Final Project

CS 498 IOT FALL 2020

team *return to_sleep;*

JJ Urgello (urgello2)
Vasundhra Agarwal (va6)
Vishesh Gupta (vvgupta2)

Motivation

When Vishesh was a child he owned a fish. One day, when he got back from a movie, he discovered that the fish had unfortunately died due to lack of food. While discussing project ideas, JJ informed us that he currently owns a fish, and Vishesh ended up sharing his experience. This is when JJ suggested how nice it would have been to have an automatic, scheduled pet feeder. On hearing this idea, Vasundhra and Vishesh decided that it would be perfect to create a fish feeder using the IoT skills from this semester so that such an incident does not happen with JJ's fish. JJ does not want his pet fish to die. This was the motivation that led us to decide that we will be creating an-

IoT Fish Feeder.

This project would allow us to feed the fish based on a pre-decided schedule that will be input by the user. In addition to feeding the fish on a schedule, the user will also be able to manually feed the fish through a web interface on their preferred device. The user can also monitor the fish via a live video feed on the interface, and a motion detection software will be used to highlight the fish on the feed and to help make judgements for the feeding routine.

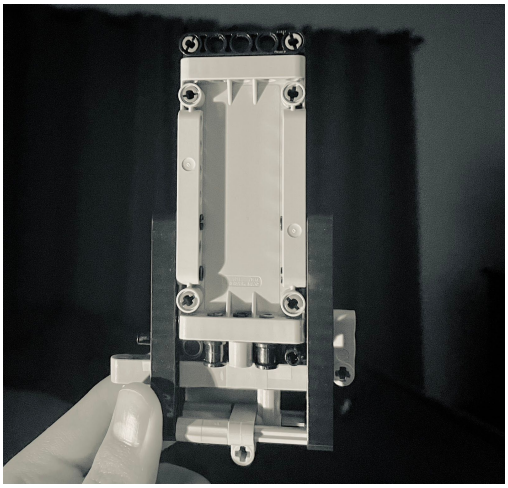
While this project is being made specifically for a fish, it can also be expanded for use for other pets. We believe that it is easy to forget to feed your pets sometimes or other situations might occur that will prevent people from feeding their fish. Therefore, our project intends to provide a solution to people so that their pet is well-fed and happy. Additionally, some people might be stuck away from their pets due to the pandemic, and we believe that the live video feed will allow them to feel connected to their pet over long distances.

Technical Approach: Feeder Hardware

Implementing the Feeder Mechanism

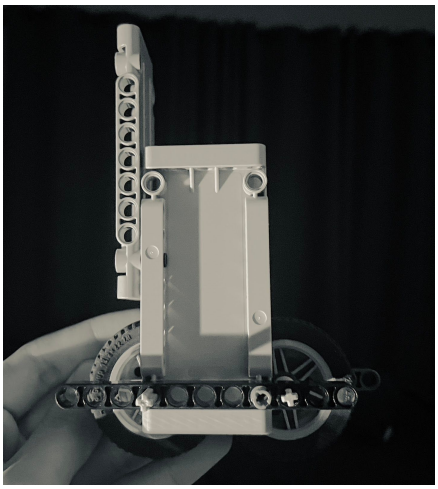
We wanted a feeder mechanism made specific to the Betta feed being used. The feeder mechanism underwent several iterations. None of us have access to a 3D printer, which would have made this step significantly more precise (and less annoying), but using Lego Pieces and tools around the house made this step interesting and very rewarding considering the constraints. This mechanism would be very difficult to make as these Betta Fish pellets are smaller than that of a grain of rice. These pellets are about a millimeter wide.

1) First prototype: Pellets pass through a sliding shuttle at the bottom of the contraption:



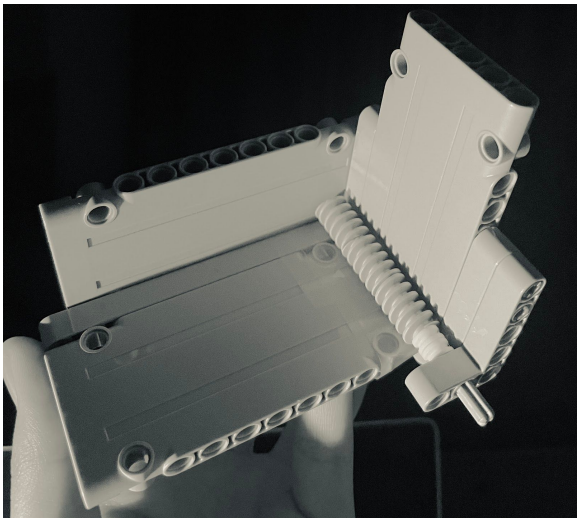
Issue: Pellets slip out of the cracks at the bottom, and the shuttle is unreliable and clunky.

2) Second prototype: Pellets squeezed through two wheels that spin slowly:



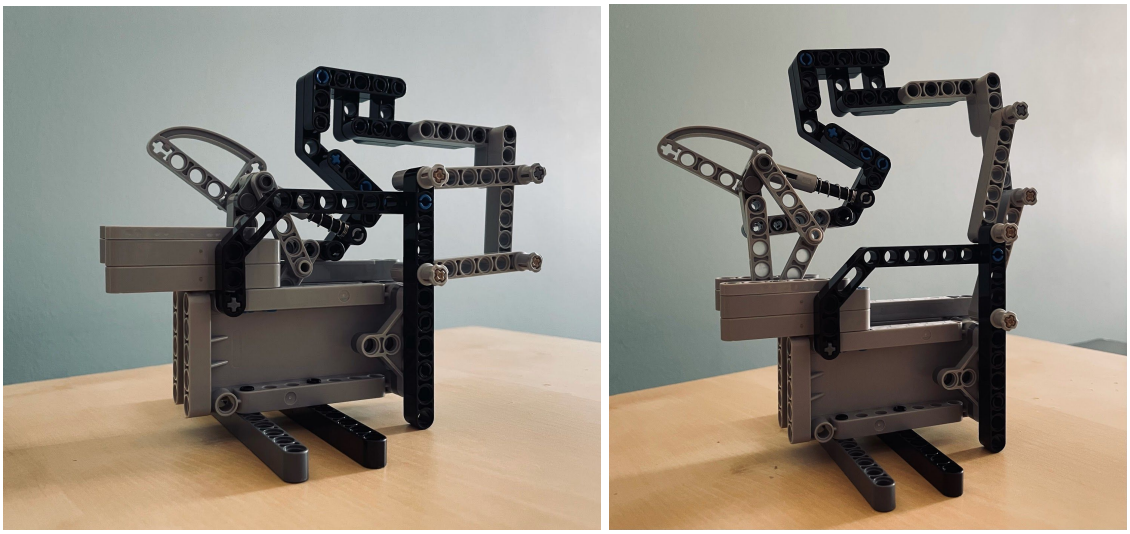
Issue: Pellets slip out of cracks between the wheels and plates, and it is difficult to control how many pellets are released at once, and it is too difficult to turn the wheels.

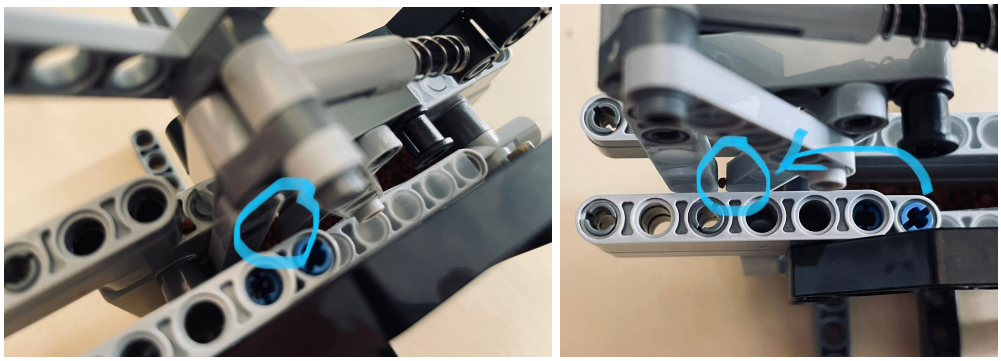
3) Third prototype: Pellets pulled at a diagonal up through worm gears:



Issue: Worm gear crushes the pellets into smaller pieces, and the pellets get trapped inside the worm gear.

