iSight

Hand Guidance for Visually Impaired

Seo Ho Kang

Nicholas Milton

Fabio Vulpi

Table of Contents

Introduction

Project Overview

Blind people generally want live normal independent lives and when they are in new environment they will need to know where certain products are. We are inspired to make a computer vision watch to let them know what the object is and guide their hand to the chosen object. We have developed a mobile watch and armband device for under $250 that can identify common objects and guide a users hand to them.

Component Description

Bluno Beetle

The Bluno Beetle is a low cost ATmega328 based microcontroller. Designed for wearable technologies, it is small and lightweight at only 29mm x 33mm and 10 grams. There are 4 digital IO pins which is exactly what we needed to run the 4 vibrating disc motors. Best of all it has a micro USB port for ease of programming and is recognized as an arduino UNO by the Arduino IDE. This allows for port manipulation using the same commands as the arduino UNO.

Figure 1. Bluno Beetle Development board

RaspberryPi 3

The RaspberryPI 3 is a small, lightweight computer running debian linux and supports the python language the Camera CSI port and USB connectivity making this a great option for the computer vision component and main processor for the project. The 5V power requirement allowed us to power it off of a portable phone charger battery pack, making the entire package small enough to strap onto the arm. Much of the online community is using python libraries for computer vision, so support of this language was crucial. The onboard Videocore 4 GPU was not powerful enough to process the image detection code in realtime, only 1 FPS with 3% accuracy. To resolve this we connected the Coral Tensor Processing Unit (TPU) UBS Accelerator to do the majority of the image processing. Because the Raspberry Pi 3 only supports USB 2.0 the full power of the TPU could not be harnessed as it requires USB 3.0.

Figure 2. Raspberry Pi 3

Coral TPU USB Accelerator

Designed by google to work with their tensorflow library, this tensor processing accelerator works with linux based systems like the raspberry pi to enable real time computer vision. Python APIs are available from google for easy implementation of computer vision on the Raspberry pi. Using this TPU we were able to accomplish image processing speeds for object detection up to 10 FPS with 50% accuracy although theoretically the hardware should be able to handle upto 30 FPS.

Figure 3. Google's Coral TPU Accelerator

Raspberry Pi Camera V2

The second generation raspberry pi camera takes 8 Megapixel images and uses the camera serial interface found on the raspberry pi for high speed image processing. This small camera provides higher resolution than we required but the cable limited the placement of the main processor to close proximity.

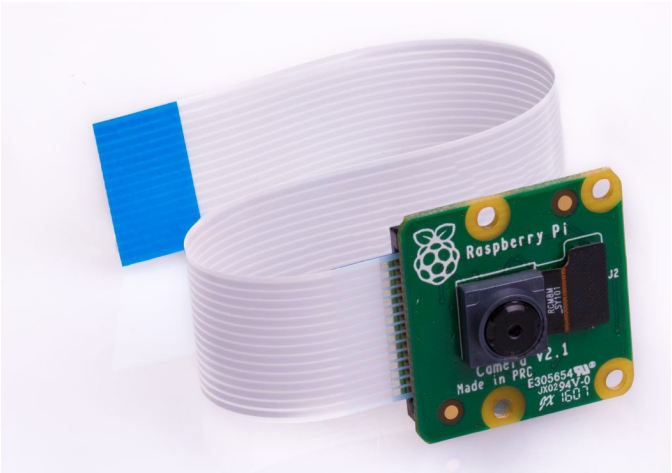


Figure 4. Raspberry Pi Camera Module V2

Vibrating Mini Disc Motors

To give the user haptic feedback we attached small 10mm vibrating disc motors to rings on the pinky and thumb fingers as well as embedded into the top and bottom of the *watch*. These 3.3V motors only require a 120mA max current, but that still required us to build circuitry to isolate the current source from the microcontroller.

