);
```

```
// You should get Auth Token in the Blynk App.
// Go to the Project Settings (nut icon).
char auth[] = "62999818c539415093b705f3be062d70";

// global variables
bool setupFailed = false; // flag for WiFi setup. if failed, blink onboard LED
WiFiClient client;        // create TCP connection
byte val = 0;             // val received from Processing app (beat in val bin)

// LED controlling variables
unsigned int r[NUMPIXELS] = {0}; // each index holds the R, G, or B value
unsigned int g[NUMPIXELS] = {0}; // of the LED with that index ranging
unsigned int b[NUMPIXELS] = {0}; // from 0 - 255

int heads[NUMPIXELS] = {-1};   // array holding the heads and tails values
int tails[NUMPIXELS] = {-1};   // of the "beams" that go across the LED strip
int beamCnt = 0;
int beamDelay = 0;      // counter that increments and every...
int beamPropDelay = 4;  // ...beamPropDelay it updates the beam
int beamLen = -7;       // length of beam in LEDs

// Blynk variables
int customize, modeSel;         // flags set by Blynk app
int blynk_r, blynk_g, blynk_b;  // r,g,b values set by Blynk app

void setup()
{
  // set up onboard LED mode
  pinMode(ESP8266_LED, OUTPUT);

  // set up Blynk
  Blynk.begin(auth, SSID_NAME, SSID_PW);

  // set up WiFi connection to PC
  const uint16_t port = 5204;
  const char * host = IP_ADDR; // ip or dns

  Serial.begin(115200);
  delay(10);

  // We start by connecting to a WiFi network
  WiFiMulti.addAP(SSID_NAME, IP_ADDR);

  Serial.println();
  Serial.println();
  Serial.print("Wait for WiFi... ");

  while(WiFiMulti.run() != WL_CONNECTED) {
      Serial.print(".");
      delay(500);
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());

  delay(500);

  Serial.print("connecting to ");
  Serial.println(host);

  if (!client.connect(host, port)) {
      Serial.println("connection failed, restart system");
      setupFailed = true;;
  }

  strip.begin();
```

```
  strip.show();

  // random seed
  randomSeed(0);
}

// Blynk functions for virtual pins
BLYNK_WRITE(V0)
{
  modeSel = 0;
}
BLYNK_WRITE(V1)
{
  modeSel = 1;
}
BLYNK_WRITE(V2)
{
  modeSel = 2;
}
BLYNK_WRITE(V3)
{
  modeSel = 3;
}
BLYNK_WRITE(V4)
{
  customize = param.asInt();
}
BLYNK_WRITE(V5)
{
  blynk_r = param.asInt();
}
BLYNK_WRITE(V6)
{
  blynk_g = param.asInt();
}
BLYNK_WRITE(V7)
{
  blynk_b = param.asInt();
}
BLYNK_WRITE(V8)
{
  beamPropDelay = param.asInt();
}
BLYNK_READ(V9)
{
  Blynk.virtualWrite(9, modeSel);
}

void loop()
{
  int i;                // counter used for misc purposes
  int gVal, rVal, bVal; // temporary holder variables for RGB values

  // blink onboard LED if wifi failed
  if (setupFailed == true)
  {
    digitalWrite(ESP8266_LED, HIGH);
    delay(1000);
    digitalWrite(ESP8266_LED, LOW);
    delay(1000);
    return;
  }

  Blynk.run();  // Initiates Blynk

  if (client.available())
  { // if data is availble to read
    val = client.read();
```

```
//////////////////// mode 0 ////////////////////
if ( modeSel == 0)  // bar flash mode
{
  if (!customize)   // custom colors
  {
    gVal = random(0x100);
    rVal = random(0x100);
    bVal = random(0x100);
  }
  else  // use sliders on Blynk app to determine color
  {
    gVal = blynk_g;
    rVal = blynk_r;
    bVal = blynk_b;
  }

  if (val == 10) // detecting == bin2 (71 < val < 86 Hz )
  {
    for ( i = 0; i < NUMPIXELS; i++)
    {
      r[i] = rVal;
      g[i] = gVal;
      b[i] = bVal;
    }
  }
}
//////////////////// mode 1 ////////////////////
else if ( modeSel == 1 )
{ // spectrum mode (sub bass, bass, midrange, high mids, high freq)
  if (customize)
  {
    r[val] = blynk_r;
    g[val] = blynk_g;
    b[val] = blynk_b;
  }
  else
  {
    if (val < 5)
    { // red
      r[val] = 0xFF;
      g[val] = 0;
      b[val] = 0;
    }
    else if (val < 10)
    { // orange
      r[val] = 0xFF;
      g[val] = 0xA5;
      b[val] = 0;
    }
    else if (val < 15)
    { // yellow
      r[val] = 0xFF;
      g[val] = 0xFF;
      b[val] = 0;
    }
    else if (val < 20)
    { // chartreuse
      r[val] = 0x7F;
      g[val] = 0xFF;
      b[val] = 0;
    }
    else if (val < 25)
    { // green
      r[val] = 0;
      g[val] = 0x80;
      b[val] = 0;
    }
    else if (val < 30)
    { // spring
```

