Fademirror

Our goal for this project was to create a device that was interactive over the internet. We wanted to create something from scratch, programming the server and the device ourselves. We decided to create an infinity mirror that is controlled by a user over wifi, connecting to a locally hosted web app.

An infinity mirror creates an effect that makes it seem like there are infinite recursions of light going ‘backwards’ into the mirror. This effect is created by placing a regular mirror on the bottom, an LED strip in the middle, and a two way mirror on top. A two way mirror reflects 70% of the light while letting around 30% through. With a setup like this, 30% of the light will be released from the LEDs, while 70% will be reflected backwards, onto the regular mirror, and back towards the top mirror again, where 30% will make it through and 70% etc... This causes an infinite recursion effect. The frame from the mirror was purchased, stained, cut and put together ourselves. We drilled the frame together with screws to hold everything in place.

The Fademirror project consists of four major parts: the aforementioned infinity mirror and its hardware, our Python library that controls the hardware, a Python Flask web application that provides an interface for the user and translates input into commands for the hardware controller library, and a traditional frontend that leverages the use of WebSockets in JavaScript to provide interactivity to the user. This is a layered architecture, as each layer can only interact with those adjacent to it:

Fademirror

Web Application Interface (HTML, CSS, JavaScript)
HTTP / WebSocket Server (Python)
Hardware Control Library (Python)
Infinity Mirror Hardware (LED strips, FadeCandy Controller)

We used a Raspberry Pi to run our code. To communicate with the Fadecandy Controller, we had to configure it to run the fadecandy-server which adafruit provides on their website. This opens up the Fadecandy Controller, when plugged into the Raspberry Pi through USB, to communication using the Open Pixel Control Library which we will discuss more about later. A Raspberry Pi also allows us to run a Flask server.

The web application interface consists of a traditional HTML server with navigation and is built out with Twitter's Bootstrap 4. However, the main component of the user interface is that it is mobile-oriented, meaning that the interface is easy-to-use on a mobile device, and responds to touch and drag motions, where applicable. For the scope of this project, the objective was to create an interface that provided what we called a "map" which is a touchpad, of sorts. The map responds to the user's drag motions and maps the locations touched by the user's map motion to a set range. In addition to this, the user is provided with several options (a mode dropdown, a length for lights to last before they fade, and a color selector) that allows for customization of the desired effect on the mirror. The web application interface also contains the SocketIO library, which allows for simple connection via WebSockets to servers running the corresponding server-side library. What happens on the interface is that the user's touches, color selection, and other data are sent on-the-fly to the WebSocket on the connected server, which are then handled by the server's WebSocket server component.

Fademirror's server uses two components: an HTTP server and a WebSocket server. The HTTP server serves static resources, such as JavaScript and CSS, and also serves HTML pages to the user, which achieves the goals of getting our interface to the user. However, the main code that drives the server lies in the WebSocket server, which handles JSON sent to it by the user via the web application interface's WebSocket. The advantage of using this WebSocket is that the amount of requests made and the speed they can be made at are greatly increased, as it bypasses the traditional GET, POST protocols in favor of raw JSON. The server takes this data, as fast as the user's client is configured to send it, and places it in a Data Transfer Object (DTO) for further transport along the layers of the program's architecture. The DTO is simply a class, but it acts as a lingua franca, as all components of the application have components that handle the same data structure. The JSON passed by the interface is packaged into this DTO and then passed along to the next layer, the hardware control library.

The hardware control library (which could be considered the other half of the project) is the main driver of the hardware. This half of the project consisted of setting up communication between the Raspberry Pi and the addressable LED strip. We used Adafruit Fadecandy as an intermediary that helps dampen/smooth changes in the light's colors. Open Pixel Control (OPC) is a library that makes it easy to communicate with Adafruit fadecandy. It allows you to simply pass an array of RGB values into it and handles the communication with the addressable LED strip for you. Therefore, our main work was translating the info received from the Flask server into an array of RGB values that the OPC library can handle.

The lights on the mirror need to be updated every frame so we created an update loop thread. This update loop recalculates positions and fade of the lights on the mirror, which we will

discuss more in depth shortly. After recalculating, it converts all the lights into a single array and sends to the OPC client. The update loop looping frequency can be configured in order to optimize performance, but we found a speed of every 0.1 milliseconds worked well.