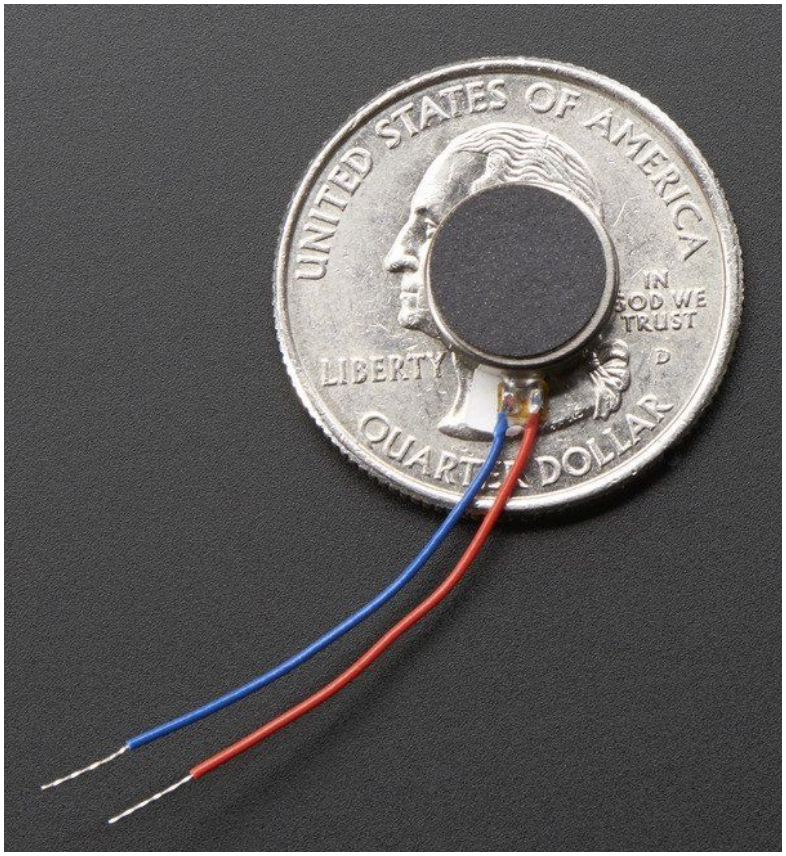


Figure 5. Vibrating Mini Disc Motor over a quarter to show scale

Custom PCB

We designed and fabricated a small printed circuit board top mount the bluno beetle, power it and isolate the power for the vibrating disc motors. A 3v voltage regulator knocks the voltage down to prevent damaging the motors. As seen in Figure 6, there are four 2n2222 NPN transistor circuits controlled by the digital IO pins of the bluno beetle switching the 3v circuit to the vibrating disc motors. These transistors are capable of supplying up to 800mA of current, giving us a healthy factor of safety. We added 1N4007 diodes to the circuit to prevent to the transistors due to a back EMF from from the motors. We designed the board in EAGLE cad and made the prototype on the Othermill Pro CNC Mill.

