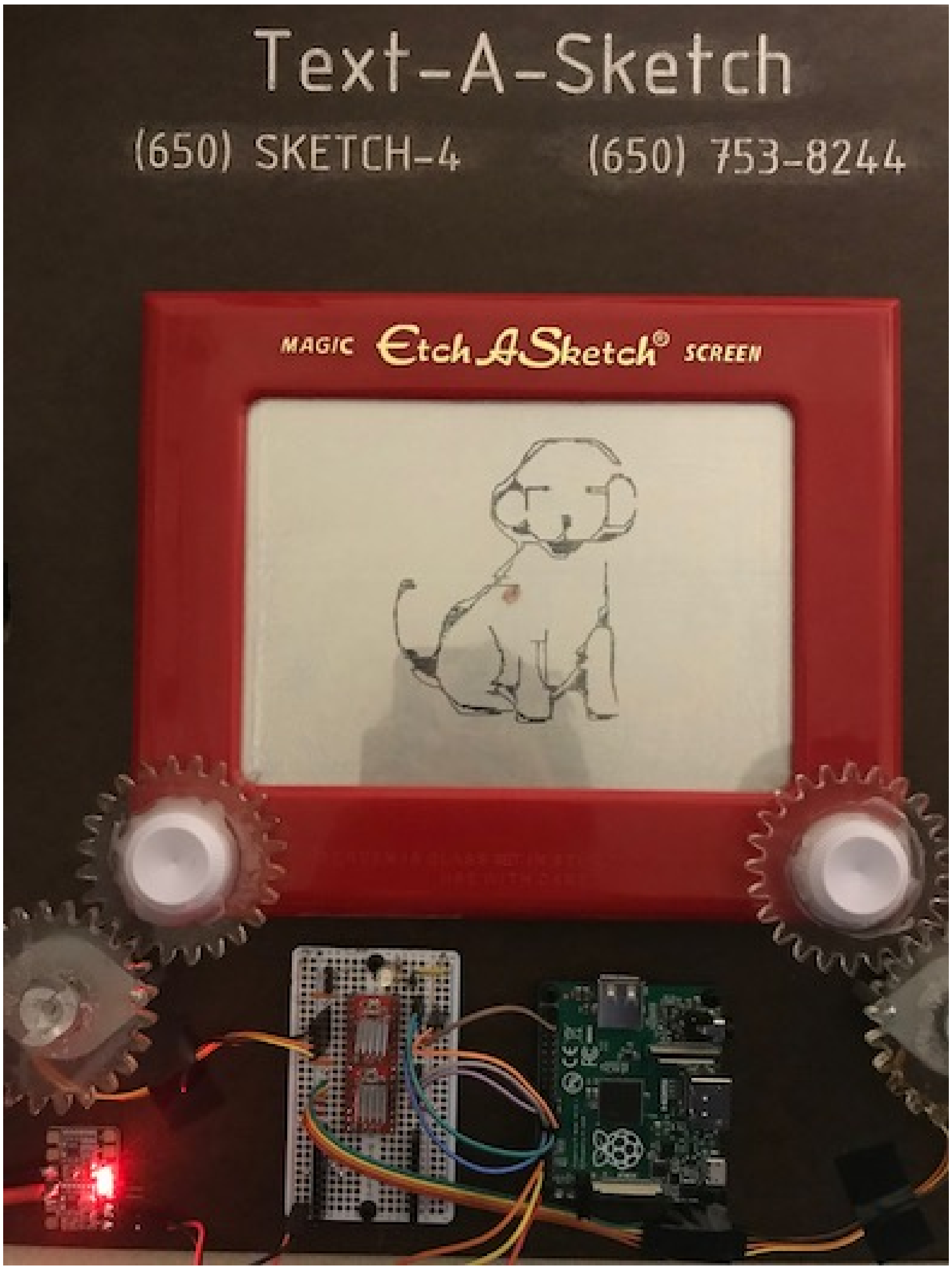
Text-A-Sketch

Mihir Garimella (`mihirg`), Kathy Huang (`khuang2`), Nikhil Cheerla (`nikcheerla`)

Project: Build an integrated, self-contained system that draws line sketches of images on the Etch-A-Sketch whenever users text it at a specified number.