We created two main classes for the types of lighting features we wanted to allow, “Light” and “Wave”. Light is a class that can be instantiated for any static positioned colored light. It also takes a time-to-live value so it can slowly fade out. When the python flask server receives an event of type ‘dot’ in JSON, it translates that event to a position on the mirror and calls a function that constructs a Light object for that event. In the update loop, all Light objects are dimmed each frame according to the time passed since the last update. Wave is a class that is similar to Light, but handles light position changing as well. Wave can be thought of as a group of lights but is really more advanced than that. It has a radius and a fade radius. The radius is how long the wave is, and the fade radius is how long the fade effect lasts on each end of the wave. Its position gets updated in the update loop so the waves can move around the mirror. When the Flask server receives an event of type wave it will create a wave object using the other specified options.

All of the Light and Wave objects need to be converted into a single array of RGB values for the OPC library. This happens every update loop with a process that we will refer to as “Layering”. The update loop loops through every Light object, and those with the same position the RGB values will be layered on top of each other. So, for example, if the light objects consisted of Lights with the same position but different colors

A:

RGB: (255,0,0)

Position: 34

B:

RGB(0,0,255)

Position: 34

The update loop's final array for the OPC library would have the color of RGB:(255,0,255). Lights A and B would be layered on top of each other. The same process happens for the waves, with all of the positions being layered. This creates cool effects as waves pass over static lights on the mirror.

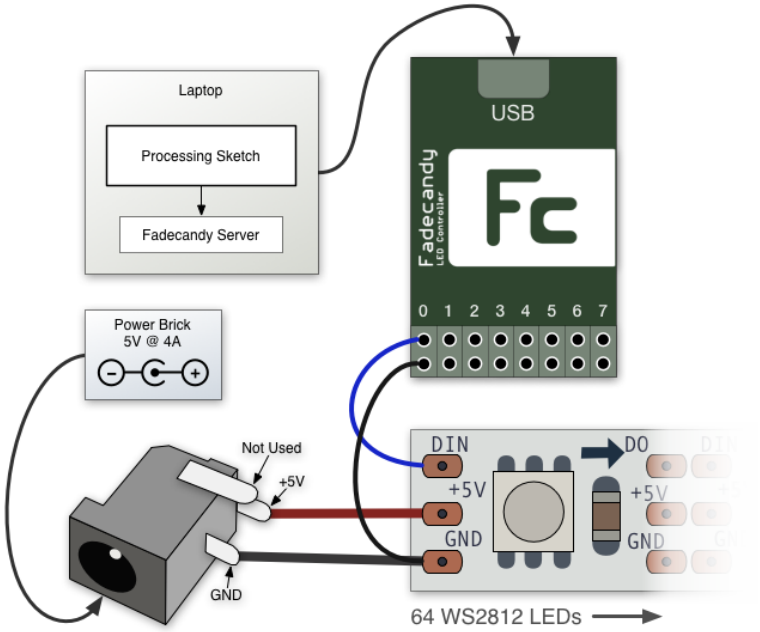
Another part of the update loop in converting to OPC is positioning. The OPC library takes an array of length 64x8 for the 64 possible LEDs on the 8 possible channels. This means indices 0,64,128,186,etc are the first indices of each channel. Our mirror consisted of 4 strands and 35 lights, so we must convert from an array of 35x4 to an array of 64x8. This was easily done with offsets in the code. We ran into a minor issue with our Fadecandy channels. 2 of the channels did not work - channels 1 and 4. Therefore, we also had to provide offsets to make sure our array of 35x4 (length 140) went into the channels 0,2,3,5, or indices 0-34,128-163,192-227, and 325-369 of the final OPC array.

The LED strips we used in the hardware are addressable, meaning each individual LED on the strips can be manipulated to be any color. We cut the strips to fit the mirror, 4 strands of

35 leds. We then soldered wires onto the strip, drilled holes in the frame, and used gorilla glue to hold the LEDs in place between the two mirrors.

For wiring the LEDs, we used a breadboard. We used 3 connections on each LED strip, ground, 5v, and Data-in (DIN). The ground and 5v each got their own group on the breadboard, while the DIN from each strand had to be connected to a channel in Adafruit's Fadecandy Controller. Adafruit's Fadecandy Controller is used to help dampen/smooth changes to the LEDs, which makes our mirror look a lot smoother when we have waves running through it or lights fading out.

Figure 1. Wiring LEDs to Fadecandy



https://learn.adafruit.com/led-art-with-fadecandy/wiring-your-leds

The last thing connected to the breadboard was a 5v power supply. The 5v power supply is barely enough to power the 4 led strips, sometimes even resulting in dimmer lights than

desired. If this project was recreated, we would consider buying a different power supply. The Fadecandy Controller doesn't run our code, rather it connects to a device that must be running some form of operating system.

Making internet devices today is fairly common and therefore a lot of tools exist to help. We utilized these tools and successfully created an infinity mirror that can be controlled over wifi using python, Flask server, a Raspberry Pi, and an Adafruit Fadecandy Controller.