```
      r[val] = 0;
      g[val] = 0xE6;
      b[val] = 0x73;
    }
    else if (val < 35)
    { // cyan
      r[val] = 0;
      g[val] = 0xFF;
      b[val] = 0xFF;
    }
    else if (val < 40)
    { // azure
      r[val] = 0xF0;
      g[val] = 0xFF;
      b[val] = 0xFF;
    }
    else if (val < 45)
    { // blue
      r[val] = 0;
      g[val] = 0;
      b[val] = 0xFF;
    }
    else if (val < 50)
    { // violet
      r[val] = 0xEE;
      g[val] = 0x82;
      b[val] = 0xEE;
    }
    else if (val < 55)
    { // magenta
      r[val] = 0xFF;
      g[val] = 0x00;
      b[val] = 0xFF;
    }
    else
    { // rose
      r[val] = 0xFF;
      g[val] = 0;
      b[val] = 0xFF;
    }
  }

}
//////////////////// mode 2 ////////////////////
else if ( modeSel == 2 )
{
  if (val == 17 )
  { // trigger the beam
    if (beamCnt >= NUMPIXELS)
      beamCnt = 0;

    heads[beamCnt] = 0;
    tails[beamCnt] = beamLen;

    if (!customize) // every beam gets a random color
    {
      r[beamCnt] = random(0x100);
      g[beamCnt] = random(0x100);
      b[beamCnt] = random(0x100);
    }
    else
    { // using zeRGBa on Blynk app to determine color
      g[beamCnt] = blynk_g;
      r[beamCnt] = blynk_r;
      b[beamCnt] = blynk_b;
    }
    beamCnt++;
  }
}
```

```
////////////////////// mode 3 ////////////////////
else if ( modeSel == 3 )
{ // random small bars (bass, mid, treble)

  // random colors if custom button not pressed on Blynk app
  if (!customize)
  {
    gVal = random(0x100);
    rVal = random(0x100);
    bVal = random(0x100);
  }

  // determine location of bass bar
  i = random(60 - MINIBAR_LEN_BASS);
  if (val == 10)
  {
    if (customize)
    {
      gVal = blynk_g;
      rVal = blynk_r;
      bVal = blynk_b;
    }
    for (int j = i ; j < i + MINIBAR_LEN_BASS; j++)
    {
      r[j] = rVal;
      g[j] = gVal;
      b[j] = bVal;
    }
  }
  // determine location of mid bar
  i = random(60 - MINIBAR_LEN_MIDL);
  if (val == 17)
  {
    if (customize)
    {
      gVal = blynk_r;
      rVal = blynk_b;
      bVal = blynk_g;
    }
    for (int j = i ; j < i + MINIBAR_LEN_MIDL; j++)
    {
      r[j] = rVal;
      g[j] = gVal;
      b[j] = bVal;
    }
  }
  // determine location of treble bar
  i = random(60 - MINIBAR_LEN_TREB);
  if (val == 54)
  {
    if (customize)
    {
      gVal = blynk_b;
      rVal = blynk_g;
      bVal = blynk_r;
    }
    for (int j = i ; j < i + MINIBAR_LEN_TREB; j++)
    {
      r[j] = rVal;
      g[j] = gVal;
      b[j] = bVal;
    }
  }
}
/* // future work - average spectrum energy
else if ( modeSel == 4 )
{ // spectrum avg
  for (i = 0; i < NUMPIXELS; i++)
  {
```

```
        r[i] = val;
        g[i] = val;
        b[i] = val;
        strip.setPixelColor(i, g[i], r[i], b[i]);
      }
    }
    */
  } // end of client available loop

  // mode 2 - beam propagation
  // update head location every beamPropDelay
  // to control beam propagation speed
  if (beamDelay++ % beamPropDelay == 0)
  {
    for (i = 0; i < NUMPIXELS; i++)
    {
      strip.setPixelColor(heads[i], g[i], r[i], b[i]);  // 'On' pixel at head
      strip.setPixelColor(tails[i], 0);                 // 'Off' pixel at tail

      if (heads[i] >= 0)
      {
        if ( ++heads[i] >= NUMPIXELS)  // reset head
          heads[i] = -1;
      }
      if (tails[i] >= beamLen)
      {
        if ( ++tails[i] >= NUMPIXELS )  // reset tail
          tails[i] = beamLen - 1;
      }
    }
  }

