Visualize the Invisible

Beverly Yee
College of Engineering
University of Utah
bevyujw@gmail.com

Alex Gray
College of Engineering
University of Utah
agray0659@gmail.com

***Abstract*—The modern world offers a variety of high-tech devices on which we can listen to music–MP3 players, phones, Bluetooth speakers, to name a few. Most of those devices, however, cannot provide visual stimulation to go alongside the audio stimulus. The first solution to this was a visualizer that reacted to music's intensity/volume via patterns followed by one that was simply a single column of lights. Neither of these solutions was stimulating enough, leading to the development of a new solution–the music frequency visualizer. In this method, small sections of frequencies are analyzed to create multiple columns of visualizations.**

The goal of this project is to create an indoor water show like that at the Bellagio Hotel to enhance the user's music-listening experience. However, unlike the preset patterns and set show times of the Bellagio, real-time calculations will be performed to generate a unique visual response based on an analysis of the music being played. This report will describe the materials and knowledge required for the project, followed by the process, learning curve, and final product.

I. Introduction & Motivation

In the beginning, music visualizers began as sheet music as a method of helping musicians perform [4]. Now, in the 21st century, music visualizers have become synonymous with software that "captures data from a music audio file" and creates colors, shapes, and images based on that data [7]. In a way, they allow listeners to "see" the songs they're listening to, which is especially important for those who are deaf or hard of hearing [3]. Since they cannot or have difficulties hearing, it takes away the experience of music. Through visualization, they could gain an experience similar to hearing audiences.

The first visualizers–electronic visualizers–were created by Robert Brown in 1976 [6], the same person who created the game Pong. At first, these electronic music visualizers were primarily used in video games. As the years passed, the visualizers transitioned from video games to personal computers. An older example of one such version would be the Windows Media Player 7, rolled out in the mid-2000s-[5].

Today, visualizers appear on the internet as freeware–that is, free software or open source–for the public to make as their own and exist in a vast variety. On the computer side, this software is mainly used to create music videos for platforms such as YouTube, Facebook, or Twitter. There are also hardware-based music visualizers–DJs use them in clubs to create lighting effects on the dance floor.

A. *Motivation*

The idea of this project is mainly influenced by the art background of the two collaborators. Initially, the goal was to create a visualizer with just LEDs, mimicking pumping bars from a software visualizer or similar effects. Since lights alone may not have enough of an impact, the concept of pulsing water columns is added.

There are already water LED speakers available commercially but with only 4 pumps and a set pattern. On a much larger scale, there is the fountain at the Bellagio Hotel in Las Vegas, Nevada. It has many more pumps to create a more intricate dance; however, the dances are all preset patterns and at scheduled times, and extremely not portable.

The final vision is to recreate a fusion of the previously mentioned water speakers and the Bellagio fountain. The main focus is the "water show" at the center, consisting of 12 pumps surrounded by a strip of LEDs aimed to enhance the water columns. A speaker sits on either side of the show such that the user can hear the music the water is dancing to with an audio input coming out from underneath one of the speakers. All circuits and controls are encapsulated in a box that keeps everything together in one piece for portability.

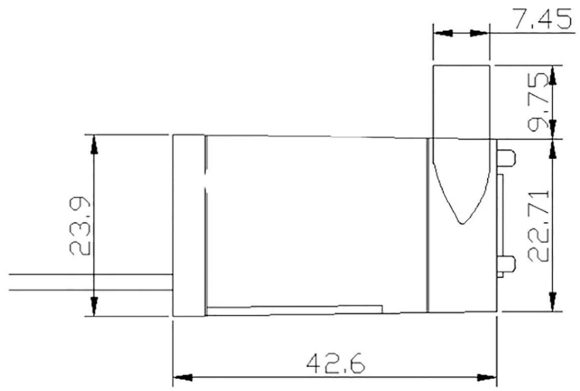
To put it simply, the mission is this: The Bellagio's mini-me: a dance of water fountains to the beat of any chosen musical piece and is highlighted by a spectrum of LEDs. The height of the water and the intensity or color of the LEDs are determined by the frequency/pitch of the music. Visualize the Invisible.

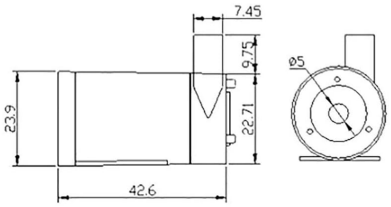
II. Project Implementation

The construction of the project is composed of several parts done incrementally or, wherever possible, concurrently. As the main focus, the operation of the pumps is considered first. It is followed by the construction of the body of the fountain, a container made up of 4 acrylic trays that house the pumps, the pool, and the LEDs; the 3D print of the encapsulation to hold all parts of the project together; software implementation; and final testing. The following sections detail the process for the individual parts and how they all are tied together.

A. *Pumps & Pump PCB*

The pumps used for this project are typically used for small aquariums, ponds, water gardens, or other places in which control over water flow and direction is needed. They operate at a minimum of 3 volts and can handle upwards to 5 volts. The varying voltages between the thresholds allow control over the height of the water–the lower the voltage, the lower the water stream. Conversely, the higher the voltage, the higher the water stream (Fig 1).





Model	Voltage Scope (DC)	Current (A)	Power (W)	Max Water Head (M)	Max Flow Rate (L/H)	Starting Voltage	waterproofing grade
JT-DC3W-3	3V	0.12	0.36	0.35	80	1	IP68
JT-DC3W-4.5	4.5V	0.18	0.91	0.55	100	1	IP68

Fig. 1. Dimension and operation data on the pumps used.

The values to determine the voltage to use are received from the microcontroller, based on the frequencies on the audio given.

16 pumps, 4 sets of 4 pumps, were acquired, but only 12 of them were used in the final assembly. The hope was to use all 16 for better resolution or representation of the frequencies. The number was reduced to 12 due to the limit of PWM (pulse-width modulation) pins on the microcontroller first selected to use for the project. By the time the final microcontroller was selected, the PCB for 12 pumps was already made and assembled.

Each set of pumps came with a meter-long, clear tube to slip over the spout of the pump. A little over 5" of tubing is cut from these pieces for each pump. Unfortunately, the inner diameter of the tubing is slightly smaller than the outer diameter of the spout. To increase the inner diameter of the tubing, a heat gun is used to soften the plastic.

Additional wires are soldered onto the pump wires to increase the length. The pumps will be placed close to the outer edge of the container with the wires coming out from the center. The original length would not be long enough to clear the bottom of the container and give enough room for manipulation. For security, heat shrink tubing is added over the solder joint.

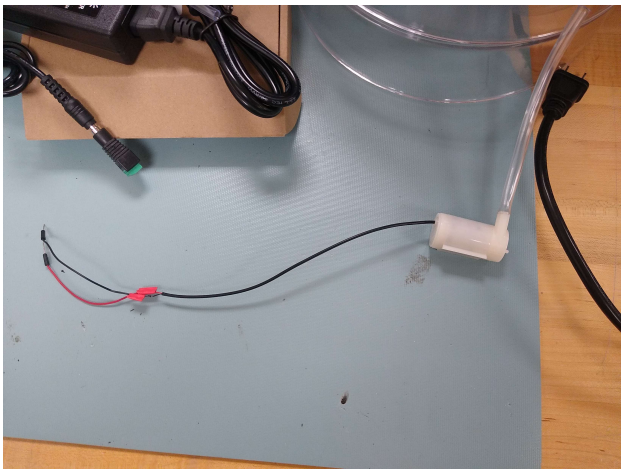


Fig. 2. A pump with the wire extension and tube.

*1) **Operating Circuit**:* Pumps operate nearly identically to motor drivers. Although simple pump schematics are sparse, motor driver schematics are plenty. The one used as a reference for this project is found in this YouTube video. It details a simple circuit for a single motor driver, consisting of an N-channel MOSFET, a switch, and a fly-back diode (Appendix E.3).

In the version of the circuit used for the project, the NMOSFET takes the PWM signal from the microcontroller and averages the voltage based on the duty cycle of said signal. Using this average, the NMOSFET opens its drain to allow that specific voltage as a direct current to the pump. The fly-back diode in the circuit is used to protect the NMOSFET from a sudden spike in voltage when power to the pumps is shut off.

Initially, only one circuit was made and hooked up to a singular pump to make sure the pump worked with an STM32 Discovery board as the test microcontroller. Out of the 16 pumps received, 2 of them didn't work immediately. The remaining 2 were set aside as extras in case another fails further down the line.

After the initial test, a potentiometer replaces the switch as input to act as a substitute for the varying signals from the microcontroller. Once the circuit and all pumps are verified to be working, the pump is swapped out with an LED. The manufacturer does not recommend running the pumps for too long, especially out of water, and LEDs can simulate the pumps.

*2) **Pump PCB**:* To control all 12 pumps, this simple circuit is duplicated 12 times on a single PCB (printed circuit board) (Appendix E.4). This way, the pumps can share a single ground and power source; keeping in mind, of course, that the power source needs to have enough power and amperage to give 5V for each pump to run at its full potential. There is an additional ground on the PCB to connect it to the ground of the microcontroller–this is key in making sure everything works together.

Manufacturing of the PCB comes with 2 extras in case of failure. The final size is 4.6" x 3", roughly the size of an index card (Fig 3).

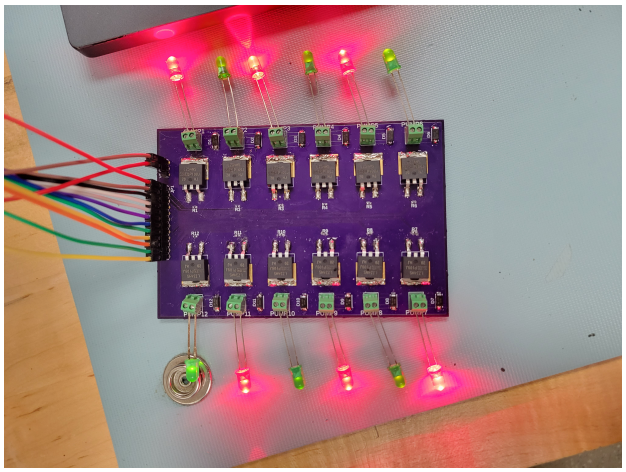


Fig. 3. Finished PCB with LEDs as substitutes for pumps.

Assembly of the PCB required solder paste for the surface mount components, mainly for the resistors. In the absence of the reflow oven, equipment that makes soldering surface mounts a breeze, a heat gun is used instead. To do this, the heat of the heat gun is to be positioned directly over the part(s) being soldered and left there until the solder bubbles coalesce. If the heat gun is placed at an angle, the part is in danger of being blown off the board.

Fear of damaging the board through excessive heat and inexperience made soldering the 1k resistors difficult. While the resistors did work, they do not look very pretty and there are bits of solder here and there. Also, at least one resistor was lost due to being blown off by the heat gun. Another simply went missing.

B. Clear Container

As mentioned previously, the main focus of the project consists of 4, clear, acrylic trays that house the 12 pumps, hold the water for the pumps, and the LED strip to highlight the water. The 4 trays were chosen since initial searches for clear containers resulted in products that were too small or too expensive ($100+ minimum!) for the right size.

The trays are 2.5" tall by 10" wide and manufactured by a florist shop located in Florida. 3 of them are laser-cut and glued together to make one, solid container. The final, uncut tray is used as the lid so water cannot splash outside of the container. As an added safety measure, a sheet of clear vinyl is wrapped around the container. It also acts as a holder for the lid.

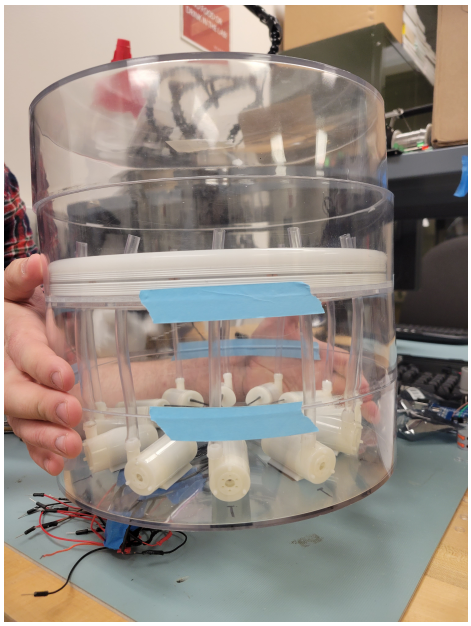


Fig. 4. Dry fit of the containers, pumps, and LED. Here, the Pool is already glued to the Stage. The pumps are also already taped to the bottom of the Pool with the tubes attached and routed through the holes of the Stage. The vinyl to surround the container is added much later.

The container was worked on after the assembly of the PCB. Because there were no other items ready to add to the project, this was not worked on concurrently with any other parts.

*1) **Pool**:* The bottom-most tray is dedicated to holding the pumps.

Each pump is taped to the bottom of the tray, roughly 1" away from the edge so that the inlet has plenty of room to suck in water. The adhesive used is a double-sided, waterproof tape made by Scotch. All pumps are placed equally around the tray with the wires facing towards the center, where a 1" hole is cut via laser-cutter.

Before the wires are threaded through the hole, a thick bead of clear silicone is added to the edge of the hole. The wires are pressed into the silicone enough to create a watertight seal between it and the tray edge. More silicone is added to fill in the hole and left for 2 days to cure completely.