4) Fourth prototype: Over-complicated arm system that picks up a single pellet:



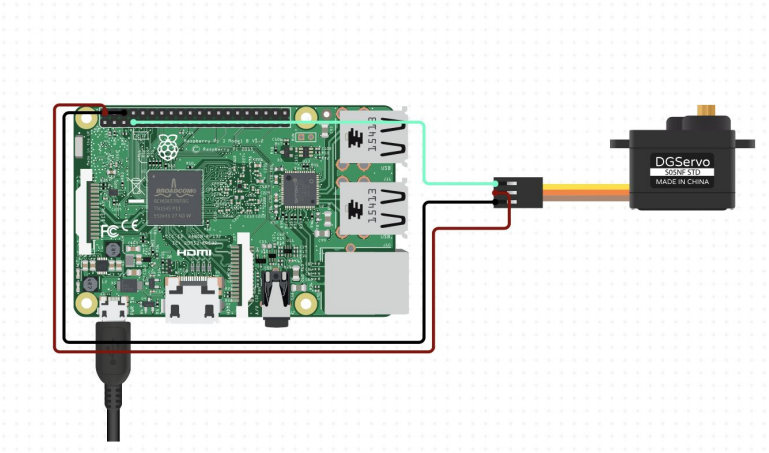


Issue: Over-complicated, difficult to refill pellets, requires too much movement and control, is unsteady and not perfectly consistent.

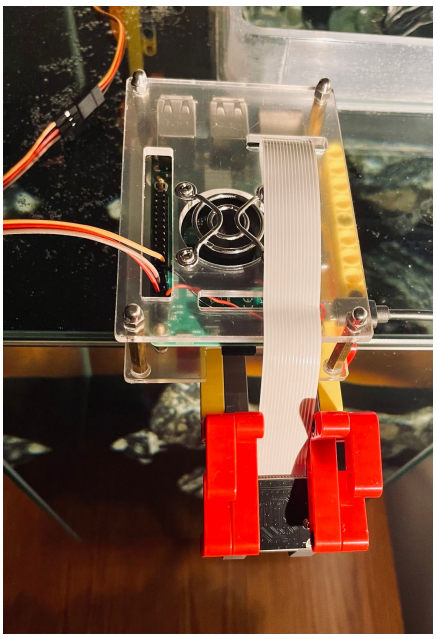
5) Fifth and final prototype: Rotating drum with scooper straw that picks up 1-2 grains every time:

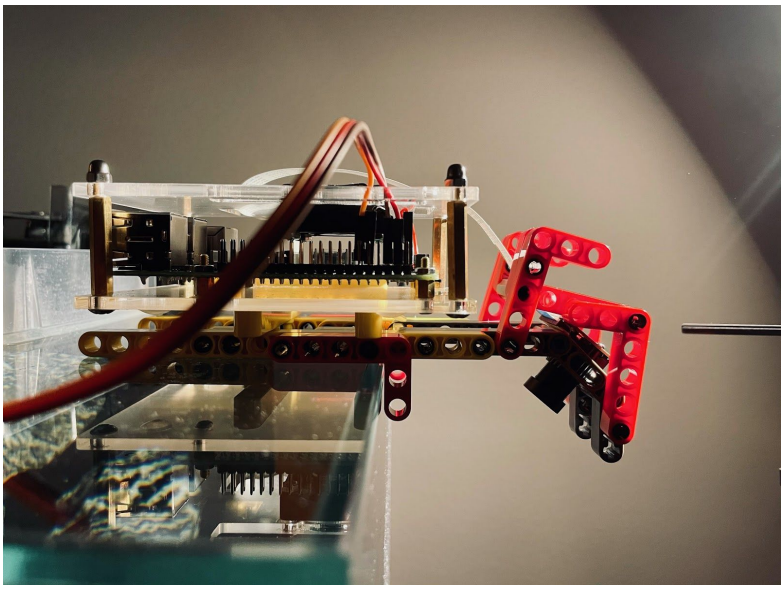
Topology

The electrical components of this project are simple as the project only requires a servo, camera, and Raspberry Pi. The camera is the Raspberry Pi Camera module, and the servo is a generic micro servo.

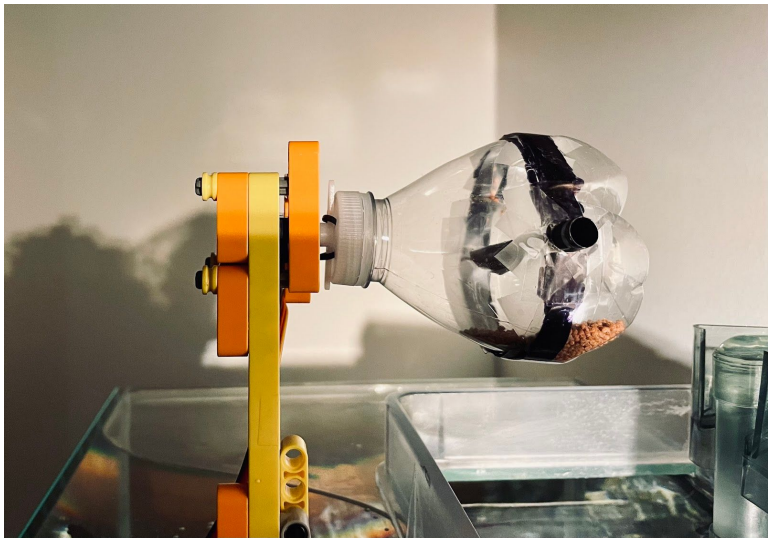


The following shows the Raspberry Pi connected to the camera module (encased in red Legos):





The following shows the feeding drum connected to the motor (the servo is encased in the yellow/orange Legos):



Implementation Details: Feeder Software

Software Overview

The feeder is designed to take requests from both the command line and/or from websites. The central object of the software is the Fish. A Fish keeps track of all of the fish's attributes necessary for maintaining proper feeding. The Feeder class manages the servo hardware feeding the food. The controller file starts off the entire process. To allow multiple tasks to be handled at once, the controller creates two threads. One thread handles the web server, while the other handles the automatic feeder schedule. Furthermore, the web server creates another two threads to run Flask and run motion detection.

The scheduling thread runs the schedule, periodically checking the current time against the assigned feeding time. If the current time is a feeding time, a request is made to feed. The Fish object does several tests for whether or not it is appropriate to feed at this time, and if so, the Fish object requests the Feeder object to start the servo routine. The scheduling thread also checks if the day has ended. If the day has ended, the statistics for the day are collected and set appropriately. These statistics include times fed, and some states of the fish such as hunger and death.

The web application thread runs a Flask application that shows some current stats of the fish (hunger, times fed, is dead). There are two buttons on this app: "feed" and "refresh". The "feed" button sends a request to feed the fish, and the refresh button refreshes the stats of the fish. There is also a live video feed from the Raspberry Pi camera, and this video feed includes motion detection software that draws a box around a detected fish. The website was designed to work on any set of devices in the home and at any time.

The motion detection thread uses the difference between consecutive frames to determine whether the fish has moved. The video feed and motion detection are designed to pause while the servos are running to prevent an overload of the Raspberry Pi. The motion detection algorithm also sets the moving flag of the Fish object, which will ultimately be used to determine if the fish has died (if the fish did not move that day).