Responsibilities:

- Mihir: Hardware, ESP8266 breakout board, computer vision pre-processing, backend (Twilio + REST endpoints + queue), ESP8266 downloading + sending images to Pi
- Kathy: Hardware, CAD for laser-cut parts, PCB soldering, stepper motor calibration + control code for drawing
- Nikhil: Hardware, computer vision pre-processing, computer vision drawing code for Pi (based on Kruskal's algorithm + tree traversal)

Hardware Process

Laser cutting gears and base

The hardware setup involved attaching two stepper motors to the two knobs on the Etch-A-Sketch. In order to turn the motors, we decided that gears would be the best option. We learned to create vector files using Adobe Illustrator in order to laser cut the gears and the base. For the base, we precisely measured the ridge on the bottom of an Etch-A-Sketch and cut an exact hole in the base to fit the Etch-A-Sketch. We struggled to make a vector file that had the correct gear design. As a solution, we found an automatic spur gear generator on Fusion 360, a CAD program. We generated the 3D gear, then created a special sketch of the 2D gear to be saved as a vector file. We initially cut the gears out of duron (the same used for the base), but these did not mesh very well because of the high-friction nature of the duron material and the root fillet radius, which we discovered by fiddling around on Fusion. In our second, working iteration of gears, we made the size smaller and changed the root fillet radius to be smaller and cut in acrylic material.

Attaching gears

After we cut the gears, we had to attach one set of gears to the Etch-A-Sketch knobs and another set to the stepper motors, and then make sure that these two sets of gears were level. We initially attached the gears via hot glue to the knob and to metal hubs attached to the motors—we also attached these hubs to the motor shafts using hot glue. This leveling process required drilling holes for standoffs for the motors and carefully leveling the gears and fine-tuning while screwing in the motors. We then hooked up the motors with jumper wires to motor drivers on a breadboard. When testing these motors, we struggled a lot with the breaking of the hot glue, particularly on the connection between the shaft and the hub. This was a difficult problem to diagnose because it presented itself as a loss of torque in the motor during testing because the hub would slip from the shaft. After redoing the hot glue multiple times, we decided to create a less, breakable connection using epoxy. The epoxy we had access to required a 1-hour work time and an 8-hour hardening time. During the 1-hour work time, we created supports for the gears using modeling clay to hold the gear/hub of the motors level with the gears attached on the Etch-A-Sketch. After drying, the epoxy worked significantly better than the hot glue. This was a significant advantage in the software-writing process because the broken hot glue would be difficult to diagnose, so when the motors loss power, we first had to examine all possible broken components from a blown motor driver, not enough current through motor driver, etc.

Soldering to PCB

Even with the epoxy, we still had some problems with our hardware setup. Due to the short length of the battery leads, they would often partially come out of the breadboard and could sometimes touch each other or other wires and cause a short, burning out the breadboard itself. This was a difficult problem to diagnose the two times it happened because the breadboard was the part we least expected to be broken; we tested and replaced all of the other components (the motor and motor drivers) before testing with a multimeter to see that the power rails on the breadboard were not working. After the second broken breadboard, we decided to permanently solder our connections onto a PCB (the Adafruit Perma-Proto 1/2-size PCB, to be precise). This proved difficult because the PRL was closed, but we were ultimately able to track down a soldering iron and make the PCB.

Building a breakout board for the ESP8266

We decided to use the ESP8266 WiFi chip to connect our project to the internet to receive text messages. However, one problem with the ESP8266 is that it isn't breadboard-friendly; it has two rows of four headers that all have different functions. To fix this, we made a custom breakout board for the ESP8266 using an Adafruit Perma-Proto 1/4-size PCB. The breakout board includes a socket for the chip and exposes various connections for power, TX, and RX from the ESP8266 in a more breadboard-friendly layout through headers sticking out the bottom. The breakout board also includes a 6-pin FTDI header (so that we could flash code onto the ESP8266 separately, since it's really a small microcontroller with built-in WiFi and can run Arduino code), an easy way to set the ESP8266 into flash mode (simple jumper connection from GPIO0 on the ESP8266 to a GND female header we exposed), and a pushbutton, which is also exposed through the pins on the bottom of the breakout board. We added a socket for this board onto our main PCB and broke out some of the connections (the TX and RX of the ESP8266 and the button input) to male header pins to connect to the Raspberry Pi. You can see the mini-breakout board in our pictures; it's the little white board with the ESP8266, a button, and headers cantilevered off of the main PCB.