*2) **Stage**:* The second tray from the bottom. The tray top is glued to the Pool tray top with E6000 industrial adhesive and aligned as much as possible with clear, waterproof duct tape made by Gorilla. The bottom of the tray, which is also

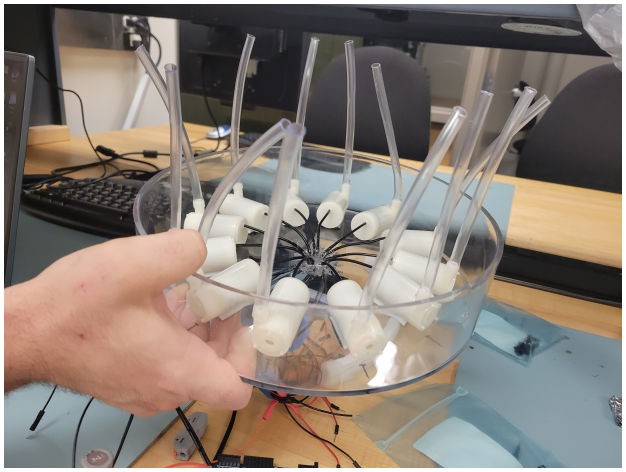


Fig. 5. All pumps secured into place with tubes added to their spouts. The center hole is filled with a large amount of silicone to make sure no water leaks out.

the stage top, has been cut out except for an approximately 1" strip on the edge.

12 holes are drilled into that 1" strip, evenly spaced around the tray and away from the edge enough for the LED strip to sit (roughly 0.5" or 12mm). An additional hole is drilled at the "back" of the container, between two pump holes, for the LED wires to go into the pool area. A final hole is made on the side of the stage tray to take those same wires out of the pool area. Routing the LED wires this way makes it so that water doesn't have easy access out of the container from the stage. Fig 6 shows the Stage tray with all the cuts and holes done and ready to adhere to the Pool tray.

*3) **Stage "Curtain"**:* The third tray, second from the top. It is connected to the Stage. The entire bottom of the tray has to be cut out to give the illusion the container is somewhat of a solid piece. To start, the laser cutter is used to carve out all but roughly 1cm of the bottom. Then, the Dremel is used to sand down the remainder of the bottom, as close as possible to the sides without damaging it.

Just like with the Stage and the Pool, the Curtain is attached to the stage with 36000. Unlike the first two trays, however, no additional tape is added due to the hole needed to feed the LED wires. Painter's tape is used to keep the alignment as perfect as possible.

*4) **Lid**:* The topmost tray. A water-repelling spray is used on the inside of this tray in an attempt to solve the condensation issue faced during the early stages of testing the entire system. It lessened the water buildup on the tray, but the condensation still persists.

*5) **Outer "Curtain"**:* A sheet of clear vinyl matches the height of the container and wraps tightly around it with a bit of overlap. The purpose of the vinyl is to act as another layer of water protection. It also creates a space where the Lid can slip into, allowing it to be held into place without additional hardware like hinges and screws.

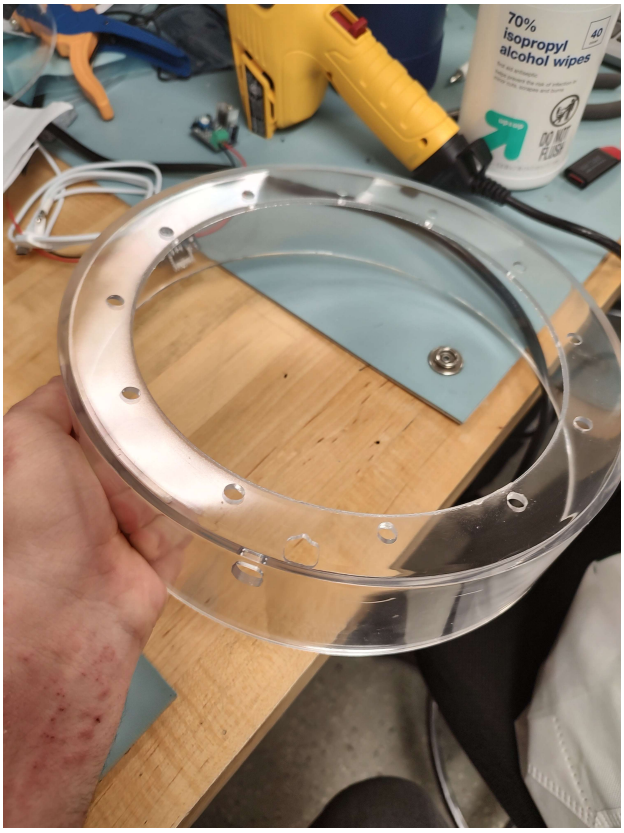


Fig. 6. Stage with center cut and holes drilled for pump tubes and LED wires.

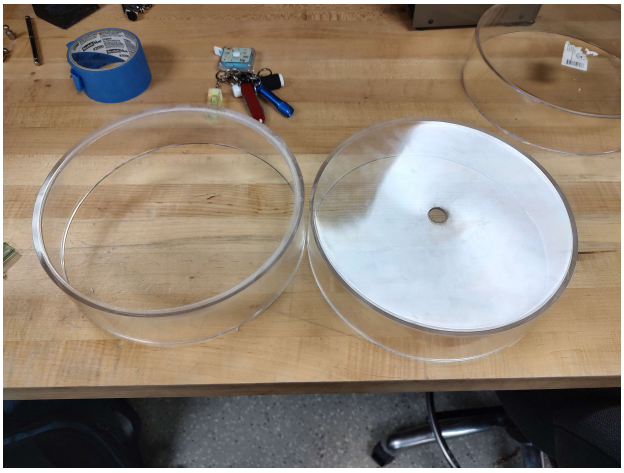


Fig. 7. The Curtain (left) and the Pool (right) after laser cutting.

C. Microcontroller

The first microcontroller considered for the project was the STM32 Discovery Board (STM32f072RBx) due to its familiarity from a previous course (ECE 5780: Embedded

Systems) and the strength of its processor. The only downside was the max of 12 pins capable of PWM generation, hence the final decision to use 12 pumps instead of the original 16. However, since the LEDs are normally powered and programmed by Arduino, the decision was made to switch to the Arduino Uno.

Unfortunately, the Arduino Uno does not have nearly enough PWM pins for the pumps, which means an additional PWM driver board would have been needed. Before testing everything together to see if they work in tandem, it is discovered that the Uno is not powerful, nor quick, enough to handle the real-time computation of an FFT at higher than 8-bit resolution.

The third and final microcontroller is the Teensy 4.0. It utilizes an ARM Cortex M7 processor, a processor that is equal to or more powerful than the processor of the Discovery board, more than enough to handle audio processing at a higher bit rate. Additionally, the Teensy is fully compatible with the Arduino IDE and even contains audio libraries built for it. It even contains a set of analog pins (pins 14-23, A0-A9), which is needed for audio input (Fig 8).

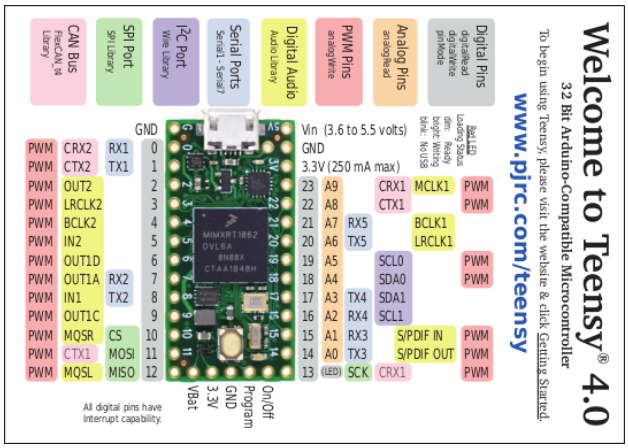


Fig. 8. Teensy 4.0 microcontroller pin sheet. A larger version can be found at the Teensy website.

After working with the Teensy for a while, it was discovered that it contains about 20 pins that can be used as a PWM. To reduce the footprint of components in the "Service area" of the project, the PWM driver board was set aside in favor of using the Teensy PWM pins. This way, everything comes directly from the Teensy, and there is no need to worry about communicating between multiple boards.

For ease of remembering, pins 0-11 are dedicated to the pumps, one for each pump. Pins 16 and 17 (A2 and A3) are used for the left and right audio input, respectively. Finally, pin 20 is dedicated to the LED strip. The reason for this can be found in Section II-I, which discusses the LEDs.

D. *Encapsulation*

The original idea of the encapsulation was more of a stand or a box in which all the circuitry is to go under. Figuring out how and where the speakers are placed will be after more of the project is completed. The parts used for the box were 3 pieces of clear, 8"x10" acrylic sheets. However, after hearing a colleague talk about 3D printing, that idea turned into having a completely custom-built stand, 3D printed, that could act as a stand for the clear container, hide the circuits inside, and mount the speakers.

The biggest hurdle is to make an encapsulation that fits the speakers and the clear container exactly or a little looser. There is also the constraint that the 3D printer in the Senior Hardware Lab can only print a max of 280mm (W) x 280mm (L) x 250mm (H), meaning the encapsulation needed to be printed in separate pieces before final assembly. Thankfully, the max height of the stand doesn't exceed the max height of the printer, so there is no need to print that part of the encapsulation separately.

In the end, there are 4 large pieces plus several smaller pieces that could be printed at the same time as one of the other 4 pieces. Table V in the Appendix shows the measurements of different pieces to be printed and measurements of the parts to contain.

Since the encapsulation needs to be printed in separate pieces, there needs to be a way to put them together. Using screws could potentially make the final assembly look chunky and the joins could be weak, depending on how the screws are placed. Instead, to make the final assembly look as streamlined as possible, dovetail joints are used on the inside of the box to tie all the pieces together.

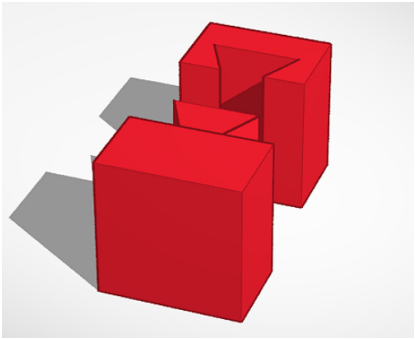


Fig. 9. A sample of a dovetail joint.

The process of creating the 3D print began in Blender and then moved to Tinkercad for ease of export. It also had premade shapes that could be combined to create another shape or to have one be a punch out for another. It does not have the same diversity as Blender to manipulate the shapes but works very well for this application.

Neither pictures nor words can fully illustrate how the following pieces look. An attempt will be made, but links will also be provided which each piece to be viewed in a 3D space. The encapsulation assembled in 3D can be found on Tinkercad.

1) ***Stand****:* The largest piece to print, estimated to take 2 days, 3 hours, and 20 minutes and use roughly 992g of material. It is made up of a cylinder about 4mm thick sitting atop a thicker one of about 25mm. The front half of the stand is completely solid while the back half has 3 openings cut out of the thicker cylinder for wire chases.

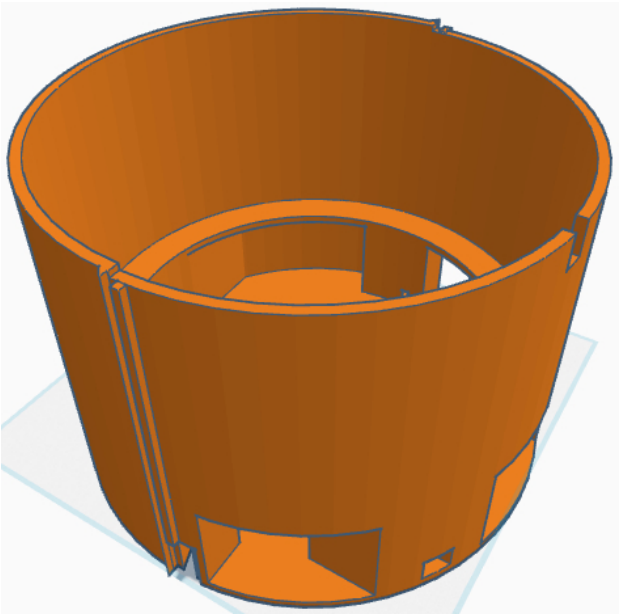


Fig. 10. The center stand piece in a top-down view to show as much as possible. The inset for the dovetail joint is not very well seen in the picture.

At the center, where the transition of the stand goes from pretty to technical, are 2 vertical tracks to hold the speaker box walls in place. The pretty side of the track is triangular-shaped and extends to the full height of the stand so that it almost appears as if the stand is blending into the speaker box. On the technical side, it is a square that runs from the top of the stand to the top of the joint from the speaker box.

Within the inset created for the above joint is a hole for a pin that prevents the speakers and service extension from sliding out from the stand when lifted. 2 square holes are cut into the pillars supporting the back side of the stand–these are to prevent the service area extension from moving up and down too much after attachment. Lastly, a hole is cut out from the top of the stand for the LED wires.

2) ***Speaker boxes****:* The left speaker (when facing the front of the project) has a large circle cut out to fit a speaker on the front face and 4 screw holes to mount said speaker. On the side of this face that attaches to the center stand, there is a full dovetail joint on the bottom corner, standing at about 1" tall and 0.5" wide. It contains a hole for the service area extension to slide into and another to hold the extension in place while preventing the speaker box from sliding out.

Half dovetail joints run along the floor and wall edges. The wall joint has a hole above the floor joint the size and shape of the cover that locks it in place. It is designed so the floors are locked first, then the walls.

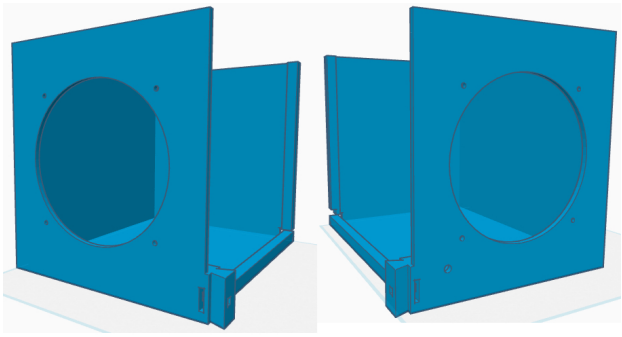


Fig. 11. The left and right speaker boxes, side by side to see the small differences.