Feeder Class

The Feeder class manages the servo. The code for controlling the servo was trivial (complexity mostly came out of logic for feeding times and checks to prevent overfeeding/underfeeding). The purpose of the time.sleep calls are to give the servo some time to reach its requested position. The duty cycle requests to 0 are to prevent the servo from jittering after a position is reached.

```
import RPi.GPIO as GPIO
import time

class Feeder():
    def __init__(self):
        # Set GPIO number mode
        GPIO.setmode(GPIO.BOARD)
        # Set pin 11 as output, and set servo as pin 11 as PWM
        GPIO.setup(11, GPIO.OUT)
        self.servo = GPIO.PWM(11, 50)
        self.servo.start(0)
        time.sleep(0.7)
        self.servo.ChangeDutyCycle(0.0)
        time.sleep(0.7)

    def set_from(self, a, b):
        time.sleep(1.0)
        while a != b:
            if a < b: a += 5
            else: a -= 5
            self.servo.ChangeDutyCycle(a)
            time.sleep(0.3)
            self.servo.ChangeDutyCycle(0)
            time.sleep(0.7)

    def run(self):
        self.set_from(7, 2)
        self.set_from(2, 12)
        self.set_from(12, 7)

    def cleanup(self):
        self.servo.stop()
        GPIO.cleanup()
```

Fish Class

The Fish class handles the maintenance of the fish with the attributes of hunger, times fed, is dead, feeder (the Feeder object), current period (current day range in days for a feeding session), current time, moved (whether or not the fish has moved this day), and feeding (whether or not the Feeder object is performing a feeding routine).

```python
from settings import *

class Fish():
    def __init__(self, hungry, times_fed, is_dead, feeder):
        self.hungry = hungry
        self.times_fed = times_fed # per period
        self.is_dead = is_dead
        self.feeder = feeder

        self.current_period = 0 # in days
        self.current_time = 0 # hh:mm
        self.moved = False
        self.feeding = False

    def new_day(self):
        # summarize today
        self.current_period += 1
        self.is_dead = not self.moved
        self.print_stats()

        # for next day
        self.moved = False
        if (self.current_period >= FEEDPERIOD):
            self.current_period = 0
            self.times_fed = 0
            self.update_hunger()

    def update_hunger(self):
        self.hungry = (self.times_fed < MAXFEED and
                not self.is_dead)

    def feed(self):
        print("FEED REQUESTED")
        self.update_hunger()
        if (self.hungry and not self.feeding):
            self.feeding = True
            self.feeder.run()
            self.times_fed += 1
            self.update_hunger()
            print("FEED DELIVERED")
            self.feeding = False
        else:
            print("FEED DENIED")

    def print_stats(self):
        print("------------------------")
        print("  FEEDTIME: " + str(FEEDTIME))
        print("  current time: " + str(self.current_time))
        print("  FEED PERIOD: " + str(FEEDPERIOD))
        print("  current period: " + str(self.current_period))
        print("  MAX FEED: " + str(MAXFEED))
        print("  times fed: " + str(self.times_fed))
        print("  hungry: " + str(self.hungry))
        print("  moved: " + str(self.moved))
        print("  is dead: " + str(self.is_dead))
        print("------------------------")
```

The Fish class accepts requests to feed and does a validation check before making a request to the Feeder object.

Controller Code

The controller houses the scheduling routine, but the controller more importantly starts the process by creating threads for the two main tasks of the feeder project: automatic feeding and the website that handles both hosting a video stream and manual feeding.

Here is the segment of the controller that begins the process by creating two threads:

```
41  def begin_routine():
42      feeder = Feeder()
43      exit = threading.Event()
44      jello = Fish(True, 0, False, feeder)
45
46      task1 = threading.Thread(target=feed_schedule, args=(jello,exit,))
47      task2 = threading.Thread(target=run_server, args=(jello,exit,))
48
49      task1.start()
50      task2.start()
51      wait_for_user_terminate(jello, exit)
52
53      print("cleaning...")
54      feeder.cleanup()
55      task1.join()
56      task2.join()
57      print("done")
```

Note: An “exit” threading event is passed to these two threads. The exit variable is used to gracefully terminate all the threads when the feeder system is requested to shutdown from the command line. The controller also accepts different requests from the command line, and manual feeding is also possible through the command line:

```
29  def wait_for_user_terminate(fish, exit):
30      while (True):
31          prompt = input()
32          if (prompt == 'f'): fish.feed()
33          elif (prompt == 's'): fish.print_stats()
34          elif (prompt == 't'): fish.new_day()
35          elif (prompt == 'd'): break
36          else: print("...")
37          time.sleep(2)
38
39      exit.set()
```

Note: Requests are sent to the Fish object, and it is not possible to activate the Feeder servo without first consulting the Fish.

The controller's automatic feed schedule is simple:

```
def feed_schedule(fish, exit):
    while not exit.is_set():
        tm = time.strftime("%H:%M", time.localtime())
        fish.current_time = tm

        if (tm == FEEDTIME):
            fish.feed()
        elif (tm == '00:00'):
            fish.new_day()
        exit.wait(30)
```

Server Code

The server code was written in "service.py". We initially started our server code by creating a simple HTTP Server. We attempted to use http.server at first. Using http.server did not last long and only functioned as a simple test for handling requests from a website.

Resources for using python's http.server:

https://appdividend.com/2019/02/06/python-simplehttpserver-tutorial-with-example-http-request-handler/

https://docs.python.org/3/library/http.server.html

From the previous link: HTTPServer is a socketserver.TCPServer subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. BaseHTTPRequestHandler provides a number of class and instance variables, and methods for use by subclasses. It is used to handle the HTTP requests that arrive at the server. It is subclassed to handle each request method (e.g. GET or POST).

```python
from http.server import HTTPServer, BaseHTTPRequestHandler
import threading

# custom request handler
class RequestHandler(BaseHTTPRequestHandler):

    # override for custom header
    def send_response(self, code, message=None):
        self.log_request(code)
        self.send_response_only(code)
        self.send_header('Server', 'python3 http.server')
        self.send_header('Date', self.date_time_string())
        self.end_headers()
        print('')

    # response for a GET
    def do_GET(self):
        out_var = "initial"

        # send html file
        if self.path == '/':
            self.path = '/index.html'
        try:
            file_to_open = open(self.path[1:]).read()%(out_var)
            self.send_response(200)
        except:
            file_to_open = "file not found"
            self.send_response(404)

        self.end_headers()
        self.wfile.write(bytes(file_to_open, 'utf-8'))

    # response for a POST
    def do_POST(self):
        out_var = "arrived"

        # send html file
        if self.path == '/':
            self.path = '/index.html'
        try:
            file_to_open = open(self.path[1:]).read()%(out_var)
            self.send_response(200)
        except:
            file_to_open = "file not found"
            self.send_response(404)

        self.end_headers()
        self.wfile.write(bytes(file_to_open, 'utf-8'))

def run_server(exit):
    server = HTTPServer(('', 8000), RequestHandler)
    serve_thread = threading.Thread(target=server.serve_forever)
    serve_thread.start()
    print('server starting... (8000)\n')

    exit.wait()

    print('shutting down server...')
    server.shutdown()
```

Issue and Resolution: While these steps were leading to a successfully running http.server, we decided to shift from the http.server library to Flask library, as we found difficulties in adding a video stream with http.server. Flask is also popular and has good documentation.

The following resources were consulted for using Flask:

https://www.pyimagesearch.com/2019/09/02/opencv-stream-video-to-web-browser-html-page/

https://flask.palletsprojects.com/en/1.1.x/api/?highlight=run#flask.Flask.run

https://stackoverflow.com/questions/19794695/flask-python-buttons

The following snippet from the service.py code shows how the Flask application starts and sends information to the browser. The two threads are for running the Flask application and for running the motion detection algorithm.

```
# the datetime in index.html prevents caching
@app.route("/video_feed")
def video_feed():
    return Response(generate(),
            mimetype="multipart/x-mixed-replace; boundary=frame")

@app.route("/", methods=['GET', 'POST'])
def index():
    if request.method == 'POST' and request.form['request'] == 'feed':
        with lock:
            fish.feed()
    flash("hungry: " + str(fish.hungry))
    flash("times fed: " + str(fish.times_fed))
    flash("is dead: " + str(fish.is_dead))
    return render_template('index.html', rand_hash = str(datetime.datetime.now()))

def run_server(in_fish, in_exit):
    global fish, exit
    fish = in_fish
    exit = in_exit

    print('starting server...')
    time.sleep(2.0) # warmup
    task1 = threading.Thread(target=app.run,
            kwargs={'host':'0.0.0.0', 'threaded':True},
            daemon=True)
    task2 = threading.Thread(target=detect_motion, args=(32,),
            daemon=True)

    task1.start()
    task2.start()
    exit.wait()

    print('shutting down server...')
    sys.exit(0)
```

The index page of the website houses the buttons for “feed” and “refresh”, information about from the Fish object, and a video stream. The video feed is sent in video_feed() as consecutive frames from generate().