Screwing pieces into place, motor troubles

After soldering the PCB, we still needed to bolt everything into the base in order to start developing and testing our software. Prior to screwing everything in, the variability in the relative positioning of the Pi, PCB, battery, and motors would cause tenuous electrical and physical connections that would break when testing. Thus, we had to drill the holes to hold all of these components and screw them in before testing. Another challenge we faced was setting the current limit for our stepper motor driver. We learned quickly that the default current through the driver (500mA, which is what our motors were rated for) would not generate enough torque in the motor to actually turn the knob. Thus, we had to manually tune the current on both motor drivers until the motor had enough torque. This current required was very high and more than the motors should have taken safely; however, we needed to do this to actually turn the Etch-A-Sketch knobs. This resulted in the motors heating up immensely when running and emitting a slightly high pitched noise, but we brushed it off in the interest of making the project work. After attaching all the hardware components to the base, we had to run calibration tests to find the number of steps for the width and height of the Etch-A-Sketch. This number first resulted in 420 steps wide and 600 steps high, which did not make much sense aspect ratio-wise. However, we realized that one motor was moving in half-steps while the other was moving in quarter-steps; when we fixed this, our new numbers were 840 steps wide by 600 steps high. We ran this testing a number of times and got the same numbers. Also, during testing, we found the motors to turn very slowly which would cause the drawing process to take a very long time. In order to speed up this process, we not only changed the software to minimize the delay between steps for the motor but also changed the hardware. We lowered the resolution slightly (our tests should no noticeable difference to the naked eye) by changing our initial setting of quarter-steps on the motors to half-steps. As opposed to simply changing the software to make the motor take two quarter-steps for every step, we decided to change the hardware so the motor would only have to move once instead of twice for every step, speeding up the process. This changed required undoing the previous soldering connection and creating a new one.

Powering everything

We also realized very quickly that we couldn't power the stepper motors from the Raspberry Pi's 5V; we needed a 12V supply to maximize the torque they could put out. To solve this, we used a 3-cell high-current LiPo battery (meant for a quadcopter) and connected it to the power rails on one side of the PCB through a power distribution board. The power distribution board has a special polarized connector for the LiPo and breaks this out into wires that connect to the PCB. We drilled a groove into our duron base for a battery strap so that we could secure the battery nicely into our prototype, and hot glued the power distribution board in the appropriate position. One issue that we had with the power distribution board was that it claimed to have a 5V switching regulator, which would have been perfect to power our Pi; unfortunately, this regulator didn't work and didn't output any voltage, so we decided to power the Pi either through USB or a wall adapter. We connected the Pi's ground to the battery's ground so that all of our components would be operating relative to a common reference voltage.

Final Touches

Since the laser cut letters on our base were quite faint in appearance, we decided to put chalk inside the letters then clean off the surface to only leave the chalked-in letters for greater readability.

Development

Although it would have been easy to do a lot of the necessary drawing logic on an external server, we wanted to make the Pi do as much algorithmic work as possible, in order to develop our bare-metal programming skills and enable a more general "graphics library" style interface. In general, we do a limited amount of pre-processing in the cloud to receive images, detect edges, and compress them to a size that can quickly be sent over UART to the Pi; on the Pi, we do some fairly intensive graph-based processing to prepare and draw the images.

External

We needed to write a ton of code to get the Pi to receive images from text messages. We set up an account on Twilio to take advantage of their MMS API, buying a cool phone number (650-SKETCH-4) for $1 a month and signing up for a pay-per-message plan.