The right speaker is an exact mirror of the left with the addition of a hole for the AUX cord to run out. It also has an extra hole above the cover one to allow power cords to run out from the back.

The left speaker was estimated to take 1 day and 45 minutes to print and use about 461g of material. The right speaker would use around the same amount of material and was estimated to take 4 minutes longer than the left.

3) ***Service Area Extension****:* The space that goes directly behind the stand and in between the two speakers. As opposed to the two speakers, it is only made up of a back wall and a floor.

The floor is cut out to fit the roundness of the center stand and given a 1" wall. This wall is added to keep water from leaking into the extension and touching electronics. It also has 2 square extrusions that match up with the cutouts made in the stand pillars.

The ends of this wall are designed to slip into the dovetail joints of the speaker boxes while the square block extensions are inserted into the cutouts in the center stand. At the same time, there is a hole made in the ends such that a pin can go through it from the speaker and into the stand.

The rest of the floor and back wall are designed to be the matching half for the speakers. The Cura slicer estimates the piece to take 16 hours and 51 minutes to print and use 306g of material.

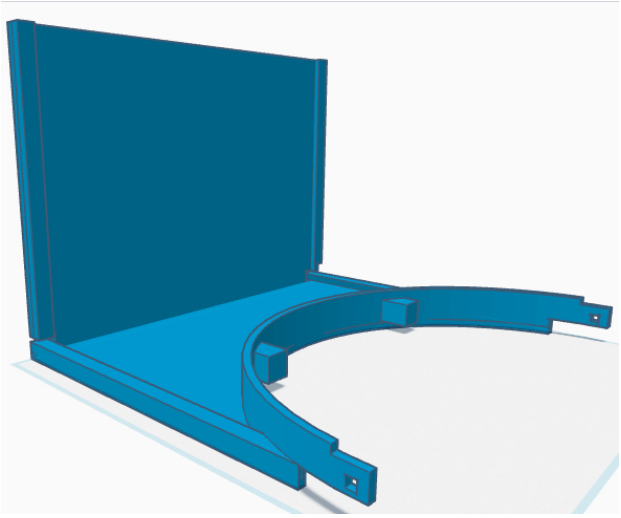


Fig. 12. The space that goes behind the center stand. Called the "extension" because it mostly directly attaches to the service area of the stand.

*4) **Dovetail Caps**:* The covers of the dovetails that slide over them and lock pieces into place. There are 2 types–caps for the wall joints and caps for the floor joints.

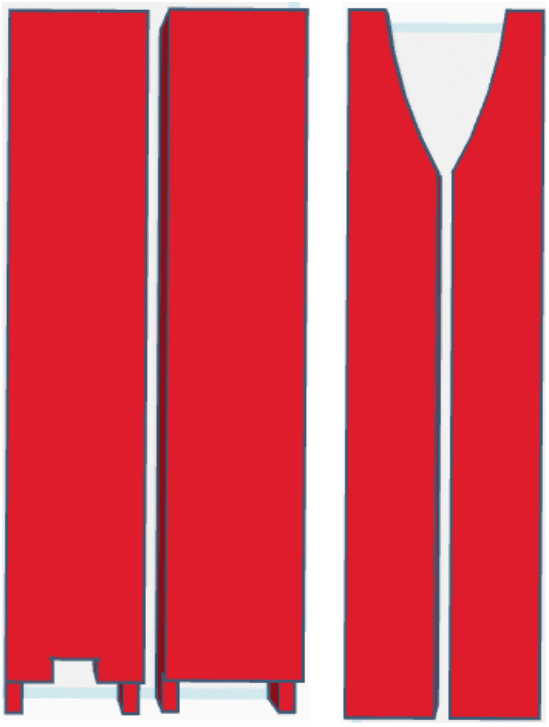


Fig. 13. All 4 caps. The left set is for the walls while the right set is for the floors.

The caps are merely long rectangles with the dovetail cut out. The floor caps have the additional cut of the curve of the service extension wall. The wall caps, on the other hand, have a space at the bottom cut out for the floor caps. This is to ensure the floor caps do not wiggle around too much if there is too much space inside the cover. One of the wall caps has an additional hole to accommodate the power cords.

The caps all together are estimated to take 8 hours and 18 minutes, using about 160g of material.

E. Pin

The most important part of the entire encapsulation. It is a small square rod that is inserted into the square hole made in the center stand, speaker boxes, and service extension wall. It holds all 4 pieces together at 2 points. Since it is so small, it was added to one of the other prints to be printed at the same time.

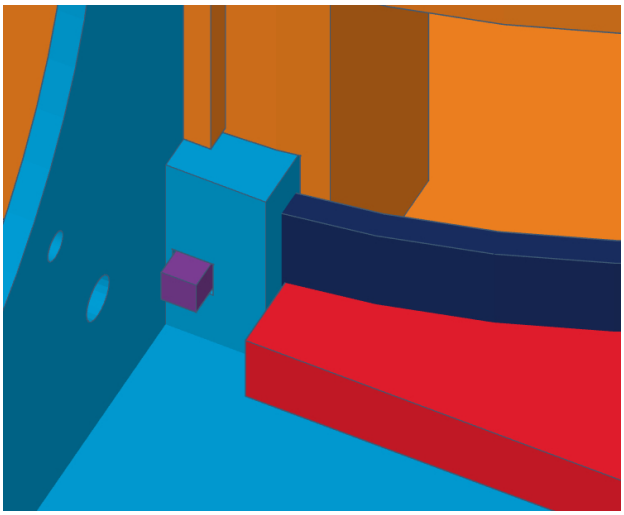


Fig. 14. One of the 2 points where all 4 pieces come together, held by the pin. The caps support the pin by preventing the separate pieces from moving independently.

F. Audio Circuit

The project must receive an audio signal of some sort, preferably music, to work at its full capacity. This audio can be taken from any source connected via the standard 3.5mm headphone jack (or AUX), USB-C cable, or lightning cable and sent through an STX-3120-3B stereo AUX port (Fig 15). To ensure that no one is left out when testing the final product, a 3-to-1 adapter cable is used in which the output has all 3 types of source cables and the input is a single AUX to be inserted into the port.

The Analog to Digital Converter (ADC) pins of the Teensy cannot read negative values while half of a typical audio signal is below 0. As such, the audio signal must first go through a level-shifting circuit before being sent to the microcontroller (Appendix E.1). From there, the signal goes through an FFT to get the numbers needed to set the height of the water leaving

Fig. 15. STX-3120-3B Stereo AUX Port

the pumps. A discussion of exactly what the FFT does can be found in Section II-H.

The audio port is a stereo audio port with outputs to a left and right speaker. Using them to create a stereo sound is not done, but each output is routed to a pin on the Teensy and a speaker.

G. ATX Power Source

The pumps in the project each use a max voltage of 5V and amperage of 0.18A. At their maximum flow rate, all pumps working together produce a current draw of 2.16A. No microcontroller can produce that kind of current, which means an external power source that not only provides 5V but a high enough (at least 3A) current draw is necessary. Additionally, a 12V power supply is needed to power the LEDs.

The LEDs' power supply is a simple 12V cord with no additional manipulation needed. As for the pumps, the first power supply unit was an AC-to-DC power supply adapter found on Amazon. It is supposed to take power from a generic, 120V power outlet and convert it to a 5V/5A power supply– more than enough for this application. It initially seemed to work when powering a single pump. After each consecutive pump is added to the draw, the voltage drops to the point where barely any water passes above the tube ends.

The second power supply unit was a supposedly more powerful version–a 5V/15A adapter. There was no difference in performance. Admittedly, it could also be because it is manufactured by the same manufacturer that made the first power supply.

In the end, an ATX power supply (Fig 16 from an old computer is used.

The ATX power supply is used to power multiple components in a computer with voltages varying from 3.3V to 12V and currents of 5A to 25A. However, the ATX power supply only works when there is a motherboard is hooked up to it. To simulate having such a load required a bridge between two pins at the center of the 24-pin output. Newer ATX power

Fig. 16. ATX Power Supply

supplies will need a power resistor in the bridge, but this older ATX (from the early 2000s) only needed a wire (Fig 17).

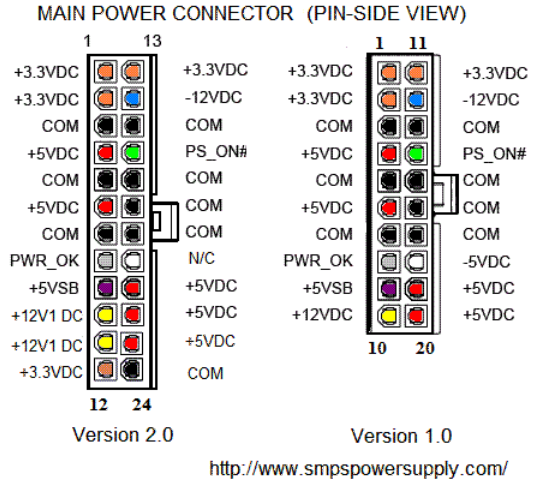


Fig. 17. Reference used to map out which pins to use and which pins to bridge on the ATX.

Since the ATX can output 12V, the 12V power cord used for the LED is replaced with a line to a 12V pin. Additionally, the 5V pin works perfectly for powering all the pumps to maximum flow rate. Another 5V pin is used for the logic shifter. A 3.3V pin is used for both the logic shifter and the audio circuit. Another 12V pin is used for the two speakers. Finally, one ground pin is used to connect all circuits.

H. Fast Fourier Transform Analysis

Fast Fourier Transform (hereafter referred to as FFT) is an algorithm for analyzing signals to convert them into sections that describe the frequencies present. The frequency resolution of each spectral line is equal to the Sampling Rate divided by the number of points in the FFT. Therefore, increasing the

number of points to sample [1] can achieve a higher spectral resolution. The downside to this, however, is that the more points given to the FFT, the more time it will take to calculate.

FFT works by taking a signal–in this case, the music piece– and sampling it over a period of 11.6 ms (resulting in new data consistently at a rate of 86 Hz). That sample is then divided into its frequency components. Each of these components is a discrete, sinusoidal oscillation with a unique frequency, amplitude, and phase [2]. Fig. 18 represents the frequency graph of a sample signal in which 3 distinct frequencies were found.

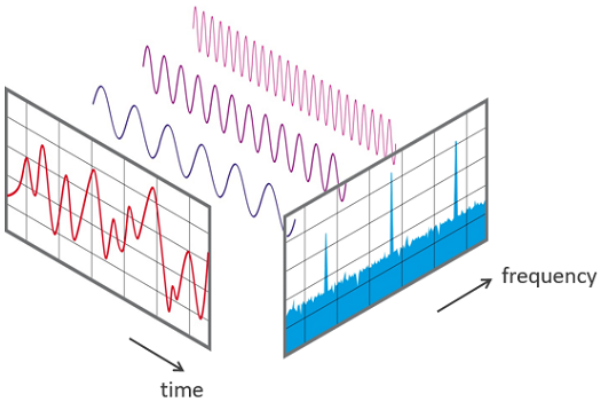


Fig. 18. A signal broken up into the 3 distinct frequencies that it is made of [2].

Whenever there is a spike in the frequency of the original signal, that spike is reflected as the amplitude of the individual frequencies. That amplitude can then be converted into a usable, numerical value to represent the height of each bar of the visualizer; in this case, the height of the water column.

There are multiple versions of the FFT algorithm. In this case, the FFT used is the one built-in for the Teensy 4.0. It creates a 256-point FFT and 127 bins of readable data. Each bin covers a range of 172Hz, which allows sampling of audio frequency data between 0Hz and 20KHz. However, this project is mainly used for the representation of music, so not all bins are needed.

Most frequencies in music fall within the range of 0Hz to 4KHz, which are the first 24 bins. Of these 24 bins, the first 11 are assigned to the first 11 pumps. The remaining 13 bins contain all of the high notes, which are normally not heard, and therefore, averaged and assigned to the last pump.

I. NeoPixel LED Strip

There are many options available to use as highlights for the water streams. One that was closest to the desired output of the project, a ring light with an opening for the stream to go through, was too expensive (roughly $15 per ring, a total of about $180 sans tax and shipping). Instead, the waterproof NeoPixel LED strip by Adafruit is used.

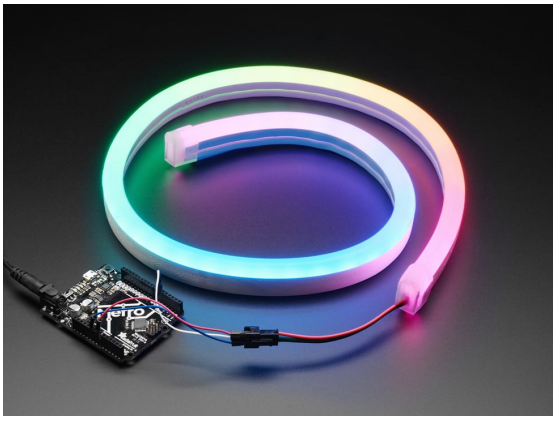


Fig. 19. NeoPixel LED Strip with waterproof Silicone Tube

The strip utilizes a WS2811 driver chip that takes in 12V of power and 5V logic. The strip is 1m long, far too long for the container it is going into. A section of 6 LEDS (out of a total of 20) is cut off. Electrical grade silicone is added to the cut-end going into the project to protect the circuitry inside from water. Adhering to the LED strip is left towards the very end and done with the E6000 adhesive. To make it fit as tightly as possible to the sides of the tray, the strip is wound opposite of the way it came in.

For the NeoPixel to work with the Teensy 4.0, it needs to utilize the Parallel Output of its native FastLED library (documentation, first paragraph only). In short, the Teensy has 3 sets of pins to which the strip connects. All LED strips within a set of pins will run parallel, although unnecessary for this project. Even so, the strip must be connected to one of the pins stated; otherwise, it will not work.