Generate takes “outframe”, which is a frame output from the motion detection function, and prepares it for the website. This function pauses when the Feeder is running a feed routine, which prevents overloading the Raspberry Pi.

```
def generate():
    global outframe, lock, exit
    while not exit.is_set():
        with lock:
            if outframe is None:
                continue
            (flag, encodedImage) = cv2.imencode(".jpg", outframe)
            if not flag: continue

        if fish.feeding: # pause video feed when feeding
            time.sleep(7.0)

        yield(b'--frame\r\n' b'Content-Type: image/jpeg\r\n\r\n' +
                bytearray(encodedImage) + b'\r\n')

```

The detect_motion function grabs a frame from the camera, performs cv2 transformations on it, and searches for motion using Detective(). The outframe variable used by the generate function above is set here. The motion detection function sleeps similarly to generate during Feeder routines.

```
def detect_motion(framecount):
    global vid, outframe, lock, exit
    det = Detective(0.1)
    total = 0

    while not exit.is_set():
        frame = vid.read()
        frame = imutils.rotate(frame, 180)
        frame = imutils.resize(frame, width=400)
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        gray = cv2.GaussianBlur(gray, (7, 7), 0)
        timestamp = datetime.datetime.now()
        cv2.putText(frame, timestamp.strftime(
            "%A %B %d %Y %I:%M:%S %p"), (70, 15),
            cv2.FONT_HERSHEY_SIMPLEX, 0.35, (0, 255, 0), 1)

        if total > framecount:
            motion = det.detect(gray)
            if motion:
                fish.moved = True
                (thresh, (minx, miny, maxx, maxy)) = motion
                cv2.rectangle(frame, (minx, miny), (maxx, maxy),
                        (0, 255, 0), 2)

        det.update(gray)
        total += 1
        if fish.feeding: # pause video feed when feeding
            time.sleep(7.0)

        with lock:
            outframe = frame.copy()
```

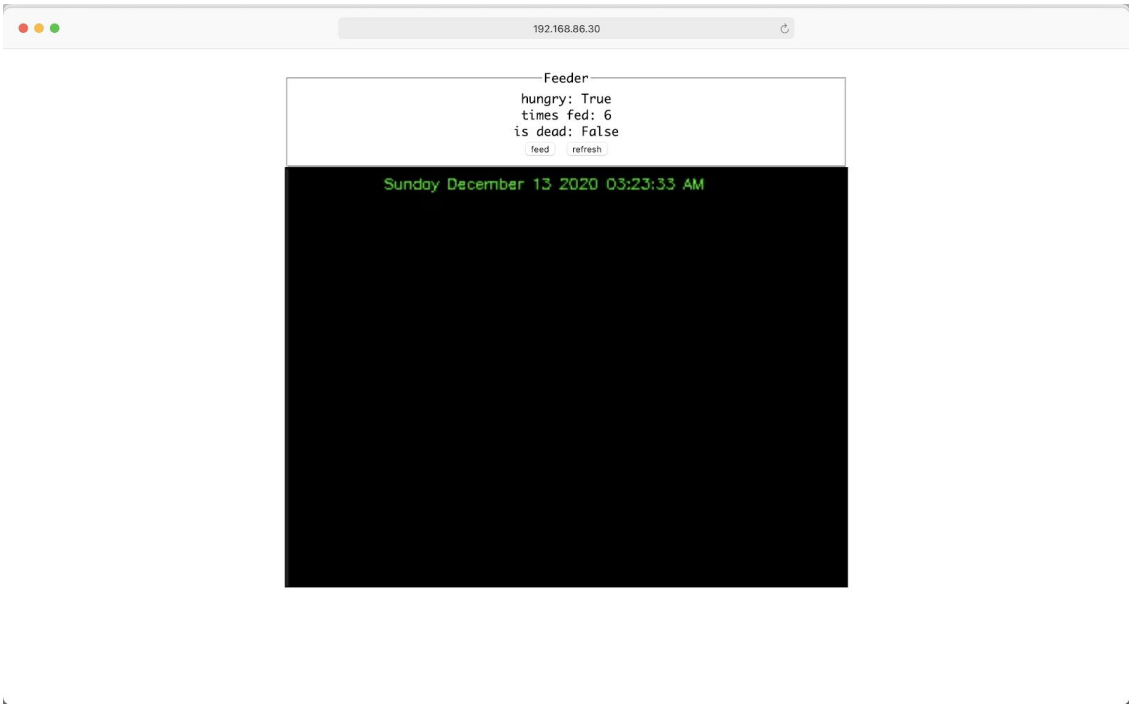
Note: The Fish attribute “moved” is updated here when motion is detected. This attribute will be read at the end of the day to determine if the fish is alive or not. If the fish has not moved for an entire day, the Feeder system determines the Fish as dead (Jello the Betta Fish is a very curious fish and swims almost all the time).

This is a snippet of the resulting index.html:

```
<html lang="en">
    <head>
        <title> feeder </title>
        <meta charset="utf-8">
    </head>

    <body>
        <form method="POST">
            <fieldset class="flashes">
                <legend> Feeder </legend>
                {% for message in get_flashed_messages() %}
                    {{message}}<br>
                {% endfor %}
                <button name="request" value="feed"> feed </button>
                <button name="request" value="refresh"> refresh </button>
            </fieldset>
        </form>
        <div class="feed">
            <img src="/video_feed?g={{rand_hash}}" >
        </div>
    </body>
```

Issue and Resolution: Note that the video feed source url has a random hash appended to it as an attribute. This is a workaround to prevent browser caching from causing the video feed to fail to display after the browser or tab is quit.

This is the result of the working Flask server and website (without motion detection displayed):



Detective Class

The Detective class handles motion detection. OpenCV was used for motion detection in this project. Initially, to download OpenCV we decided to follow:

http://www.pyimagesearch.com/2016/04/18/install-guide-raspberry-pi-3-raspbian-jessie-opencv-3/

Issue: However, this process would take several hours, and upon reaching the 'make -j4' step, the Raspberry Pi froze (even after setting the swap size).

Resolution: After trying this several times, we then decided to delete all our openCV downloads and restart with a new approach. Following the setup tutorial below was successful and took a fraction of the time:

https://pysource.com/2018/10/31/raspberry-pi-3-and-opencv-3-installation-tutorial/

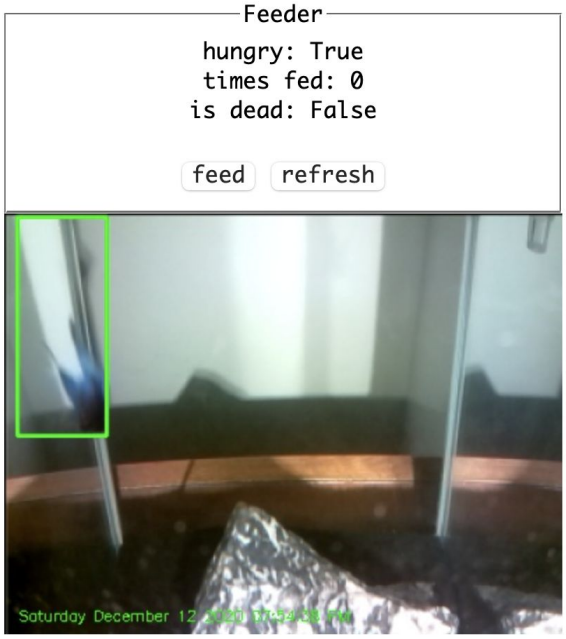
We continued to implement motion detection from the same walkthrough given for flask:

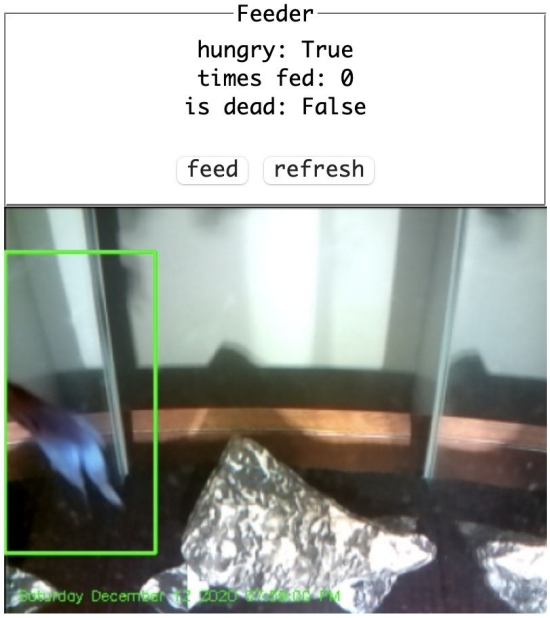
https://www.pyimagesearch.com/2019/09/02/opencv-stream-video-to-web-browser-html-page/

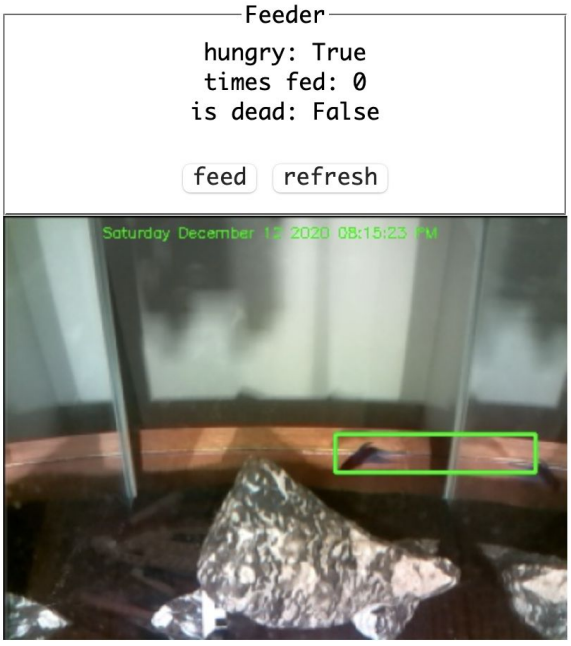
The following code snippet shows how cv2 (OpenCV) is used to run motion detection on a given image/frame. This detects motion by running a background subtraction algorithm:

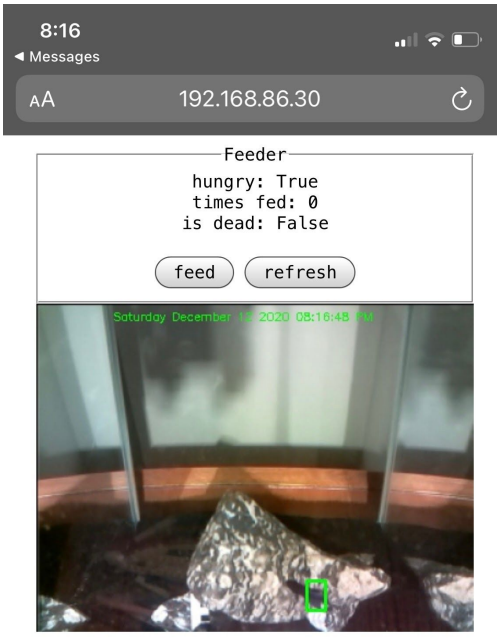
```
class Detective:
    def __init__(self, weight=0.5):
        self.weight = weight # accumulation weight
        self.back = None # background

    def update(self, im):
        if self.back is None:
            self.back = im.copy().astype("float")
            return
        cv2.accumulateWeighted(im, self.back, self.weight)

    def detect(self, im, tvar=25):
        delta = cv2.absdiff(self.back.astype("uint8"), im)
        thresh = cv2.threshold(delta, tvar, 255, cv2.THRESH_BINARY)[1]
        thresh = cv2.erode(thresh, None, iterations=2)
        thresh = cv2.dilate(thresh, None, iterations=2)

        contours = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
                cv2.CHAIN_APPROX_SIMPLE)
        contours = imutils.grab_contours(contours)
        (minx, miny) = (np.inf, np.inf)
        (maxx, maxy) = (-np.inf, -np.inf)

        if len(contours) == 0:
            return None

        for c in contours:
            (x, y, w, h) = cv2.boundingRect(c)
            (minx, miny) = (min(minx, x), min(miny, y))
            (maxx, maxy) = (max(maxx, x + w), max(maxy, y + h))

        return (thresh, (minx, miny, maxx, maxy))
```

Here are the results of the working Flask application and motion detection. These screenshots feature multiple devices throughout the household, working on both laptop browsers and smartphones:









Coordination

Issue: As noted prior, running the video stream with motion detection while running the servos caused the Raspberry Pi to drop in voltage, thus impacting the performance of the feeder and causing a mis-feed to occur.

Solution: To ensure that both the video stream and button work concurrently we took the following steps:

1. We reduced the frames per second resolution to reduce the load on the raspberry pi.
2. Using mutex locks between the threads. The mutex locks were being implemented to switch between using camera and motor for load reduction as well as for correctly allocating the “outframe” variable between the motion detector and generator.
3. The video threads will also pause when noticing that the Fish object is currently feeding.
4. Issues still persisted, and after much time we realized that the “threading” flag in Flask’s “app.run()” should have been set to True.

The following shows the entire Feeder software running in coordination (Feeds being requested and delivered while multiple devices are using the website interface). The second snippet shows the extra commands being used from the command line.

```
pijj feeder sudo python3 controller.py
===== FISH FEEDER ======
 team return to_sleep;
 CS 498 IoT FA2020
=======================
starting server...
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
192.168.86.23 - - [13/Dec/2020 01:14:00] "GET / HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:01] "GET /video_feed?g=2020-12-13%2001:14:00.921672 HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:01] "GET /apple-touch-icon-precomposed.png HTTP/1.1" 404 -
192.168.86.23 - - [13/Dec/2020 01:14:01] "GET /apple-touch-icon.png HTTP/1.1" 404 -
192.168.86.23 - - [13/Dec/2020 01:14:11] "POST / HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:11] "GET /video_feed?g=2020-12-13%2001:14:11.742663 HTTP/1.1" 200 -
FEED REQUESTED
FEED DELIVERED
192.168.86.23 - - [13/Dec/2020 01:14:19] "POST / HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:19] "GET /video_feed?g=2020-12-13%2001:14:19.728830 HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:34] "POST / HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:34] "GET /video_feed?g=2020-12-13%2001:14:34.691954 HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:36] "POST / HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:36] "GET /video_feed?g=2020-12-13%2001:14:36.528923 HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:48] "POST / HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:48] "GET /video_feed?g=2020-12-13%2001:14:48.849649 HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:53] "GET / HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:14:53] "GET /video_feed?g=2020-12-13%2001:14:53.787603 HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:15:09] "GET / HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:15:09] "GET /video_feed?g=2020-12-13%2001:15:09.086710 HTTP/1.1" 200 -
192.168.86.23 - - [13/Dec/2020 01:15:09] "GET /apple-touch-icon-precomposed.png HTTP/1.1" 404 -
192.168.86.23 - - [13/Dec/2020 01:15:09] "GET /apple-touch-icon.png HTTP/1.1" 404 -
192.168.86.48 - - [13/Dec/2020 01:15:22] "GET / HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:15:22] "GET /video_feed?g=2020-12-13%2001:15:22.144483 HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:15:22] "GET /apple-touch-icon-120x120-precomposed.png HTTP/1.1" 404 -
192.168.86.48 - - [13/Dec/2020 01:15:22] "GET /apple-touch-icon-120x120.png HTTP/1.1" 404 -
192.168.86.48 - - [13/Dec/2020 01:15:22] "GET /apple-touch-icon-precomposed.png HTTP/1.1" 404 -
192.168.86.48 - - [13/Dec/2020 01:15:22] "GET /apple-touch-icon.png HTTP/1.1" 404 -
192.168.86.48 - - [13/Dec/2020 01:15:25] "GET / HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:15:25] "GET /video_feed?g=2020-12-13%2001:15:25.237729 HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:15:31] "GET / HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:15:31] "GET /video_feed?g=2020-12-13%2001:15:31.649402 HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:15:33] "POST / HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:15:33] "GET /video_feed?g=2020-12-13%2001:15:33.789297 HTTP/1.1" 200 -
FEED REQUESTED
FEED DELIVERED
192.168.86.48 - - [13/Dec/2020 01:15:42] "POST / HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:15:42] "GET /video_feed?g=2020-12-13%2001:15:42.018534 HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:16:30] "GET / HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:16:30] "GET /video_feed?g=2020-12-13%2001:16:30.798854 HTTP/1.1" 200 -
192.168.86.48 - - [13/Dec/2020 01:16:30] "GET /apple-touch-icon-120x120-precomposed.png HTTP/1.1" 404 -
192.168.86.48 - - [13/Dec/2020 01:16:30] "GET /apple-touch-icon-120x120.png HTTP/1.1" 404 -
```

```
pijj feeder sudo python3 controller.py
===== FISH FEEDER ======
 team return to_sleep;
 CS 498 IoT FA2020
=======================
starting server...
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
s
------------------------
  FEEDTIME: 12:12
  current time: 01:17
  FEED PERIOD: 2
  current period: 0
  MAX FEED: 10
  times fed: 0
  hungry: True
  moved: False
  is dead: False
------------------------
f
FEED REQUESTED
FEED DELIVERED
s
------------------------
  FEEDTIME: 12:12
  current time: 01:17
  FEED PERIOD: 2
  current period: 0
  MAX FEED: 10
  times fed: 1
  hungry: True
  moved: False
  is dead: False
------------------------
t
------------------------
  FEEDTIME: 12:12
  current time: 01:17
  FEED PERIOD: 2
  current period: 1
  MAX FEED: 10
  times fed: 1
  hungry: True
  moved: False
  is dead: True
------------------------
```