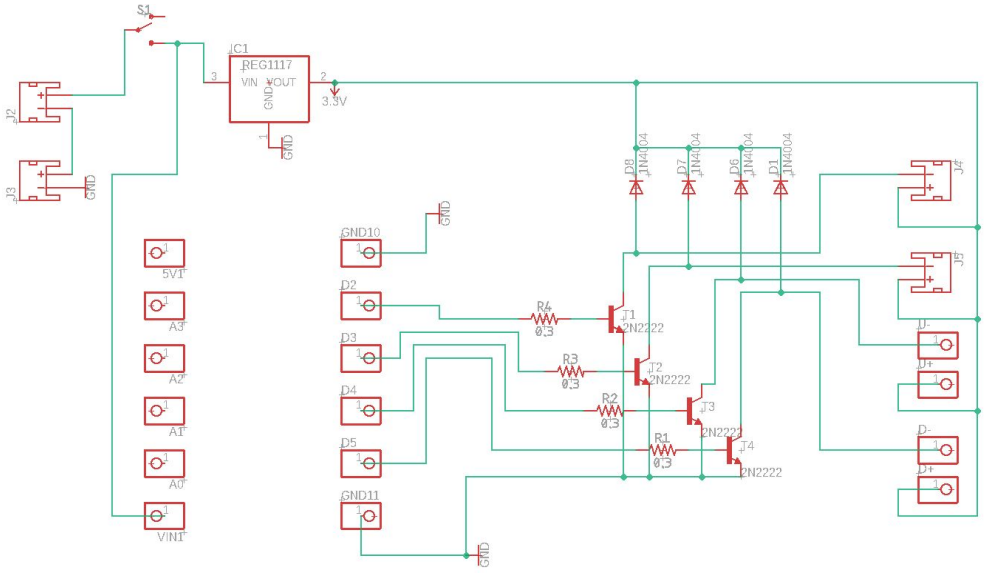


Figure 6. EAGLE Schematic of Printed Circuit Board

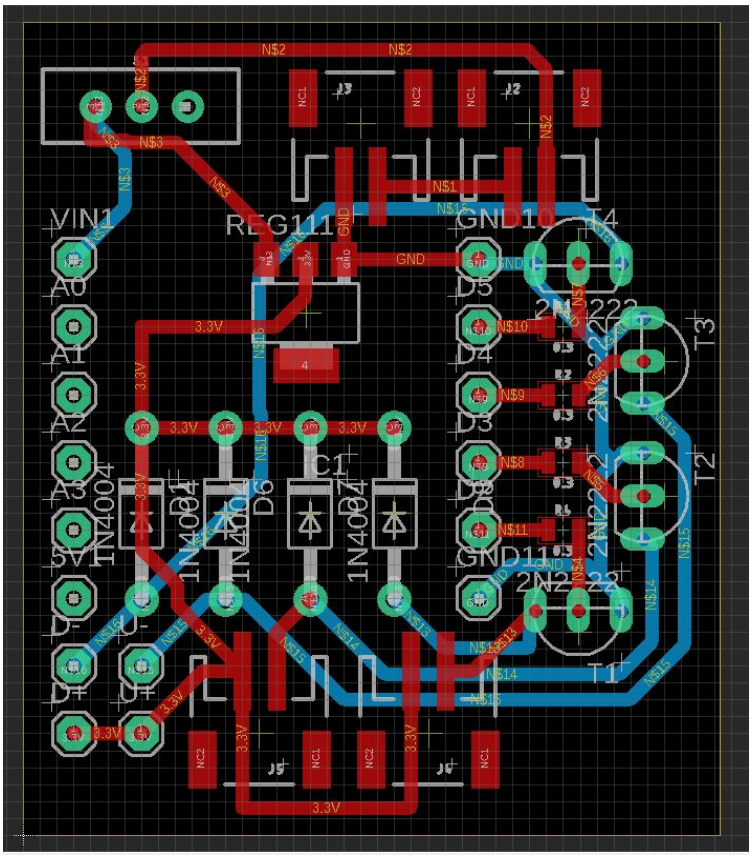


Figure 7. EAGLE PCB Layout

Batteries

Anker Lithium ion 5v 6700mAh portable charger was used to power the raspberry pi via a micro usb cable. Two 150mAh 3.7v lithium ion battery packs are attached to the PCB to run the Bluno beetle at 5V and the vibrating disc motors regulated down to 3V. These batteries would give an estimated 6 hrs of image processing and 3 hrs of active guiding.

Cases and Mounting

Custom enclosures were designed to protect the more delicate electronics, provide a more consistent mounting point for the Raspberry Pi Camera and make the product more fashionable. All pieces were designed in Autodesk Inventor and printed in black ABS on the MOJO 3D printer.