As mentioned previously in section II-C about the micro-controller, the LED strip is connected to pin 20. To ensure maximum life of the LED, Adafruit recommends adding a 1000μF across ground and power and a 330Ω-Ω resistor. The capacitor prevents any damage to the LEDs caused by sudden power spikes. The resistor stabilizes the signal coming to the LEDs and prevents flickering (Fig E.5).

J. Code

All programming is done in the Arduino IDE in C++. The process, partially illustrated in Fig 20, begins by using the ADC to read analog pins A2 and A3 (labeled as pins 16 and 17 on the Teensy). The analog pins are connected to the left and right channels of the audio port where the input is retrieved.

Once the analog data is transformed into digital data, the signal is fed through a mixer object and then an amp to

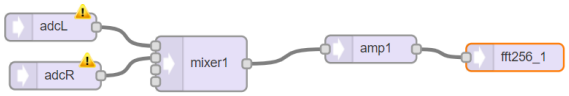


Fig. 20. Code path for audio. The warnings on the inputs are the designer merely stating multiple inputs are going into the mixer.

combine the stereo signal into a mono signal. The amp also boosts (amplifies, as it should) the signal before giving it to the FFT.

Here, the FFT sets a flag to true when there is data ready to read, false otherwise. Only when the flag is true are the bins containing the FFT data read. That data is given to an averaging function that allocates the 24 bins down to an array of 12 bins with each bin representing a pump. As mentioned in section II-H about the FFT, only the first 24 bins are needed as frequencies of music only fall within the range those bins represent.

Before the data in the final 12 bins are sent to the pumps, they first go through a check to make sure the value does not fall below a certain threshold to filter out noise. There is no set threshold to use for all pumps–each pump has a threshold value as the voltage they use to operate varies from one another. Because each pump operates slightly differently, the value that is given to the pumps as voltage is also set a bit differently. For instance, one pump requires a higher value to meet the others at max height while another uses a lower value. See Appendix D1, setPumps() function for all the calculations.

At the same time the FFT/pump is running, the LED strip is cycling through a set of patterns around the water streams. To prevent pauses in the program with delays, counters are utilized to create effects that are based on how many iterations the main loop() function has gone through.

Here is a list of patterns created:

1) **Fill solid** – The simplest pattern. Fills the entire strip with a single, solid color with no effects or timing done.
2) **Fade in/out** – The entire strip is filled with a single color (selected or random) and faded into full brightness then back to black. The color can be set to a single one or randomly selected from an array of pre-selected colors. The changing of colors happens on every instance of the pattern fading in.
3) **Chase** – Fills every other LED pixel with a color, waits (value provided as an argument), then switches to fill the other LED's with the same color. The color can be set to a single one or randomly selected from an array of pre-selected colors. The changing of colors is determined by a value passed as an argument. This pattern creates the illusion of spinning.
4) **Rainbow cycle** – Fills the strip with the rainbow colors softly blending. At the same time, the colors move along the strip, creating a moving rainbow.
5) **Color blend cycle** – Fills the entire strip with a single, solid color, then slowly blends it with the next color. The main colors are the colors of the rainbow but cycle through the entire color palette.
6) **Increment fill to black** – Set the strip to black, then turn on one LED at a time, consecutively, until the strip is filled. Once filled, the LEDs are turned off one at a time in the same manner as they were turned on. All LEDs are the same color which is either a single set color or a randomly selected one from an array of pre-selected colors. The changing of colors happens each time the LEDs turn back on. This pattern resembles a loading bar.
7) **Increment fill to new color** – At the very start of the pattern, it begins like the previous pattern where it incrementally turns on from black. However, after the color is filled, the strip is not turned off. Instead, a new color is filled in.
8) **Single color wipe** – Only a single LED is turned on at any given time. It begins at the start of the strip and travels along until it hits the end. When the next LED is turned on, the previous one is turned off. The color is the same across the strip and is either set to a single set color or randomly chosen from an array of pre-selected colors. The color change happens when the lit LED returns to the beginning of the strip.

For details of how these patterns are implemented, see Appendix section D2. Note that there are 2 additional patterns at the very end of the code that are not implemented.

K. Final Assembly

Once all the previous individual parts are finished, start by mounting the speakers to each of the speaker boxes. The speakers use screws that are the same size as the ones used for VESA monitor mounts. The holes are slightly smaller than the screws–use a screwdriver to "turn" the screws into place. Secure them with washers (for additional strength) and nuts. The speakers need to be as flush as possible to the face of the boxes or else it could cause a bit of warping.

Slide each speaker box into position in the stand, one at a time. It could be done at the same time, but positioning the boxes just right is difficult. Add the service area extension–it is a very tight, friction fit, so a little more force is required to get it into place. Once it is in place, insert the locking pin, then the two floor caps, and finally, the wall caps.

Put the clear container into the center stand and pull the pump wires through one of the openings in the service area, preferably to the right side (facing the back of the stand). This allows more space for the other circuits to sit in the extension

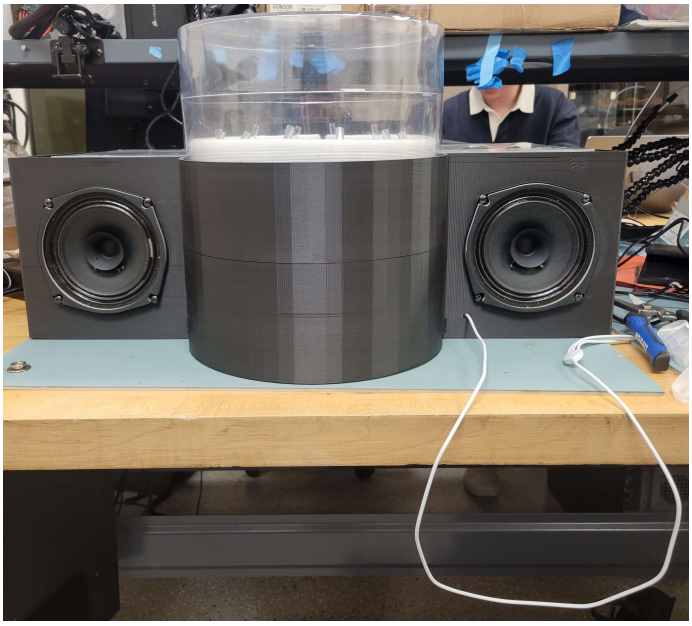


Fig. 21. Front view of the final product. Missing from the picture is the ATX power source.

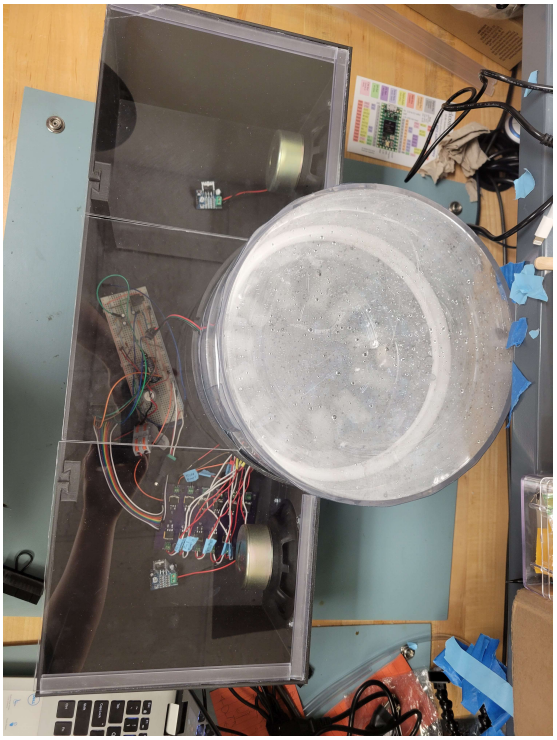


Fig. 22. Top down view of service area. The size of the Teensy is so small, it is not easily spotted in the picture, but it is there, in between the breadboard and the pump PCB.

area. There is not enough room for the pump PCB to feed through the service area, so that needs to be unhooked prior to the insertion of the container. It is re-wired once the clear container is in place and wires pulled out.

The rest of the circuit, mostly contained on a single breadboard, and the Teensy is added in afterwards. If the pump PCB was previously wired to the Teensy, then likely the entire circuit is put in at the same time the PCB is. The wires providing power from the ATX is fed through the premade hole in the encapsulation and plugged into their respective spots on the breadboard. The ATX will sit behind the encapsulation during operation, but during transportation, it will be placed inside the encapsulation such connections are lost or misplaced upon setup.

The acrylic pieces originally intended to be used as the service area and stand of the clear container are repurposed to be the lid of the encapsulation. This way, anyone can look into the guts of the project without having to repeatedly take off an opaque lid. 2 of the pieces are cut down to the length of the speaker boxes then have a small corner carved out into the curve of the clear container. The final 3rd piece merely has a curve cut out for the stand and serves as the middle piece of the lid (shown in Fig 22).

A set of plastic, flex hinges were obtained to, originally, serve as hinges for the lid and a means to keep it into place. But the hinges could not conform to the shape of the clear container. They also created about a 1/4" gap between the Stage Curtain and the Lid, which is a prime spot for water to come gushing out. Instead, they are reused as the holder for the lid. Since there are two sides to the hinge, the flexible part of the hinge is cut out to give two tracks. These tracks are cut to size and glued to the very top of the front and side walls of the speaker box. The back is the access to get the lid off, which means the middle lid is free floating.

The last thing is to thread the AUX adapter cord through the hole in the speaker and plug it into the audio jack.

III. Evaluation

Thankfully, many of the parts of the project operated as expected, with only some minor setbacks. The following sections discuss the setbacks faced by each part.

A. *Pumps heights and Pump PCB*

When testing the pumps at the same time with a single 5V, 5A power source, and a maxed PWM signal, the max height of the water streams ended up being much lower than expected. Additionally, the more pumps added to the test, the lower the collective height becomes.

The original motor driver circuit included a 1kΩ resistor between the ground and the NMOSFET and was also included in manufacturing the PCB. During troubleshooting the initial

low power output, the 1k resistors were removed to check if it was a point of bottleneck. Removing the resistors made no difference in performance, even after fixing the power issue. As such, the resistors were not added back in.

The solution to this problem was to switch the power source used. Once it was swapped for the ATX power supply, the pump heights increased multiple times over. Unfortunately, however, after the swap was made, one of the 12 pumps burnt out. There was no easy way of replacing it–it meant cutting out the silicone seal made at the bottom of the tank, removing the damaged pump without damaging the tank, adding a new pump, testing it, resealing the hole, and making sure the seal is just as good as it was before.

In the interest of time, replacing the pump was an afterthought. Getting all other pumps working as intended came first and thankfully, not many noticed the failed pump.

B. Acrylic Casing

The biggest thing the acrylic trays needed to do is to make sure every seal made is waterproof and keep the water away from the electronics. All seals held extremely well except for the very last one added, which is the hole the LED strip wires come out of. During testing, no water leaked. During demo, towards the end of the session, there was a small leak. The biggest difference between testing and demo is the duration the fountain is run each time. The fountain is turned on for only short periods of time while during demo, the fountain was running the entire time.

Thankfully, no water made it into the service area extension where the electrical components were. The water did leak from the crack between the stand and the extension, however...

C. Encapsulation

There was the fear that it would be too flimsy since the maximum thickness is about 3.5mm, but it turned out to be incredibly sturdy, especially with all the pins and dovetail joints secured.

To make sure the encapsulation was printed as perfectly as possible, several test runs were done, especially the joints. Through these tests, it was found that the main joints where everything came together was slightly off. The inset and dovetail joints came out to be the same size. What's more, the barrier wall of the service extension went out further than the hole in the dovetail joint allowed. After some small tweaking of the sizes, all prints came out as friction fits.

Out of all the pieces, the center stand was the one with the most issues. Issues with the other pieces mainly consisted of making sure that they fit against each other close to perfectly and that the screw holes for the speakers were in the right spot. But for the center stand, initial sketches of it would have made for really long print times–the first one was roughly 3 days! To reduce the time and material used, the thicker cylinder part of the stand was hollowed out so only a 3.5mm wall remained. A single, 1" support was added to the front and the two sides for the dovetail insets were left in.

The second issue came about when printing the center stand. There was not enough filament to print the entire thing, and the LulzBot Taz 6 did not have the feature where it stops printing when there is no more filament. As a result, the print "finished" with only about 108mm actually printed. As to not waste money, as that was a lot of material used, it was decided to print the remaining 83mm in a separate print and glue the pieces together. Thankfully, the seam of the two parts was not visible unless pointed out.

D. FFT

One main issue we ran into during the implementation of the FFT algorithm was the filtering of noise on the analog values. During our initial testing of the code, we had no music playing into the circuit and had the values of the FFT bins printed to the serial monitor of the IDE every time it updated. We noticed every few seconds that the values in many of the lower frequency bins would jump from their natural low values to a fairly large value for a split second before returning to normal. When tested with the pumps connected, those spikes would cause their respective pumps to shoot a single brief stream of water. We found that the only solution was to decrease the amplifier value in the code and increase the threshold value that prevents the pumps from running if the value is below it.

E. LED strip

The only setback we had with the LED strip was getting it to work with the Teensy. While the initial code we had worked on the Arduino UNO, it unfortunately failed to run on the Teensy. We discovered that the LED strip requires 5v logic on the data pin, and because the Teensy 4.0 logic level maxed at 3.3v, the LED strip wasn't able to read the data. The solution was to use a logic shifter to boost the level to 5v. Luckily we found one in the lab that allowed us to feed the initial signal in as well as a 5v source to adjust the value correctly. Only then would the data line be read by the LED strip.

F. Speakers

While the amplifier board for one of the speakers worked, the other one of the same type didn't. We found during our final testing phase of the project, one speaker sounded more static-like than the other speaker, we tried swapping out the speaker for another of the same type, but with no difference.

We also tried changing the wire that fed the analog signal to the amp to a thicker gauge to reduce noise, also with no difference. We determined that the amp was the faulty part in the line, and we were able to request a different one. The new amplifier worked with only 5v instead of 12v, so it had to be turned up slightly to match the volume of the other speaker.