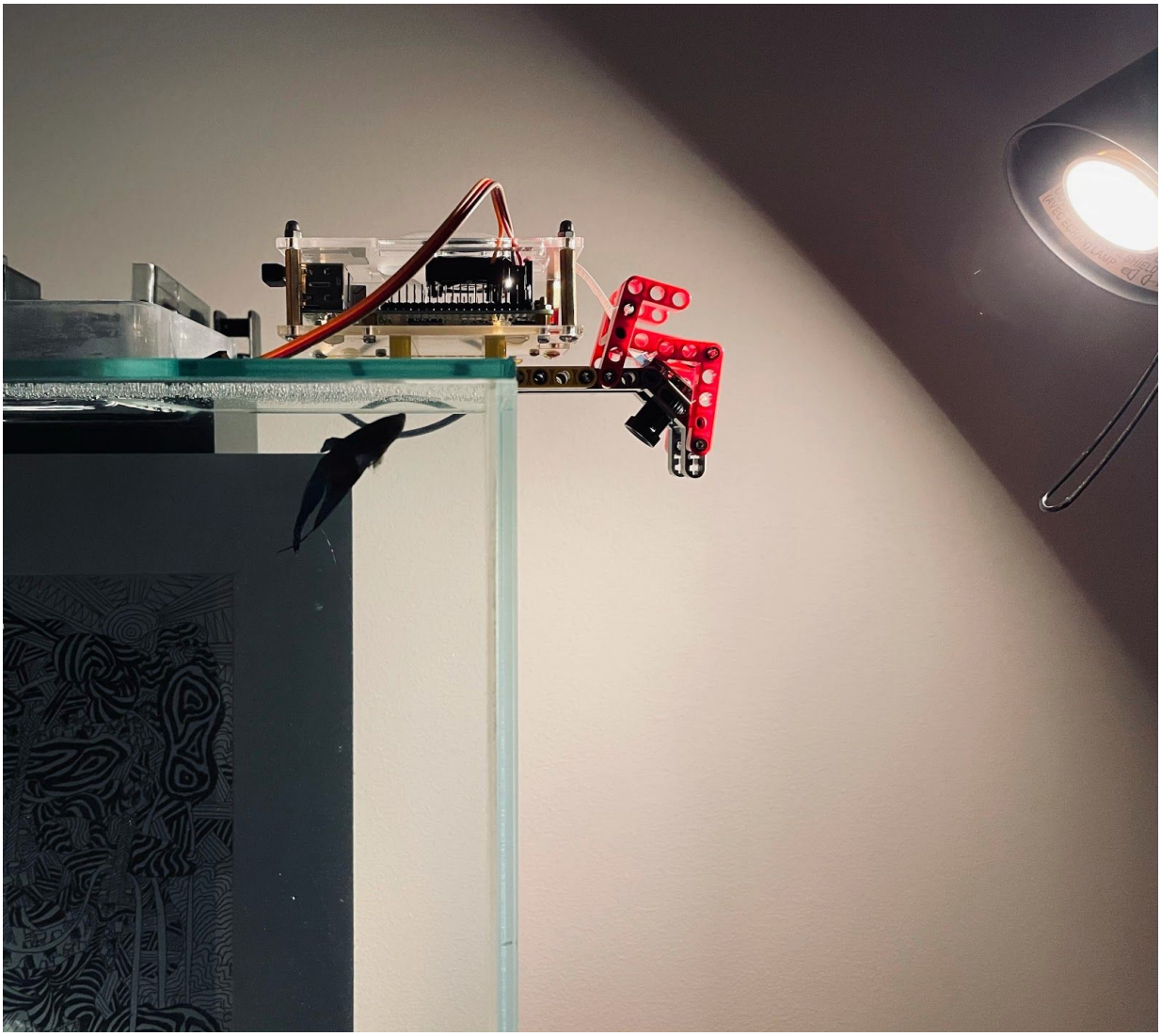
Note: In the first snippet, the “video_feed” attribute is different for each session, which prevents the device’s browser from caching the frame and preventing the video feed from displaying.

Note: In the second snippet, the Fish flag “is_dead” is set to true when a day passes (the command ‘t’ makes a day pass) and the Fish’s flag “move” is false.

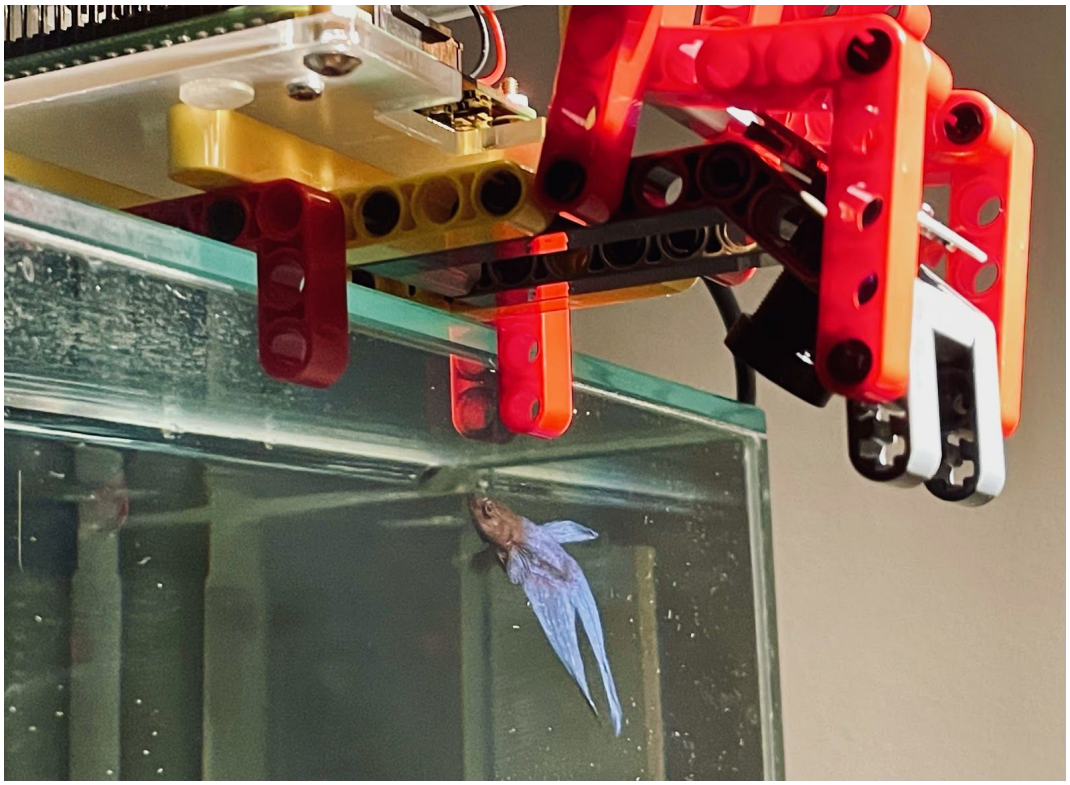
Results

The project successfully completed all proposed tasks and performs the following functions: automatic, scheduled feeding; manual feeding with checks against custom settings; a web application that opens on any home device and shows some information, a feed request, and a live video feed with a motion detection software that highlights the fish and updates the fish's information based on motion.