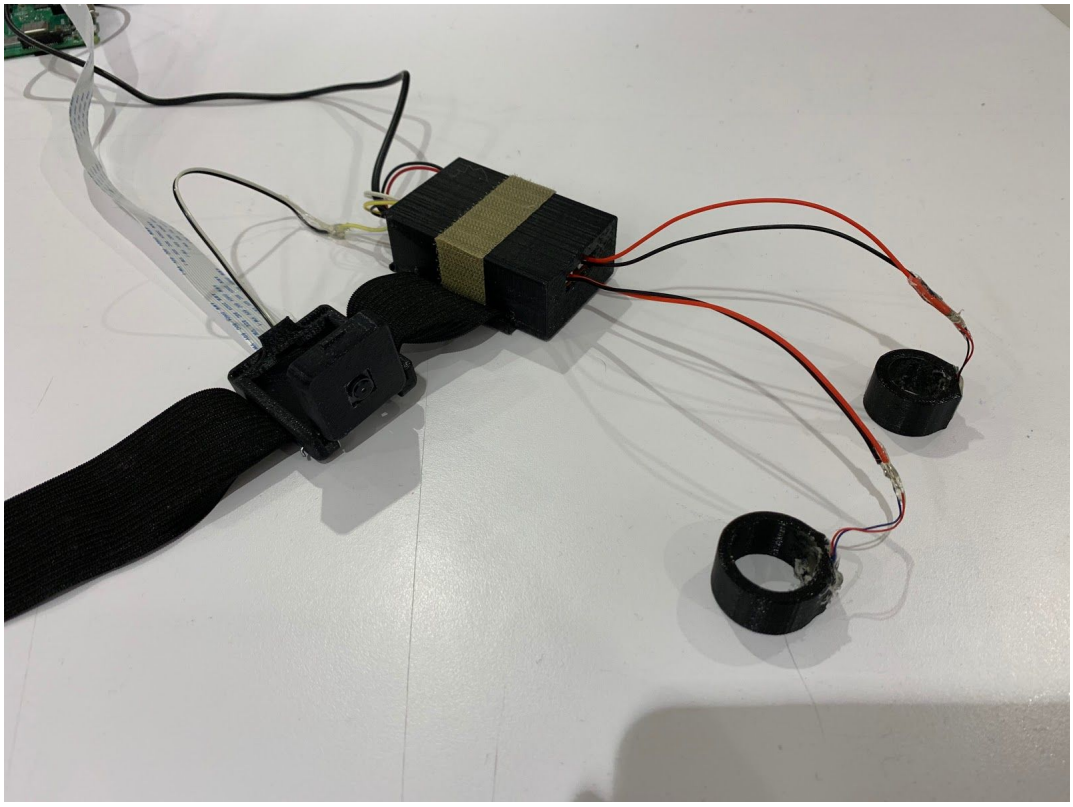


Figure 8. Printed enclosures and rings to hold vibrating disc motors, camera and microcontroller