  // dimming, all modes except beam (mode 2)
  if ( modeSel == 0 || modeSel == 1 || modeSel == 3 )
  {
    for ( i = 0; i < NUMPIXELS; i++ )
    {
      if (g[i] > 0)
        g[i] /= 1.04;

      if (r[i] > 0)
        r[i] /= 1.04;

      if (b[i] > 0)
        b[i] /= 1.04;

      strip.setPixelColor(i, g[i], r[i], b[i]);
    }
  }
  strip.show();

  delay(5); // give uController time to maintain behind the scenes tasks
}
```

Software

```
/*
 * Student: Peter Chu
 * Advisor: Dr Bridget Benson
 *
 * Senior Project: musicLED
 * Description:
 *  This project visualizes a strip of 60 RGB LEDs (APA102C)
 *  to music information delivered by the accompanying
 *  Processing app. It uses Blynk on iOS/Andorid to adjust
 *  the various modes and settings of the application.
 *
 *  This program analyzes sound from the default recording device of the machine.
 *  It uses the Fast Fourier Transform to see audio activity in certain frequency bands.
 */

/**
  * An FFT object is used to convert an audio signal into its frequency domain representation. This
representation
  * lets you see how much of each frequency is contained in an audio signal. Sometimes you might not want
to
  * work with the entire spectrum, so it's possible to have the FFT object calculate average frequency
bands by
  * simply averaging the values of adjacent frequency bands in the full spectrum. There are two different
ways
  * these can be calculated: Linearly, by grouping equal numbers of adjacent frequency bands, or
  * Logarithmically, by grouping frequency bands by octave, which is more akin to how humans hear sound.
  *
  * This sketch uses the logarithmic average because its application deals with music and its
visualization.
  *
  * Moving the mouse across the sketch will display the center frequency of that band.
  * For more information about Minim and additional features, visit http://code.compartmental.net/minim/
  */

import processing.net.*;
import processing.serial.*;
import ddf.minim.analysis.*;
import ddf.minim.*;
import papaya.*;

// minim
Minim minim;
AudioInput in;
FFT fftLin;
FFT fftLog;

// constants defines
final int avgSens = 75;         // number of averages for calculating running average
final int maxSens = 300;        // number of averages for calculating running maximums
final float spectrumScale = 2;  // scalar to make bars more visible
final int timeSens = 250;       // ms to wait until next beat is valid
final float avgMult = 1.8;      // scales the threshold
final float avgOffset = 4;      // higher number prevents noisy detections
final int bufferSize = 1024;    // for the FFT analysis

// FUTURE WORK
final int spectrumAvgSens = 5;   // number of averages for calculating running spectrum avg
final int spectrumMaxSens = 400; // number of averages for calculating running spectrum max
// FUTURE WORK

// beat finding variables
float runAvgs[][] = new float[60][avgSens];  // array holds ith bin time average value for calculating
threshold
float runMaxs[][] = new float[60][maxSens];  // array holds ith bin time average value for calculating max
int timer[] = new int[60];      // holds the last time a beat was detected for the ith freq band
int counter;                    // used for knowing how many iterations draw has run
byte beats[] = new byte[60];    // array holds beats (info)
```

```
// draw variables
float height3;
float height23;
PFont font;

// FUTURE WORK
float spectrumSum = 0;
float spectrumAvg = 0;
float spectrumAvgMaxRatio = 0;
float runSpectrumAvgs[] = new float[spectrumAvgSens];
float runSpectrumMaxs[] = new float[spectrumMaxSens];
// FUTURE WORK


// network variables
Server myServer;
int bps = 0;
int mils = 0;

void setup()
{
  // canvas
  size(512, 512);
  height3 = height/3;
  height23 = 2*height/3;

  // start myServer on port 5204
  myServer = new Server(this, 5204);

  // minim
  minim = new Minim(this);
  in = minim.getLineIn(Minim.STEREO, bufferSize);

  // create an FFT object for calculating logarithmically spaced averages
  // note that bufferSize needs to be a power of two
  fftLog = new FFT(in.bufferSize(), in.sampleRate());

  // calculate averages based on a miminum octave width of 22 Hz
  // split each octave into 6 bands
  // this should result in 60 averages
  fftLog.logAverages(22, 6);

  rectMode(CORNERS);
  font = loadFont("ArialMT-12.vlw");
}

void draw()
{
  background(0);

  textFont(font);
  textSize( 18 );

  float centerFrequency = 0;