IV. Conclusion

All-in-all, this capstone project is like a combination of the Bellagio Hotel water show and commercially sold speakers with the four water pumps. The main difference is it shrinks the Bellagio's fountain to a more personable size and can take any music to create a unique visual response calculated in real-time.

Future versions of this project could be done to improve the quality of the show and fix some shortcomings we didn't have time to include. Implementing more, better quality pumps to increase the resolution of the spectrum analysis. A new singular PCB could be designed to utilize all the circuitry needed for the pumps, as well as the circuits we made using a breadboard, the PCB could also be designed to be mounted upright to greatly reduce the size of the service area needed to hold the circuitry and make the product even more portable. The acrylic container for the water would be made using a solid cylindrical container instead of multiple trays put together to reduce the risk of leakage. The 3D print could be redesigned to be done in one solid piece instead of multiple parts joined together. The tubing for the pumps could be redone to include nozzles that better control the direction of the water flow so that each pump is equal.

In terms of code, a future version can implement our stretch goals (see Appendix Section A) including a beat detection algorithm to make the LED strip react to the music or a more in-depth analysis of the audio spectrum to create pump patterns similar to those at the Bellagio fountain instead of just a display of the spectrum.

V. Links

Website: https://my.eng.utah.edu/~byee/index.html

Repository: https://github.com/BYee7127/public_html

References

[1] Anon. *FFT Size*. [Online]. URL: https://www.spectraplus.com/DT_help/fft_size.htm. [Accessed: 2023-Mar-22].

[2] NTi Audio. *Fast Fourier Transformation FFT - Basics*. [Online]. 2019. URL: https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft. [Accessed: 2023-Mar-22].

[3] David. Forney and Deborah I. Fels. "Creating Access to Music Through Visualization". In: *2009 IEE Toronto International Conference Science and Technology for Humanity (TIC-STH)*. Toronto, ON, Canada, Sept. 27, 2009, pp. 939–944. URL: https://ieeexplore.ieee.org/abstract/document/5444364.

[4] Eric J Isaacson. "What You See Is What You Get: on Visualizing Music." In: *ISMIR*. 2005, pp. 389–395. URL: https://ismir2005.ismir.net/proceedings/1129.pdf.

[5] Microsoft. *Microsoft Windows Media Player 7 Brings Click and Play Digital Media To Millions Around the Globe*. [Online]. July 17, 2000. URL: https://news.microsoft.com/2000/07/17/microsoft-windows-media-player-7-brings-click-and-play-digital-media-to-millions-around-the-globe/. [Accessed: 2023-Mar-16].

[6] A. Rothstein. *The Basics of Music Visualizers*. [Online]. Dec. 6, 2019. URL: https://www.ipr.edu/blogs/audio-production/the-basics-of-music-visualizers/. [Accessed: 2023-Mar-16].

[7] Wanda Marie Thibodeaux. *What is Music Visulaization?* [Online]. Feb. 7, 2023. URL: https://www.musicalexpert.org/what-is-music-visualization.htm. [Accessed: 2023-Mar-16].

VI. Biographies

Alex Gray is a student at the University of Utah pursuing a Bachelor of Science degree with a major in Computer Engineering and a minor in Arts Technology. He has done work with embedded system design and software development with a focus on the digital media and arts industry, including image and audio processing as well as interactive computer graphics. He is passionate about utilizing computers in the media and arts industry thanks to his experience with electronic music production, digital photography, and digital graphic design.

A 2nd generation Chinese-American, Beverly Yee is a senior at the University of Utah pursuing a Bachelor of Science degree in Computer Engineering. Her specialty lies in web programming and digital media, but has worked with computer hardware, digital system design, and software development. Beverly has experience with graphic art design and does sketch/digital art as a hobby.

APPENDIX

A. *Stretch Goals*

The following ideas are implementations we were unable to add to our project but could be added in the future, listed from highest to lowest priority/desire.

1) ***Beat Sync****:* This particular goal was the most highly anticipated one as the desired output would have stepped up the visualization of the input music. The idea was that each time there was a kick in the beat, the LED strip would respond with a brightness pulse synced to the beat. The method of recognizing the beat for this goal is a little more complicated. In addition to computing the FFT of the signal, an analysis of the FFT bin data is needed to detect a particular pattern that depicts the beat. The potential drawback lies in the accuracy of the detection at the cost of the speed of the calculation. As such, this goal remained a stretch goal such that the processing speed of the signal remains in real-time.

2) ***User Interface****:* For this goal, the idea is to set up an LCD screen with buttons so a user can adjust the visualizer to their preferences, such as changing the song or switching up the settings of the LEDs to different options (eg. rainbow cycle, solid color, beat sync, etc). Having a user interface is not crucially necessary for the project to work, hence why it remained a stretch goal.

3) ***Bluetooth****:* As the title suggests, instead of using an audio jack to receive audio, attach a module to the microcontroller so the visualizer can connect to Bluetooth-enabled devices, such as a smartphone.

B. *Bill of Materials*

TABLE I: Hardware

#	Item	Qty	Vendor	Item #	Additional Notes
1	Pumps	16	Amazon	B097F4576N	4 extra. Comes in a pack of 4 and tubing.
2	Acrylic trays	4	LO Florist Supplies	ACYL102	Container for water show.
3	8” x 10” acrylic sheets	3	Lowe’s	55844	Lid for the enclosure. Enables viewers to look inside without having to take things apart.
4	NeoPixel LED strip	1	Adafruit	3869	1 meter strip with 20 LEDs; only used 14 (cut off 6).
5	Contour tool	1	Amazon	B0BWN5NG15	Made by MECHEER. Used to determine the contour of the bottom of the acrylic trays.
6	Audio adapter	1	Amazon	B0994FL13N	Splitter for 3.5mm audio port, USB-C, or lightning cable to a single 3.5mm audio jack.
7	Speakers	2	–	–	Output for the project. It is wired separately from everything else and retrieved directly from the AUX port. It does not go through the level shifter the AUX port uses.
8	Screws	8	–	–	Used to mount the speakers to the 3D enclosures. They are the size of the screws found on VESA monitor mounts. No.8 screw size (?).
9	Washers	8	–	–	No.8 screw size, ID = 3/16”, OD = 7/16”.
10	Nuts	8	–	–	No.8 screw size (?). The same size was used for screws found on VESA monitor mounts.

11	Vinyl sheet	1	Amazon	B0B7G2SQVH	Used to encase the clear container and create a spot where the lid can slip into and stay in place.

TABLE II: Circuit Components

#	Item	Qty	Vendor	Item #	Additional Notes
1	Pump PCB	1	OshPark	–	2 extra. Manufactured in packs of 3. See E.4.
2	Audio port	3	Digi-Key	STX-3120-3B	2 extra. Point to receive input.
3	Surface-mount transistors	16	Digi-Key	IRLZ34NSTRLPBF	4 extra. NMOSFET to monitor the signal being sent to the pumps. Helps with operating the PWM.
4	Surface-mount diodes	16	Digi-Key	B140-13-F	4 extra. Used to protect the pumps and NMOS from a sudden drop or spike in voltage.
5	Terminal blocks	16	Digi-Key	282834-2	4 extra. Connection points for the pumps to the pump PCB.
6	24-Pin ATX jumper bridge	1	Amazon	B01N8Q0TOE	Made by CRJ Electronics. Used to trick the ATX power supply into thinking there is a motherboard attached to it. Otherwise, it will not power on.
7	Teensy 4.0	1	Amazon		Brains of the entire project.
8	Logic shifter	1	–	TXB0108	Provided in senior hardware lab. Needed to shift the output signal from the Teensy (3.3v) to 5v the NeoPpixel needs.
9	Speaker amplifiers	2	–	TDA2030a	Provided in senior hardware lab. Used to amplify the signal going out to the speakers.
10	MISC resistors / capacitors / wires	–	–	–	Provided in senior hardware lab. To assemble and connect the circuits, refer to schematics

TABLE III: Equipment

#	Item	Vendor	Additional Notes
1	Dremel	Dremel	Used for cutting pieces off acrylic sheets, drilling holes into trays, and refining the edges of the laser-cut holes.
2	Soldering iron	Plusivo	–
3	Crimping tool kit	KAIWEETS	Used to add ferrules to the end of wires. Helps prevent strands from being misplaced. Bought from Amazon (item #: B0BQ6LQ12G).
4	Shop shears	Kobalt	For heavy duty cutting. Bought from Lowe's (item #: 961371).
5	Multimeter	Plusivo	–
6	TAZ 6 3D Printer	Lulzbot	in senior hardware lab
7	Laser Cutter	Full Spectrum	120w CO2 laser cutter with a 24x36" bed. Used for cutting out large sections of the acrylic trays.
8	Helping hands	NEWACALOX	Adjustable stand with multiple arms and clips. Helps to hold components in place while soldering.

9	Oscilloscope	Tektronix	for testing audio signals
10	Wire striping tool	–	provided by lab
11	1 gal water	–	To fill the containers with. Also, pumps cannot run without water for extended periods, or else possible damage may occur.
12	ATX Power supply	Dell Inc.	305W power supply. Contains pins for 12v, 5v, and 3.3v with up to 25A current.
13	Heat gun	Dewalt	Heat soldering paste and heat-shrink tubes.
14	Microscope	–	Used to see the extremely small surface mount resistors.
15	TinkerCAD	–	Simple, online 3D builder. Used to make the encapsulation.
16	ArduinoIDE	–	Software dedicated to uploading code to compatible microcontrollers.
17	Cura slicer	Lulzbot	Slicer to prepare the GCode needed to print 3D materials.

TABLE IV: Adhesives/Other items

#	Item	Qty	Vendor	Item #	Additional Notes
1	Clear caulking	1	Lowe's	727380	Silicone adhesive and sealant. Used for sealing the holes made for wires.
2	Double-sided tape	1	Lowe's	394705	Made by Scotch. 1" x 5'. Waterproof. Used to adhere the pumps to the bottom of the acrylic trays.
3	Clear duct tape	1	Lowe's	488028	Made by Gorilla. 1.88" by 27'. Waterproof. Used to tape the trays together.
4	Solder paste	1	Amazon	B075ZR52JM	Made by MG Chemicals. Used to solder the surface mount components onto the pump PCB.
5	Heat shrink tubing	1	Amazon	B0919GN3SX	Made by XHF. Adhesive and waterproof. Used for tying wires together without having to solder.
6	Electronic-grade silicone	1	Amazon	B0063U2RT8	Made by Gordon Glass Co. Used to cover the cut end portion of the LED strip such that water or dust doesn't get inside. Especially water.
7	3D-print filament	3	Amazon	B08QMM8T2S	Made by PolyMaker. PLA-TerraMaker (?), 2.8mm, black.

*C. **Tables***

TABLE V: Measurements to Make Encapsulation

#	Part	Size	Notes
1	Wall thickness	3.5mm	Thickness of the sides of the clear acrylic container. It seemed thick enough, so the thickness of the encapsulation mirrors it.
2	Stand diameter (inner)	256mm-258mm	The inside of the stand for the clear container. The diameter of the acrylic tray is roughly 254mm. Added 2-4mm for wiggle room.
3	Stand diameter (outer)	263mm-265mm	Outside of the stand for the clear container. Includes the 3.5mm thickness on both sides of the circle, 7mm.

4	Pool bottom to stage bottom	127mm	The height of the tank container the pumps. Also the height of 2 trays, which is 2.5” (63.5mm) each.
5	Service area (pool) height	64mm	The area underneath the clear container for the pump wires to feed through. For an easy number, the tray height rounded up is chosen. It also accounts for the thickness of the floor and the thickness of the ledge holding the clear container.
6	Total Stand height	191mm	#4 + #5 = 127mm + 64mm = 191mm.
7	Speaker back lip	124mm	Measured the circumference of the back lip and calculated the diameter from it. Circumference = 38.6cm; Diameter = 122.87. Rounded it up and added a millimeter for wiggle room.
8	Speaker hole edge to box edge	25mm	Space for the speaker from the edge of its box. Roughly 1”. Measurement is only for the padding to the left and right of the speaker.
9	Speaker box (height)	191mm	For a streamlined look, the height of the speaker box matches the height of the stand.
10	Speaker box (width)	177.5mm	#7 + #8 * 2 (to account for padding on both sides of the speaker) + wall thickness.
11	Speaker box (depth)	221.5mm	The speakers are set halfway back from the front of the clear container. The box covers the remaining distance (62mm) + roughly 3” (76.2mm) for circuit items.
12	Pin	–	4.5mm (W) x 4.5mm (H) x 40mm (L) The key that keeps everything together.