Next, we created a server with two REST endpoints and deployed it to Heroku so that it would be visible from any computer/device in order for clients (like our WiFi chip or Twilio's backend) to be able to push and pull image data easily. We wrote a Python backend app using Flask and deployed it using a server stack called gunicorn. Our app exposed a `/message` endpoint that Twilio could POST incoming messages to (our code then makes sure that each message contains an image, downloads that image, pre-processes it a bit, compresses it, stores it in a queue, and then texts the user back to let them know their position in the queue) and a `/image` endpoint that our WiFi chip could GET from that would dequeue one image, return its data in compressed format as a long string (in which each character represents 8 pixels' data), and text the user who sent that image alerting them that we're now drawing their picture. After some mysterious HTTP problems that occurred when multiple images were texted at similar times and/or our WiFi chip requested multiple images in quick succession, we eventually used a thread-safe queue data structure to store images.

Next, we used our WiFi chip, the ESP8266, to query our server and download images whenever prompted (the Raspberry Pi ended up polling a big green button and prompting the ESP8266 over UART for a new image whenever the button was pressed). The chip connected to WiFi (we got it working with the Stanford Residences network!), called our `/image` endpoint, and forwarded the response over UART to the Pi. We initially ran into some errors wherein the ESP8266 didn't have enough memory to download the entire image response before forwarding it over; we tried several approaches, including digging into the ESP8266 HTTPRequest code to enable downloading the data as a stream and forwarding it over UART chunk-by-chunk (for some reason, we received too many chunks when we tried this and couldn't figure out exactly which chunks contained our image data), but none of them panned out, so we just decided to limit the image resolution we were using to a size that, when compressed, the ESP8266 could fit in memory.

As an alternative for our demo, we also wrote a Python script that replicated what the ESP8266 did in case we had trouble connecting the chip to WiFi in the Gates basement—the script similarly waited for a prompt, queried our server, and forwarded the response using the same protocol (described below). This script proved to be hugely helpful, since we couldn't find any WiFi networks in the Gates basement that were both unsecured (or had just a single password rather than a username/password combination) and didn't have a "captive portal" that a human could navigate easily but that would pose difficulty for our embedded chip; in Gates, we ended up using our Python script instead on a laptop connected to Stanford Visitor.

To actually send the images over UART from the ESP8266 (or Python script) to the Pi, we first sent a byte indicating status (either '<', indicating that we're about to send an image, '+', indicating that no images are available on the server, or '-', indicating that we couldn't connect to WiFi). Then, we transmit the data using the XMODEM protocol, similar to Julie's bootloader code (this was really helpful for us; we used `rpi-install.py` as a reference to write our Python script and used `bootloader.c` as a reference to write the image receiving code on the Pi). We wrote XMODEM sending code from scratch for the ESP8266 as the "opposite" of Julie's receiving code in `bootloader.c`. Ultimately, our solution of compressing the images on the server was great because it limited the size of the data that we had to store in memory on the ESP8266 while buffering our UART writing and the data that we actually had to send over UART; our final compressed edge bitmap images took 1/24th the memory of the original RGB images we received.

Internal

On the Pi, we receive a compressed image bitmap using XMODEM receiving code and decompress it into a 2D image. We then turn this image into a graph in which nodes are black "edge" pixels so that we can compute the shortest path to sketch between those nodes. Since the Etch-A-Sketch can only draw one continuous line, we apply Kruskal's minimum spanning tree algorithm to connect edges, looking at the pairwise distances between edge pixels to determine which connections between nodes to include. We spent a ton of time optimizing the Kruskal's code to run as quickly as possible to minimize the wait time needed before the Pi can actually start drawing a picture. In addition, implementing Kruskal's on a bare-metal system was very difficult since we didn't have access to standard libraries (e.g., we had to implement our own ”argsort” function from scratch that sorted one array based on the contents of another corresponding-indexed array) and data structures (e.g., hash tables or variable length arrays would have made the implementation considerably easier).

We then start a tree traversal from the center of the Etch-A-Sketch screen, rerooting the tree at this center (because trees can be arbitrarily rooted), visiting every node in the tree in a preorder fashion, and returning back to the starting point. In this process, we trace out every point on the image as well as some low-cost connections required to draw them in one continuous line.