  // perform a forward FFT on the AudioInput in
  fftLog.forward(in.mix);

  // draw the logarithmic averages
  int w = int( width/fftLog.avgSize() );  // indicates where to draw rectangles

  for(int i = 0; i < fftLog.avgSize(); i++)
  {
    centerFrequency    = fftLog.getAverageCenterFrequency(i);
    // how wide is this average in Hz?
    float averageWidth = fftLog.getAverageBandWidth(i);

    // we calculate the lowest and highest frequencies
```

```
    // contained in this average using the center frequency
    // and bandwidth of this average.
    float lowFreq  = centerFrequency - averageWidth/2;
    float highFreq = centerFrequency + averageWidth/2;

    // store values into array of -Sens size
    // altering -Sens values alters the sensitivity of the running average & max
    // by controlling the number of values used for calculating the running average & max
    runAvgs[i][counter % avgSens] = (int)fftLog.getAvg(i)*spectrumScale;
    runMaxs[i][counter % maxSens] = (int)fftLog.getAvg(i)*spectrumScale;

    // draw the maximum rect
    // maximum = mean + 3 standard deviations
    fill(0, 255, 0);
    rect(i*w, height, i*w + w, height - (Descriptive.mean(runMaxs[i]) + 3 * Descriptive.std(runMaxs[i],
true)));

    // draw the threshold rect
    // instantaneous value in ith bin must exceed the threshold value to count as a beat
    // treshold = mean + 1 standard deviation
    fill(0, 0, 255);
    rect(i*w, height, i*w + w, height - (Descriptive.mean(runAvgs[i]) + 1 * Descriptive.std(runAvgs[i],
true)));

    // if ((inst pwr > threshold) AND (sensitivity timer is up)), send beats to client
    //if (fftLog.getAvg(i)*spectrumScale > avgs[i] +  1 * Descriptive.std(runAvgs[i], true) + avgOffset) {
    if (fftLog.getAvg(i)*spectrumScale > Descriptive.mean(runAvgs[i]) +  1 * Descriptive.std(runAvgs[i],
true) + avgOffset) {
      if (millis() > timer[i] + timeSens) {
        timer[i] = millis();      // reset timer
        beats[i] = (byte)i;       // mark beat in ith frequency band
        myServer.write(beats[i]);
        ++bps;
      }
    }

    // if the mouse is inside of this average's rectangle
    if ( mouseX >= i * w && mouseX < i*w + w)
    {
      fill(255, 255, 0);
      text("Log Avg Center Frequency: " + centerFrequency + " bin " + i, 5, 0.1 * height);
    }

    // draw a rectangle for each average, multiply the value by spectrumScale so we can see it better
    fill(255);
    rect(i*w, height, i*w + w, height - fftLog.getAvg(i)*spectrumScale);







    // FUTURE WORK -- calculate instant spectrum sum and avg
    spectrumSum += fftLog.getAvg(i) * spectrumScale;
    //println("bin " + i + " lowFreq = "+ lowFreq + " hiFreq = " + highFreq);
    // FUTURE WORK

    if (millis() >= mils + 1000)
    {
      mils = millis();
      println("bytes per second" + bps);
      bps = 0;
    }
  }
```

```
  // FUTURE WORK -- spectrum-wide analysis
  // proportion to 255 for LED brightness
  spectrumAvg = spectrumSum / 50;
  runSpectrumAvgs[counter % spectrumAvgSens] = spectrumAvg;
  runSpectrumMaxs[counter % spectrumMaxSens] = spectrumAvg;
  spectrumAvgMaxRatio = Descriptive.mean(runSpectrumAvgs) / (Descriptive.mean(runSpectrumMaxs) + 3 *
Descriptive.std(runSpectrumMaxs, true)) * 255;
  // draw spectrum max
  fill(255, 0, 0);
  rect(59*w, height, 60*w, height - (Descriptive.mean(runSpectrumMaxs) + 3 *
Descriptive.std(runSpectrumMaxs, true)));

  // draw spectrum avg
  fill(255, 255, 0);
  rect(59*w, height, 60*w, height - Descriptive.mean(runSpectrumAvgs));
  if (spectrumAvgMaxRatio > 255) {
    //println("spectrum avg/max ratio = 255");
    //myServer.write(255);
  }
  else {
    //println("spectrum avg/max ratio = " + spectrumAvgMaxRatio);
    //myServer.write((byte)spectrumAvgMaxRatio);
  }
  spectrumSum = 0;
  // FUTURE WORK




  counter++;  // marks another iteration through draw
  delay(15);  // delay to let ESP run
}

void serverEvent(Server someServer, Client someClient) {
  println("We have a new client: " + someClient.ip());
}
```