D. ***Code***

1) ***WaterFountainMain.ino****:*

```
/*****************************************************************************
 *
 * File:       WaterFountainMain.ino
 * Author:     Alex Gray & Beverly Yee
 * Date:       November 08, 2023
 *
 * Calls to the different parts of the Water Fountain are made here.
 * To add a sketch to be a part of the project, make sure the sketch is saved in the
 * Sketches folder of WaterFountainMain. Go to Sketch -> Add File..., and navigate to
 * the sketch to add.
 *
 * DO NOT call setup() or loop() in any of the other sketches. If it is necessary to
 * do some setup, call it setup[object]() where [object] is the thing to setup.
 *         i.e: setupLEDStrip() or setupFFT()
 *
 *****************************************************************************/
#include <Audio.h>
#include <Wire.h>
#include <SPI.h>
#include <SD.h>
#include <SerialFlash.h>

AudioInputAnalog adcL(A2);
AudioInputAnalog adcR(A3);
```

```
AudioMixer4 mixer1;
AudioAmplifier amp1;
AudioRecordQueue queue1;
AudioAnalyzeFFT256 fft256;
AudioConnection patchCord1(adcL, 0, mixer1, 0);
AudioConnection patchCord2(adcR, 0, mixer1, 1);
AudioConnection patchCord3(mixer1, amp1);
AudioConnection patchCord4(amp1, queue1);
AudioConnection patchCord5(amp1, fft256);

static float FFTarray[12];

#if defined(__IMXRT1062__)
extern "C" uint32_t set_arm_clock(uint32_t frequency);
#endif

// ================ Pump macros ================
#define PUMP_1 11
#define PUMP_2 10
#define PUMP_3 9
#define PUMP_4 8
#define PUMP_5 7
#define PUMP_6 6
#define PUMP_7 5
#define PUMP_8 4
#define PUMP_9 3
#define PUMP_10 2
#define PUMP_11 1
#define PUMP_12 0

// ================ Main code ================
/*
 * Setup everything at startup.
 * Runs only once.
 */
void setup() {
  Serial.begin(9600);
  AudioMemory(13);
#if defined(__IMXRT1062__)
  set_arm_clock(600000000);
#endif
  setupLED();
  amp1.gain(35.0);
}

/*
 * Endless loop.
 */
void loop() {
  if (fft256.available())  //verify fft has output data
  {
    averageFFT();
    for (int i = 0; i < 12; i++)  //for every fft, bin set the pump
    {
      // Serial.print((double)FFTarray[i], 3);  //print fft bin for testing
      // Serial.print(" ");
```

```
      setPump(i, FFTarray[i]);
    }
    // Serial.println();  //new line
  }

  // individually test all patterns before combining them in a cycle
  // lineUpTest();
  // cycleAllColors(800);
  // fadeInOut(8);
  // fadeInOut(8, 5);
  // showSingleFill(1000);
  // showSolid(7);
  // incrementFillToBlack(60, 2);
  // incrementFillToBlack(60);
  // incrementColorFill(60);
  // singleColorWipe(1, 30);
  // singleColorWipe(30);
  // chaseRandom(80, 10);
  // chaseIndex(80, 8);
  // fillRainbowCycle();
  // fillCycle();
  cycleAllPatterns();

  // pumpsOff();
  // pumpsHigh();
}

// ================ Helper functions ================
/*
 * Sets given pump based on value from FFT.
 *
 * Calculates the bottom threshold of the bin attached to the pump.
 * Do not turn on the pump. Otherwise, calculate the max height of the pump
 * in respect to the other pumps and turn it on.
 */
void setPump(int pump, float val) {
  float newVal;  // value to pass to the pump
  if (pump == PUMP_12) {
    if (FFTarray[pump] < 0.10)  //lower threshold
    {
      analogWrite(pump, 0);  // don't turn on
    } else {
      newVal = (val)*70 + 125;
      analogWrite(pump, newVal);
    }
  }

  if (pump == PUMP_11) {
    if (FFTarray[pump] < 0.10) {
      analogWrite(pump, 0);
    } else {
      newVal = (val * 2) * 55 + 175;
      analogWrite(pump, newVal);
    }
  }
```

```
if (pump == PUMP_10) {
  if (FFTarray[pump] < 0.10) {
    analogWrite(pump, 0);
  } else {
    newVal = (val * 2) * 55 + 190;
    analogWrite(pump, newVal);
  }
}

if (pump == PUMP_9) {
  if (FFTarray[pump] < 0.10) {
    analogWrite(pump, 0);
  } else {
    newVal = (val * 3) * 70 + 175;
    analogWrite(pump, newVal);
  }
}

if (pump == PUMP_8) {
  if (FFTarray[pump] < 0.10) {
    analogWrite(pump, 0);
  } else {
    newVal = (val * 2) * 65 + 160;
    analogWrite(pump, newVal);
  }
}

if (pump == PUMP_7) {
  if (FFTarray[pump] < 0.10) {
    analogWrite(pump, 0);
  } else {
    newVal = (val * 4) * 40 + 205;
    analogWrite(pump, newVal);
  }
}

if (pump == PUMP_6) {
  if (FFTarray[pump] < 0.08) {
    analogWrite(pump, 0);
  } else {
    newVal = (val * 6) * 60 + 180;
    analogWrite(pump, newVal);
  }
}

if (pump == PUMP_5) {
  if (FFTarray[pump] < 0.08) {
    analogWrite(pump, 0);
  } else {
    newVal = (val * 8) * 65 + 180;
    analogWrite(pump, newVal);
  }
}

if (pump == PUMP_4) {
  if (FFTarray[pump] < 0.07) {
```

```
      analogWrite(pump, 0);
    } else {
      newVal = (val * 12) * 50 + 185;
      analogWrite(pump, newVal);
    }
  }

  if (pump == PUMP_3) {
    if (FFTarray[pump] < 0.10) {
      analogWrite(pump, 0);
    } else {
      newVal = (val * 2) * 70 + 170;
      analogWrite(pump, newVal);
    }
  }

  if (pump == PUMP_2) {
    if (FFTarray[pump] < 0.10) {
      analogWrite(pump, 0);
    } else {
      newVal = (val * 8) * 55 + 200;
      analogWrite(pump, newVal);
    }
  }

  if (pump == PUMP_1) {
    if (FFTarray[pump] < 0.06) {
      analogWrite(pump, 0);
    } else {
      newVal = (val * 20) * 90 + 130;
      analogWrite(pump, newVal);
    }
  }
}

/*
 * Turns off all the pumps. Essential if we want to test things without the pumps running.
 */
void pumpsOff() {
  analogWrite(0, 0);   // pump 12
  analogWrite(1, 0);   // pump 11
  analogWrite(2, 0);   // pump 10
  analogWrite(3, 0);   // pump 09
  analogWrite(4, 0);   // pump 08
  analogWrite(5, 0);   // pump 07
  analogWrite(6, 0);   // pump 06
  analogWrite(7, 0);   // pump 05
  analogWrite(8, 0);   // pump 04
  analogWrite(9, 0);   // pump 03
  analogWrite(10, 0);  // pump 02
  analogWrite(11, 0);  // pump 01
}

/*
 * Sets the max values of all pumps to the pump with the lowest height on the max value.
 * for testing pumps
```

```
 */
void pumpsHigh() {
  analogWrite(0, 220);   // pump 12
  analogWrite(1, 255);   // pump 11
  analogWrite(2, 245);   // pump 10
  analogWrite(3, 245);   // pump 09
  analogWrite(4, 225);   // pump 08
  analogWrite(5, 255);   // pump 07
  analogWrite(6, 220);   // pump 06
  analogWrite(7, 245);   // pump 05
  analogWrite(8, 235);   // pump 04
  analogWrite(9, 225);   // pump 03
  analogWrite(10, 255);  // pump 02
  analogWrite(11, 220);  // pump 01
}

/*
 * Averages the first 36 out of 128 bins in the FFT to 12 and places the values into FFTarrary
 * to access.
 */
void averageFFT() {
  for (int i = 0; i < 12; i++) {
    if (i == 11)  // (4 Khz - 6 khz | presence range of frequencies)
    {
      FFTarray[i] = fft256.read(10, 23) / 12;
    } else  //all other bins from 0-10 are not averaged (344 hz - 1.9 Khz | low to mid range f
    {
      FFTarray[i] = fft256.read((i - 1) + 1);
    }
  }
}
```