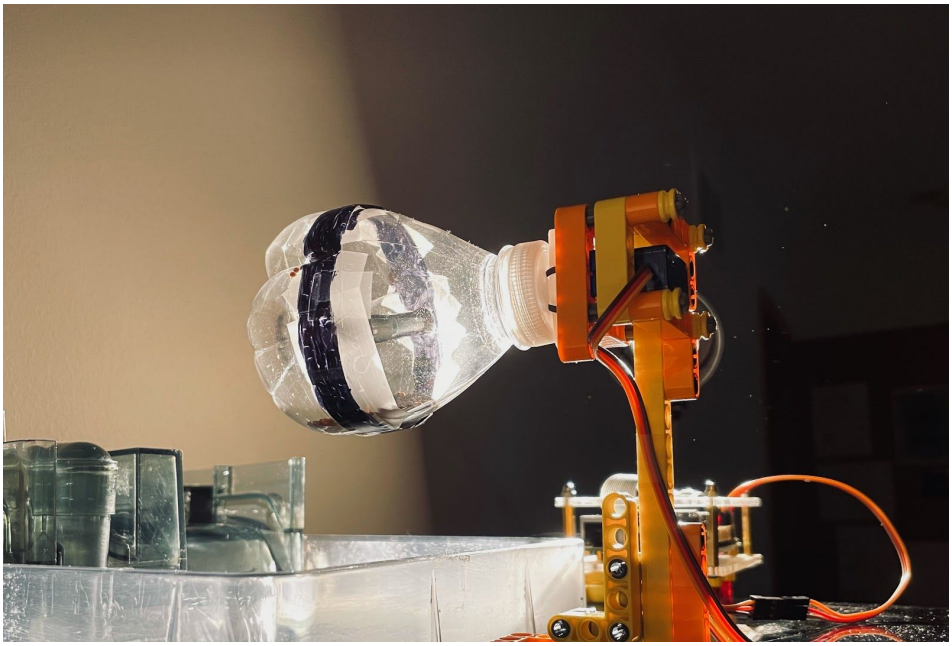
A curious Jello the Betta Fish:



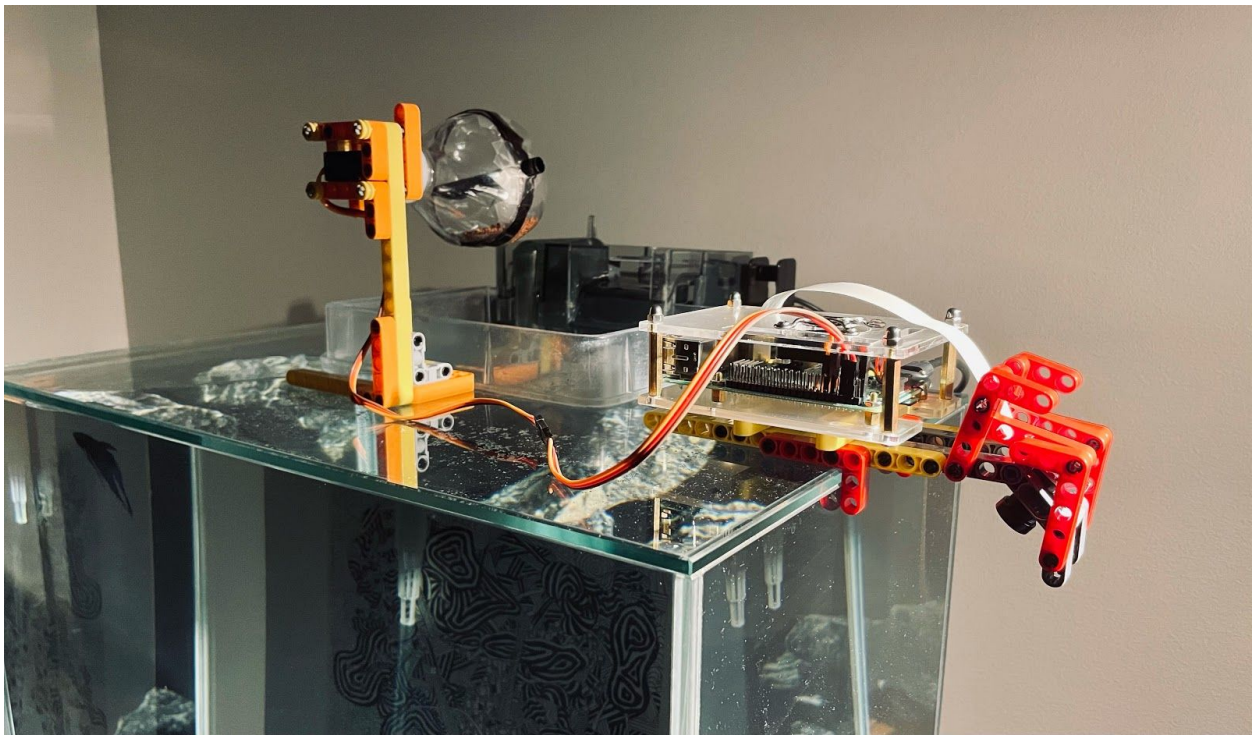
A closer shot of curious Jello the Betta Fish:



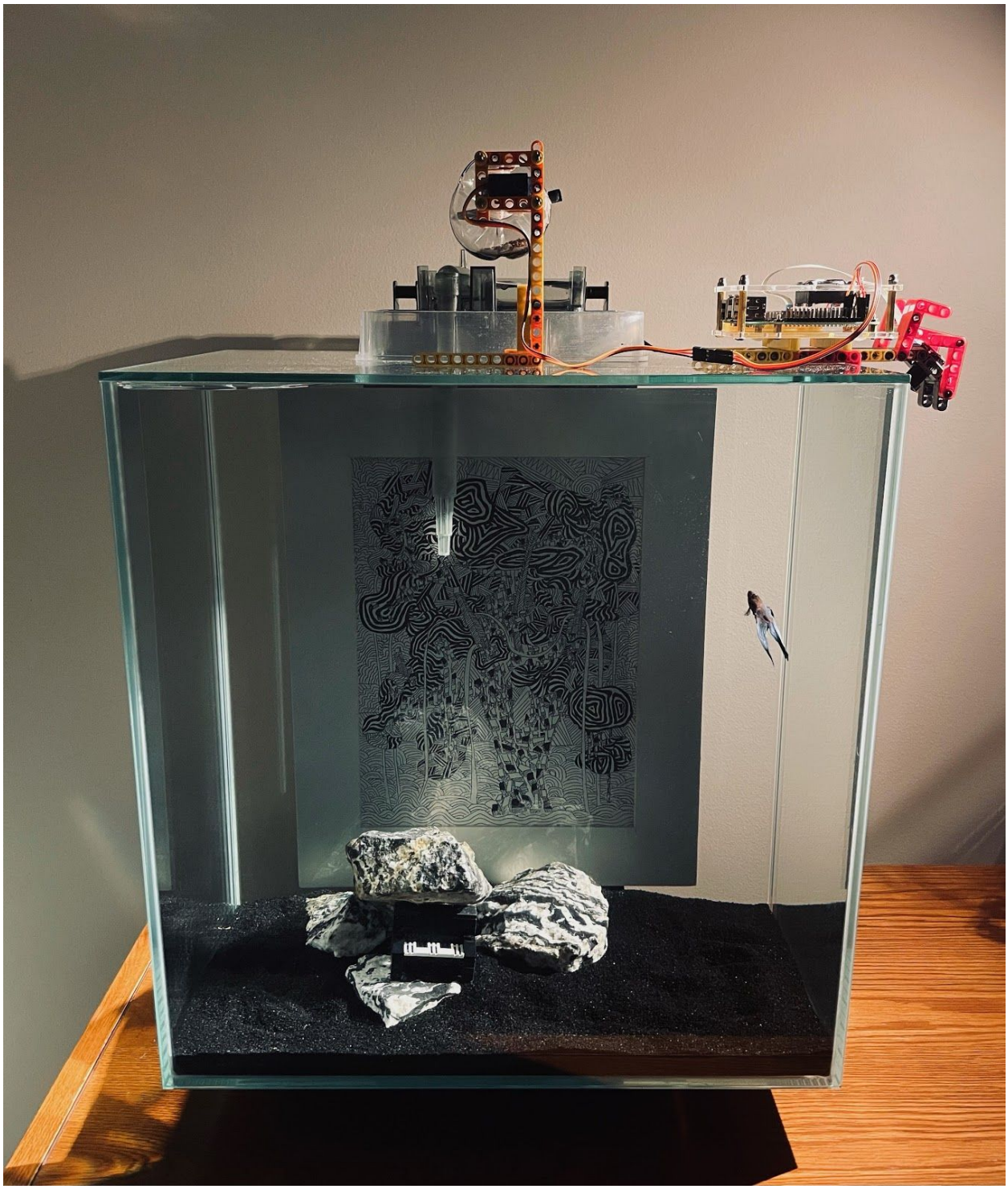
Different angle of the feeder mechanism:



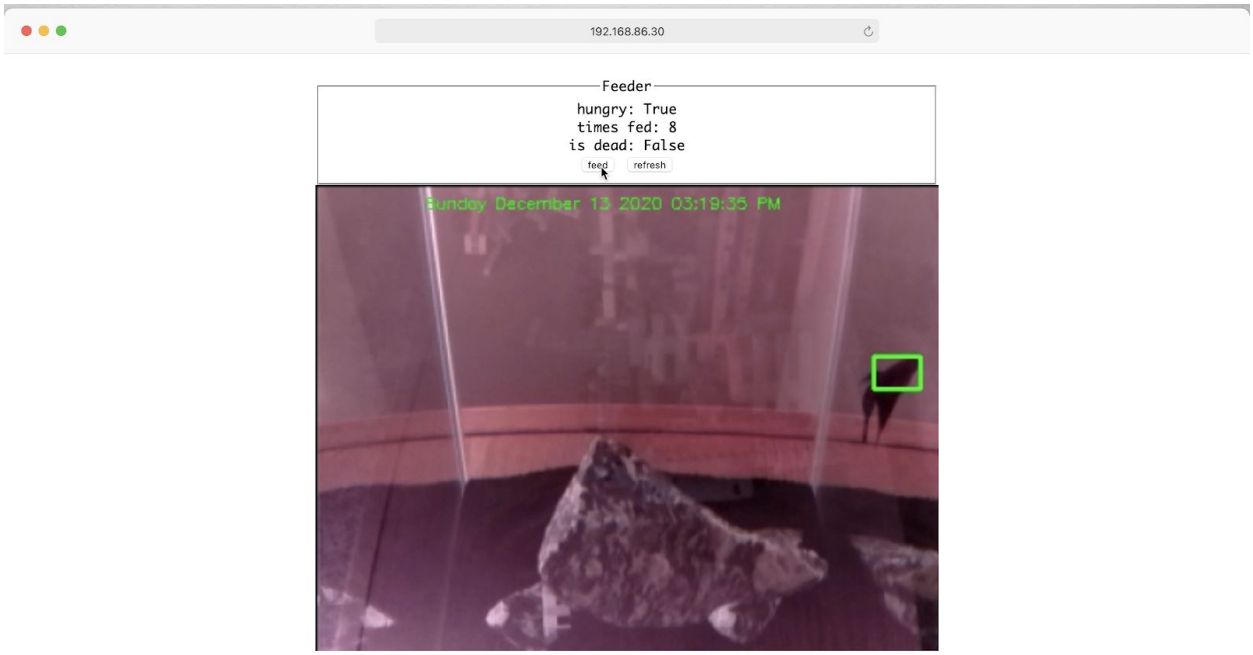
View of feeder hardware in place:

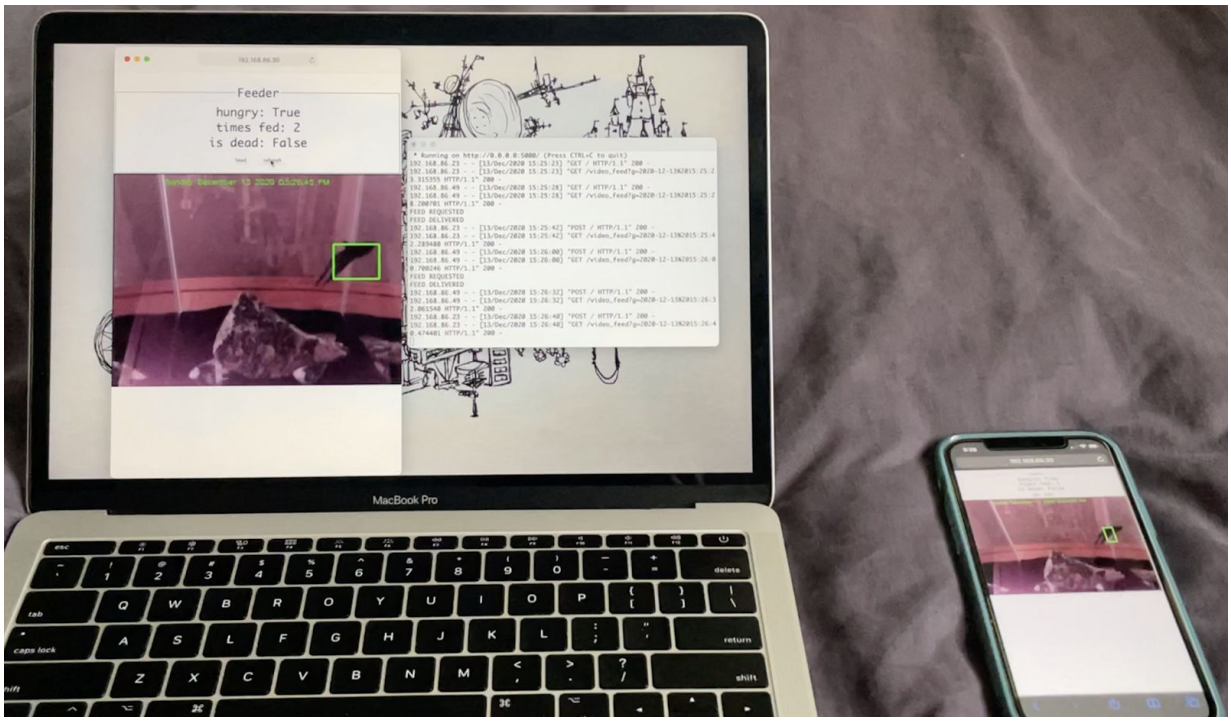


Resulting tank setup:



Web application:





Happy customer:

Team Contributions

We all worked together for this lab, contributing equally and meeting through zoom. The group collaborated via Zoom and Slack. The hardware was developed and tested in JJ's home and the software was developed and tested with all group members. The final product runs locally in JJ's home and machine. Jello the Betta Fish was not harmed during this project.

Video URL

https://mediaspace.illinois.edu/media/t/1_ljdzccb0

Github URL

https://github.com/jjurgello/feeder