*2) **LEDStrip.ino**:*

```
/*****************************************************************************
 *
 * File:      LEDStrip.ino
 * Author:    Alex Gray & Beverly Yee
 * Date:      August 27, 2023
 *
 * Anything and everything to do with the LED strip.
 *
 *****************************************************************************/
#include <FastLED.h>

// ================ Teensy 4.0 Stuff ================
// check the list of FastLED Parallel Output pins to get the correct one to use
#define DATA_PIN 20
#define NUM_LEDS_PER_STRIP 14
#define NUM_STRIPS 1
#define NUM_LEDS NUM_LEDS_PER_STRIP
#define STEPS 300

// declare the array of LEDs
CRGB leds[NUM_LEDS_PER_STRIP * NUM_STRIPS];

// array of colors to use on the strip
CRGB clist[10] = { CRGB::Red, CRGB::Lime, CRGB::Magenta, CRGB::Blue, CRGB::OrangeRed,
                   CRGB::ForestGreen, CRGB::Crimson, CRGB::Gold, CRGB::Cyan, CRGB::White };

// length of the color array
int arrLength;
// variables for ease of access on random colors and clearing
CRGB previous, clear, current;

// ================ Delay counters ================
int lineUpDelay;
int cycleColorsDelay, cycleColorsLength, cycleColorsEnd;
int fadeLength, fadeInOutEnd, fadeDelay, fadeCount;
int incrementFillDelay, incrementFillLength, incrementFillEnd, incrementFillCount;
int incrementColorLength, incrementColorEnd, incrementColorCount;
int wipeDelay, wipeEnd, wipeCount, wipeLength;
int chaseDelay, chaseLength, chaseCount, chaseColor, chaseEnd;
int cycleDelay, cycleRainbowEnd;

bool fadeOutYN, showColor, switchLED;

static float colorIndexFull = 0.25;

// master items - DO NOT CHANGE
int delayCount, colorNum, LEDNum, counter, iterateCount;



/*
 * Setup the LED strip to be called by WaterFountainMain.setup();
 */
void setupLED() {
```

```
  Serial.begin(9600);

  // get the arrLength of the color list
  arrLength = sizeof(clist) / sizeof(clist[0]);
  clear = CRGB::Black;
  previous = clear;

  // add the LED strip
  FastLED.addLeds<NUM_STRIPS, WS2811, DATA_PIN, GRB>(leds, NUM_LEDS_PER_STRIP);

  // initially set it to black
  clearStrip();

  // connect to an unconnected pin to reset random function
  randomSeed(analogRead(16));

  resetCounts();
  setupDelays();
}

/*
 *
 */
void setupDelays() {
  // line up test delay
  lineUpDelay = 2000;

  // cycle all colors calculations
  cycleColorsDelay = 600;
  cycleColorsLength = cycleColorsDelay * arrLength;

  // fade in out calculations
  fadeDelay = 4;
  fadeLength = (fadeDelay * 255) * 2;  // 255 = max brightness, x2 to account for in then out
  fadeCount = 3;                       // iterations of fade

  // increment fill to black calculations
  incrementFillDelay = 50;
  incrementFillLength = (incrementFillDelay * NUM_LEDS) * 2;  // x2 to do color then black
  incrementFillCount = 3;

  // increment fill color to color
  incrementColorLength = (incrementFillDelay * NUM_LEDS);
  incrementColorCount = 6;

  // single LED color wipe
  wipeDelay = 30;
  wipeLength = (wipeDelay * NUM_LEDS);
  wipeCount = 5;

  // chase/switching LEDs
  chaseDelay = 70;  // delay between the odd and even LEDs
  chaseCount = 9;  // how many times to show before switching colors
  chaseLength = chaseDelay * (chaseCount * 2);
  chaseColor = 4;  // number of colors to show
```

```
  // rainbow cycle calculations
  cycleDelay = 5000;
}

/*
 *
 */
void resetCounts() {
  // master counter for cycleAllPatterns
  counter = 0;

  // master delay counter
  delayCount = 0;
  iterateCount = 0;

  // LED strip index
  LEDNum = 0;

  // color array index
  colorNum = 0;

  // used for increment to fill then to black functions
  // must reset to true to show the color
  showColor = true;

  // end of pattern variables
  cycleColorsEnd = 0;
  fadeInOutEnd = 0;
  incrementFillEnd = 0;
  incrementColorEnd = 0;
  wipeEnd = 0;
  chaseEnd = 0;
  cycleRainbowEnd = 0;

  // this one may not matter, but good for initialization
  switchLED = true;
}

/*
 *
 */
void cycleAllPatterns() {
  if (counter < lineUpDelay) {
    lineUpTest();
  } else if (counter == lineUpDelay) {
    // SETUP lineUpTest -> cycleAllColors setup
    cycleColorsEnd = lineUpDelay + cycleColorsLength;
    delayCount = 0;
  } else if (lineUpDelay < counter && counter < cycleColorsEnd) {
    // cycle through all colors
    cycleAllColors(cycleColorsDelay);
  } else if (counter == cycleColorsEnd) {
    // SETUP cycleAllColors -> fadeInOut
    fadeInOutEnd = cycleColorsEnd + (fadeLength * fadeCount);
    delayCount = 0;
  } else if (cycleColorsEnd < counter && counter < fadeInOutEnd) {
```

```
    // fade in/out random colors
    fadeInOut(fadeDelay);
  } else if (counter == fadeInOutEnd) {
    // SETUP fadeInOut -> incrementFilltoBlack
    clearStrip();  // need to reset the brightness after fade items
    incrementFillEnd = fadeInOutEnd + (incrementFillLength * incrementFillCount);
    delayCount = 0;
  } else if (fadeInOutEnd < counter && counter < incrementFillEnd) {
    // increment fill then to black
    incrementFillToBlack(incrementFillDelay);
  } else if (counter == incrementFillEnd) {
    // SETUP incrementFillToBlack -> incrementColorFill
    incrementColorEnd = incrementFillEnd + (incrementColorLength * incrementColorCount);
    delayCount = 0;
  } else if (incrementFillEnd < counter && counter < incrementColorEnd) {
    // increment color fill without clearing
    incrementColorFill(incrementFillDelay);
  } else if (counter == incrementColorEnd) {
    // SETUP incrementColorFill -> singleColorWipe
    delayCount = 0;
    wipeEnd = incrementColorEnd + (wipeLength * wipeCount);
  } else if (incrementColorEnd < counter && counter < wipeEnd) {
    // only one LED is on at all times
    singleColorWipe(wipeDelay);
  } else if (counter == wipeEnd) {
    // SETUP singleColorWipe -> chaseRandom
    delayCount = 0;
    LEDNum = 0;
    chaseEnd = wipeEnd + (chaseLength * chaseColor);
    clearStrip();
    getRandomColor();
  } else if (wipeEnd < counter && counter < chaseEnd) {
    // switch LEDs are on at a given time
    chaseRandom(chaseDelay, chaseCount);
  } else if (counter == chaseEnd) {
    // SETUP chaseRandom -> fullCycle
    delayCount = 0;
    cycleRainbowEnd = chaseEnd + cycleDelay;
  } else if(chaseEnd < counter && counter < cycleRainbowEnd) {
    // full strip blend into next color, rainbow style
    fillCycle(colorIndexFull);
  }
  // reset everything
  else {
    resetCounts();
    clearStrip();
  }

  // increase the count with each iteration of the main loop
  counter++;
}

// ================ pattern functions ================
/*
 * Assign each LED with a different color from clist.
 * A test to see how the LEDs are addressable and if they all work.
```

```
 */
void lineUpTest() {
  int c = 0;
  for (int i = 0; i < NUM_LEDS; i++) {
    // since the arrLength of the color list is less than the number of LEDs we have
    if (c >= arrLength) {
      c = 0;
    }

    leds[i] = clist[c];
    c++;
  }

  FastLED.show();
}

/*
 * Cycles through all the colors of the color array
 *
 * params:  wait - how long the counter iterates before switching colors
 *                 Simulates a delay
 */
void cycleAllColors(int wait) {
  if (delayCount == wait) {
    // reset delay count when the delay is met
    delayCount = 0;
    // update the index to the color array
    colorNum++;
  }

  if (colorNum == arrLength) {
    // reset colorNum to the beginning of the array
    colorNum = 0;
  }

  fill_solid(leds, NUM_LEDS, clist[colorNum]);
  FastLED.show();
  delayCount++;
}

/*
 * Fade into color and fade out into black. When fading back into color,
 * change to a different color, chosen at random.
 *
 * params:  wait - the time between adjustments of brightness level
 */
void fadeInOut(int wait) {
  // the arrLength of half of the pattern
  int bsDelay = wait * 255;

  // utilize a boolean to determine whether to fade in or out
  if (delayCount == bsDelay) {
    // start fade out when hitting the peak of pattern
    fadeOutYN = true;
  } else if (delayCount == 0) {
    // flip the boolean to fade back into color
```

```
    fadeOutYN = false;
    // swap the color
    getRandomColor();
  }

  if (fadeOutYN) {
    // fading out, so decrement
    delayCount--;
  } else {
    delayCount++;
  }

  fill_solid(leds, NUM_LEDS, current);

  // delayCount is going to be higher than 255, so divide by wait
  FastLED.setBrightness(delayCount / wait);
  FastLED.show();
}

/*
 * Fade into color and fade out into black.
 * Override of the random color version for single color only.
 *
 * params:  wait - the time between adjustments of brightness level
 *          index - the number attached to the color in the array
 */
void fadeInOut(int wait, int index) {
  int bsDelay = wait * 255;
  if (delayCount == bsDelay) {
    fadeOutYN = true;
  } else if (delayCount == 0) {
    fadeOutYN = false;
  }

  if (fadeOutYN) {
    // fading out, so decrement
    delayCount--;
  } else {
    delayCount++;
  }

  fill_solid(leds, NUM_LEDS, clist[index]);
  FastLED.setBrightness(delayCount / wait);
  FastLED.show();
}

/*
 * Fill the entire strip with a singluar color.
 * After "wait" is met, use a different color.
 *
 * params:  wait - the time between the next color
 */
void showSingleFill(int wait) {
  if (delayCount == wait) {
    delayCount = 0;
    getRandomColor();
```

```
  }

  fill_solid(leds, NUM_LEDS, current);
  FastLED.show();

  delayCount++;
}

/*
 * Fill the entire strip with a singular color, passed in as an argument.
 *
 * params:  index - the number attached to the color in the array.
 */
void showSolid(int index) {
  fill_solid(leds, NUM_LEDS, clist[index]);
  FastLED.show();
}

/*
 * Incrementally fill the LED strip with a color.
 * The next time the strip fills, it will be a different color.
 * i.e., after the LED is filled with a color, it doesn't clear.
 *
 * params:    wait - how long to delay the function before continuing
 */
void incrementColorFill(int wait) {
  // the time between lighting the previous LED to lighting the next LED
  int fillDelay = wait * NUM_LEDS;

  if (delayCount == fillDelay) {
    delayCount = 0;
  }

  if (delayCount == 0) {
    // change the color at the start of the new pattern
    // this also allows a random color at startup
    getRandomColor();
  }

  // the delayCount works on the index of the strip
  leds[delayCount / wait] = current;
  FastLED.show();

  delayCount++;
}

/*
 * Incrementally fill the LED strip with a color then increment fill to black.
 * When the pattern executes again, it will be a different color.
 *
 * params:    index - number attached to a color in clist
 *            wait - how long to delay the function before continuing
 */
void incrementFillToBlack(int wait) {
  // the time between lighting the previous LED to lighting the next LED
  int fillDelay = wait * NUM_LEDS;
```

```
  if (delayCount == fillDelay) {
    // flip the bool to start the other part of the pattern
    showColor = !showColor;
    getRandomColor();
    delayCount = 0;
  }

  // show the color
  if (showColor) {
    // the delayCount works on the index of the strip
    leds[delayCount / wait] = current;
    FastLED.show();
  } else {
    // when showColor == false, clear it
    leds[delayCount / wait] = clear;
    FastLED.show();
  }

  delayCount++;
}

/*
 * params:    wait - delay between switching the LEDs
 *            index - number of the color attached to the array
 */
void chaseIndex(int wait, int index) {
  if (delayCount == wait) {
    clearStrip();
    // change the LEDs that light up
    switchLED = !switchLED;
    delayCount = 0;
  }

  if (switchLED) {
    // turn on the even numbered LEDs
    for (int i = 0; i < NUM_LEDS; i += 2) {
      leds[i] = clist[index];
    }
  } else {
    for (int i = 1; i < NUM_LEDS; i += 2) {
      leds[i] = clist[index];
    }
  }

  FastLED.show();
  delayCount++;
}

/*
 * params:    wait - delay between switching the LEDs
 *            iterate - how many times to iterate before changing colors
 */
void chaseRandom(int wait, int iterate) {
  int chaseCount = iterate * 2;  // x2 to account for the full strip (half otherwise)
  if (iterateCount == chaseCount) {
```

```
    // reset the count and get another color
    iterateCount = 0;
    getRandomColor();
  }

  // delay between odd and even LED indexes
  if (delayCount == wait) {
    // change the LEDs that light up
    switchLED = !switchLED;
    iterateCount++;  // one side is done, iterate
    delayCount = 0;
  }

  if (switchLED) {
    // turn on the even numbered LEDs
    for (int i = 0; i < NUM_LEDS; i += 2) {
      leds[i] = current;
      leds[i + 1] = clear;
    }
  } else {
    for (int i = 1; i < NUM_LEDS; i += 2) {
      leds[i] = current;
      leds[i - 1] = clear;
    }
  }

  FastLED.show();
  delayCount++;
}

/*
 * Incrementally fill the LED strip with a color then increment fill to black.
 *
 * params:    wait - how long to delay the function before continuing
 *            index - number attached to the color in the array
 */
void incrementFillToBlack(int wait, int index) {
  // the time between lighting the previous LED to lighting the next LED
  int fillDelay = wait * NUM_LEDS;

  if (delayCount == fillDelay) {
    // flip the bool to start the other part of the pattern
    showColor = !showColor;
    delayCount = 0;
  }

  // show the color
  if (showColor) {
    // the delayCount works on the index of the strip
    leds[delayCount / wait] = clist[index];
    FastLED.show();
  } else {
    // when showColor == false, clear it
    leds[delayCount / wait] = clear;
    FastLED.show();
  }
```

```
  delayCount++;
}

/*
 * Only one LED is illuminated the entire time and runs across the strip.
 *
 * params:    index - number attached to a color in clist
 *            wait - how long to delay the function before continuing
 */
void singleColorWipe(int index, int wait) {

  if (delayCount == wait) {
    // delay met, turn off the current LED
    leds[LEDNum] = clear;
    FastLED.show();
    // and prepare to light the next one
    LEDNum++;
    // and reset the delayCount
    delayCount = 0;
  }

  if (LEDNum == NUM_LEDS) {
    LEDNum = 0;  // reset the index as to not go out of bounds
  }

  // make this an else statement?
  leds[LEDNum] = clist[index];
  FastLED.show();

  delayCount++;
}

/*
 * Only one LED is illuminated the entire time and runs across the strip.
 *
 * params:    wait - how long to delay the function before continuing
 */
void singleColorWipe(int wait) {
  if (delayCount == wait) {
    // delay met, turn off the current LED
    leds[LEDNum] = clear;
    FastLED.show();
    // and prepare to light the next one
    LEDNum++;
    // and reset the delayCount
    delayCount = 0;
  }

  if (LEDNum == NUM_LEDS) {
    LEDNum = 0;  // reset the index as to not go out of bounds
    getRandomColor();
  }

  // make this an else statement?
  leds[LEDNum] = current;
```

```
  FastLED.show();

  delayCount++;
}

/*
 * Fills the entire strip with one color and blends it to the next color.
 * Goes through the colors of the rainbow.
 */
void fillCycle(uint8_t colorIndex) {
  // for (int i = 0; i < NUM_LEDS_PER_STRIP; i++) {
  //   leds[i] = ColorFromPalette(RainbowColors_p, colorIndex, 255, LINEARBLEND);
  // }

  fill_solid(leds, NUM_LEDS, ColorFromPalette(RainbowColors_p, colorIndex, 255, LINEARBLEND));

  FastLED.show();
}

/*
 * The entire strip is colored as a rainbow with each color blending into the next.
 * At the same time, the rainbow moves along the strip.
 */
void fillRainbowCycle() {
  float colorIndex = 0.5;
  for (int i = 0; i < NUM_LEDS_PER_STRIP; i++) {
    leds[i] = ColorFromPalette(RainbowColors_p, colorIndex, 255, LINEARBLEND);
    colorIndex += STEPS;
  }
  FastLED.show();
}


// ================ helper functions ================
/*
 * Fill the entire strip with black to clear the strip.
 */
void clearStrip() {
  fill_solid(leds, NUM_LEDS, CRGB::Black);
  FastLED.setBrightness(255);
  FastLED.show();
}

/*
 * Helper function to grab a random color then determine whether it is the same as
 * the previously generated color. While it is still the same, keep randomizing
 * until the current color is different than the previous.
 */
void getRandomColor() {
  current = clist[random(arrLength)];

  // change this into a while loop in case more than 2 in a row are the same numbers?
  while (current == previous) {
    // grab another random color if it ends up being the same as the previous color
    current = clist[random(arrLength)];
  }
```

```
  // set the previous color to the current color so the next call will have a different color
  // regardless of where it is called from
  previous = current;
}

/*************/

/*
 * Only two LEDs is illuminated the entire time and runs across the strip.
 *
 * params:    index1 - number attached to a color in clist
              index2 - number attached to a color in clist, follows index1
 *            wait - how long to delay the function before continuing
 */
void doubleColorWipe(int index1, int index2, int wait) {
  for (int i = 0; i < NUM_LEDS; i++) {
    leds[i] = clist[index1];
    leds[i - 1] = clist[index2];
    FastLED.delay(wait);
    leds[i - 1] = clear;
    leds[i] = clear;
    FastLED.show();
  }
}

void fullColorWipe(int wait) {
  int c = 0;
  for (int i = 0; i < NUM_LEDS; i++) {
    for (int j = NUM_LEDS; j > 0; j--) {
      leds[j - i] = clist[c];
      if (c >= arrLength) {
        c = 9;
      }
      c++;
    }

    FastLED.show();
    FastLED.delay(wait);
  }

  // for (int i = 0; i < NUM_LEDS; i++) {
  //   for (int j = arrLength - 1; j >= 0; j++) {
  //     leds[i - j] = CRGB::White;
  //   }
  //   FastLED.show();
  // }
}
```

E. Schematics

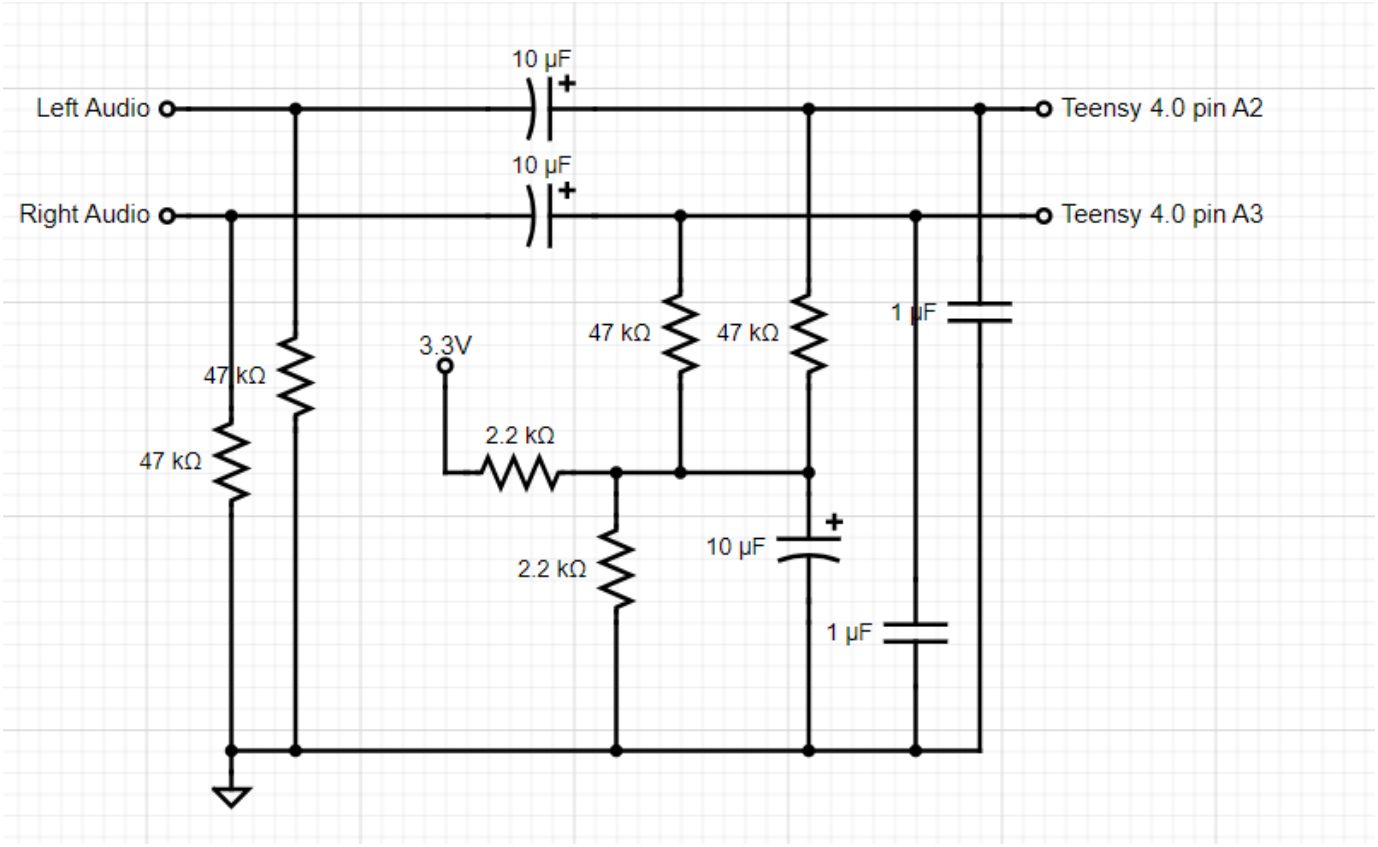


Fig. E.1. Audio input circuit

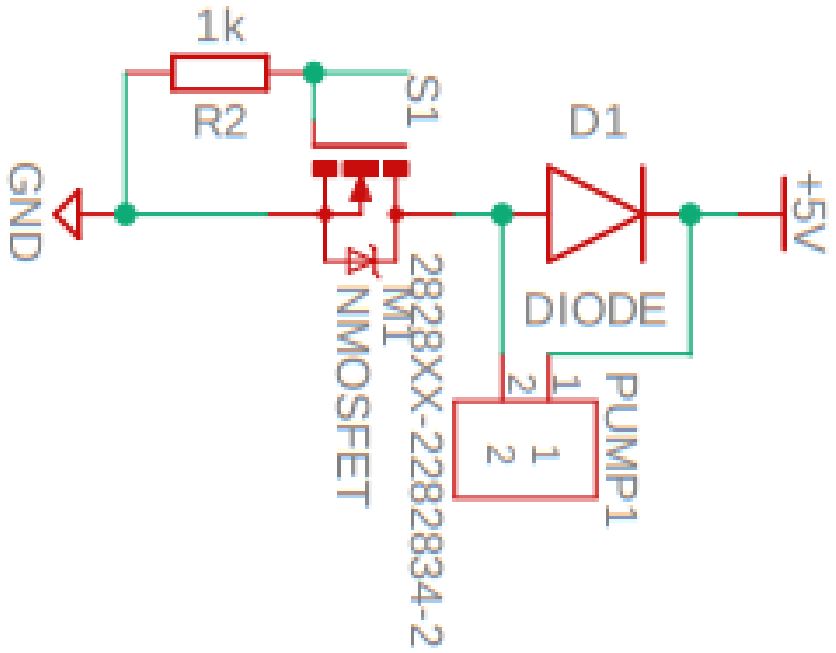


Fig. E.2. Circuit schematic for a singular pump used for the prototype, based on https://youtu.be/UPTU6nYSaMo?si=Bcf-2Wz6ASSrwG9J

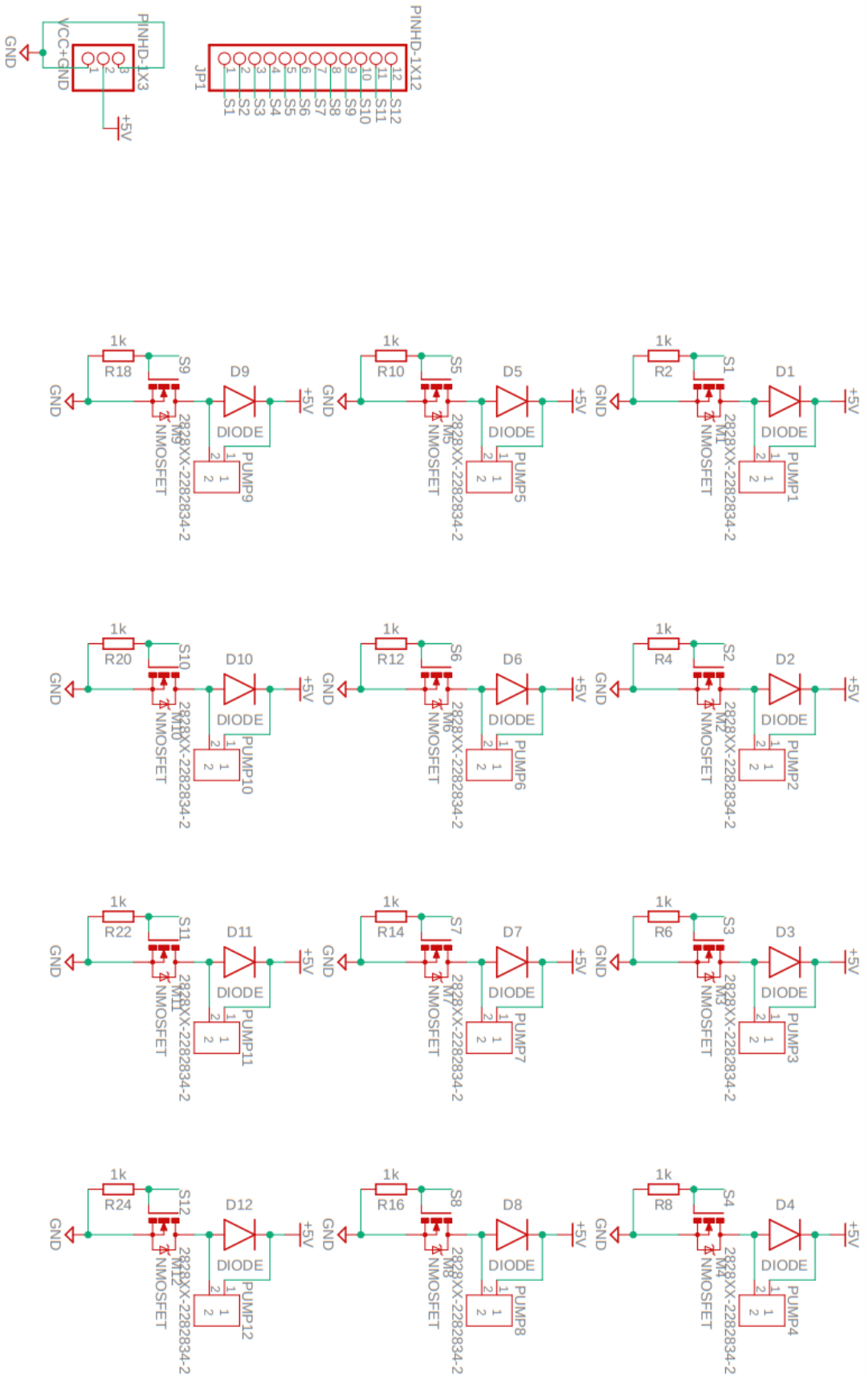


Fig. E.3. Pump Driver Schematic

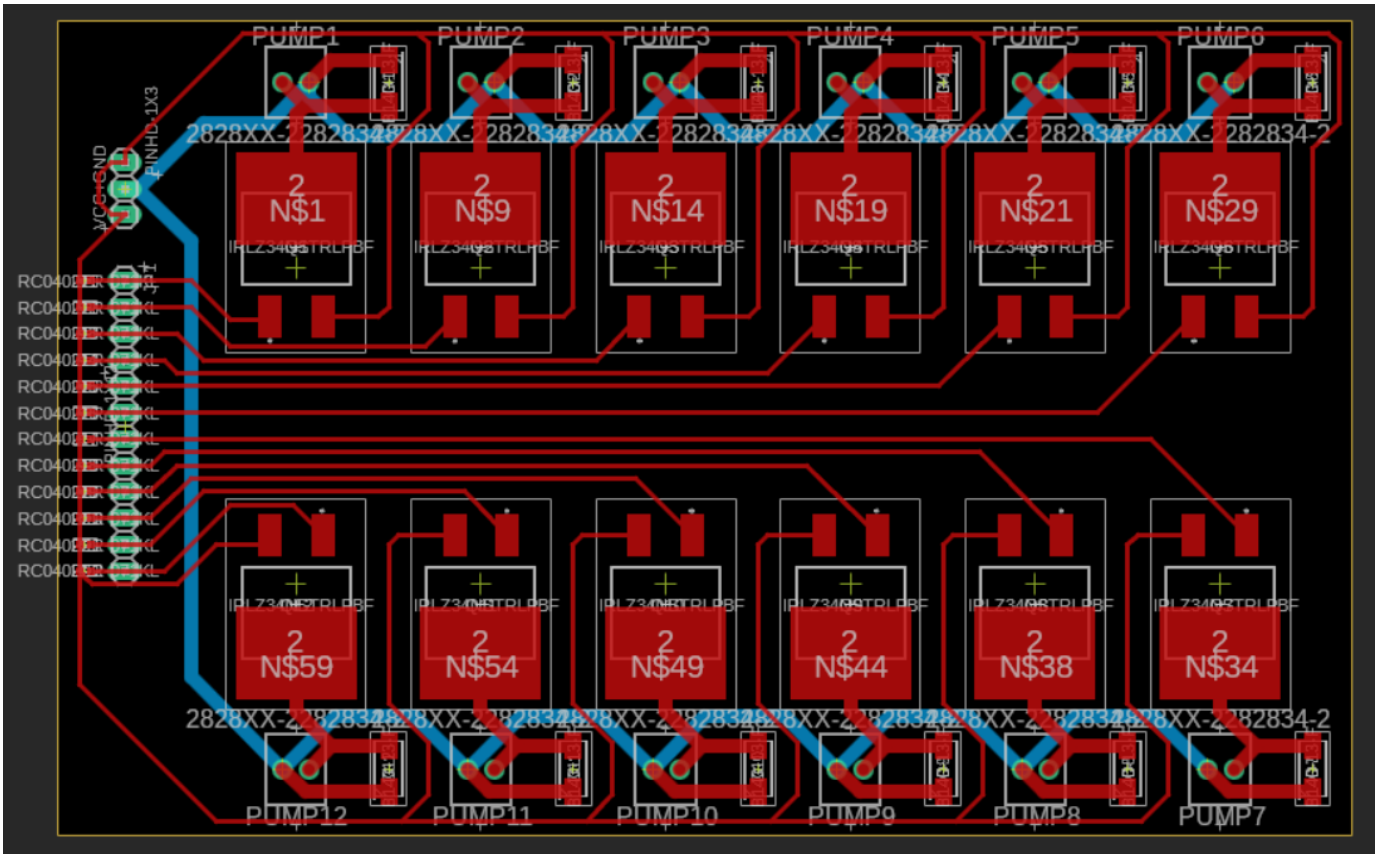


Fig. E.4. Pump Driver PCB

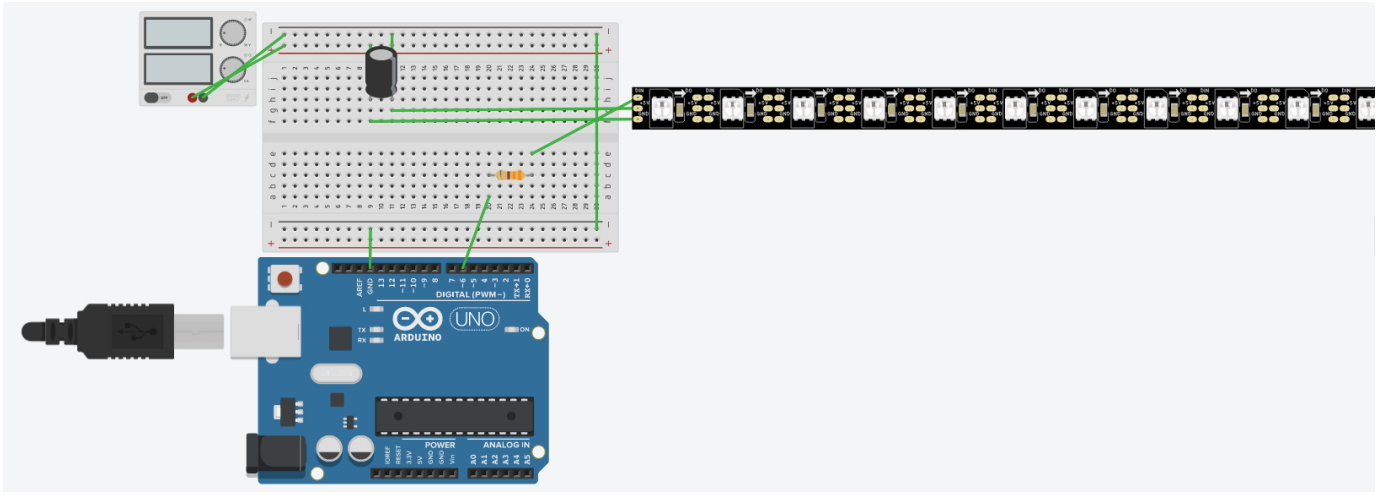


Fig. E.5. Best practices for NeoPixels–attach a capacitor over power and ground and a resistor between the microcontroller and the LED. An Arduino Uno is used in place of the Teensy here. The block in the upper left corner represents the LED power supply.