Software Description

Summary of Architecture

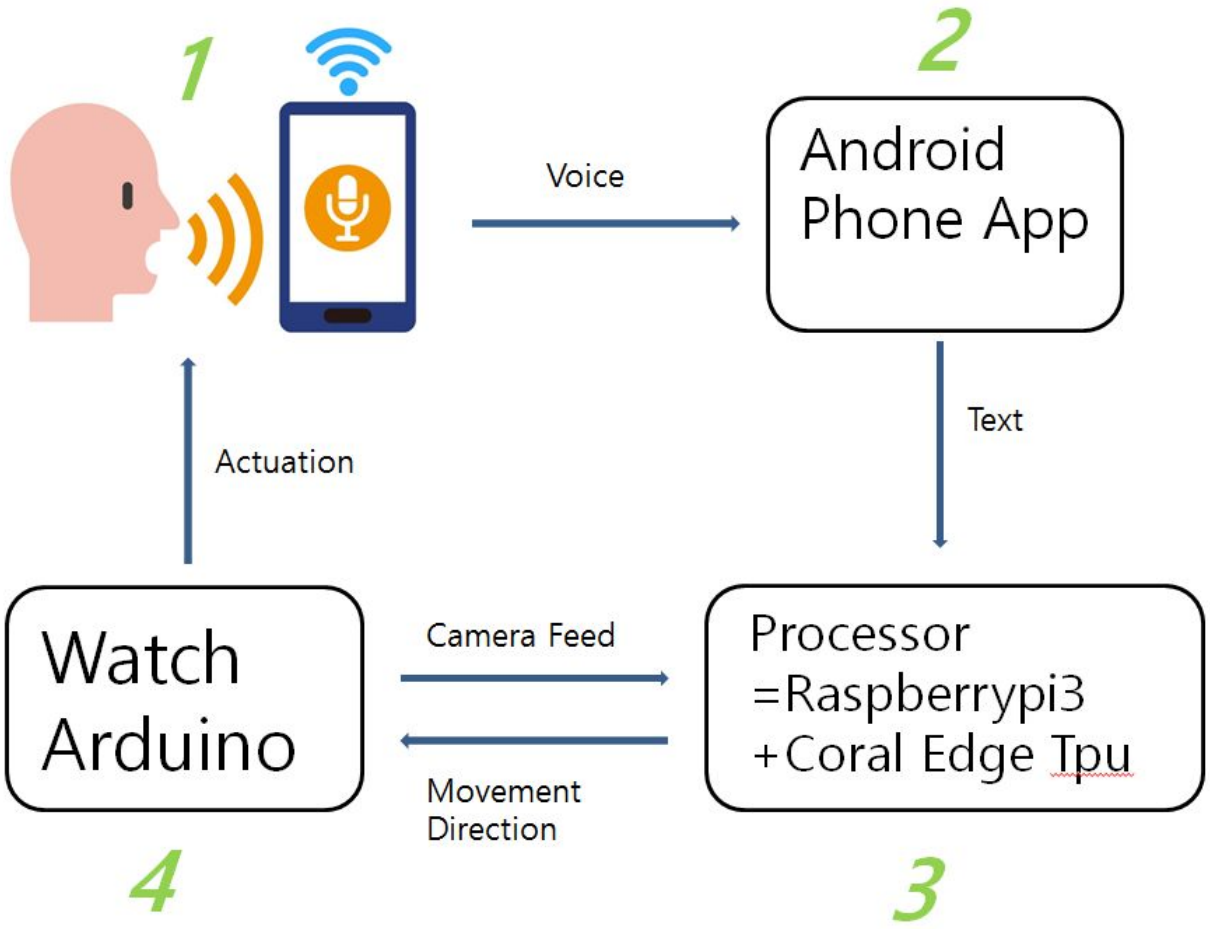


Figure 12. High Level System Overview

Android Phone App : On-Offline voice recognition

1.PocketSphinx[0] - Offline Voice Recognition

There were 2 pros of offline voice recognition and many cons of offline voice recognition. It's good because 1) 4G LTE module is expansive to put on raspberry pi3(Cheapest module $130) 2) It's difficult to get wifi connection for the users when place changes with raspberry pi3. Plus,

our object detection labels are fixed, offline object detection model works perfect logically and we can only update at each user's home. There were oo categories to implement this.

1) Set external usb sound card as default.[1]
2) Change asound.conf file to solve aplay issue.[2]
3) Create first, pronunciation dictionary, second, language model
4) Run voice recognition
5) Remove log file on command by "logfn /dev/null" command
6) Make raspberry pi3 work asynchronous subprocess"from subprocess import Popen" to listen what is printed on command line to pass the voice → text value to object detection model.

However, there were 2 big problems of offline voice recognition models.
First, it was very slow(One circulation took ~10 seconds from voice → text → object detection → actuation)
Second, the accuracy of the voice recognition model was very low.(Succeeded one circulation by telling same command less than 20% since voice → text part was inaccurate)

2.Google Speech to Text API - On-Offline Voice Recognition

We opt to make Android app to use google speech to text api which was accurate and fast(~3 seconds for one circulation when offline) and works both online and offline.

Processor : Raspberry Pi3 + Coral Edge Tpu

Processor is connected to same wifi with android phone app.

1) Test_speech.py : This script is server side of speech recognition. First, load 90 labels from coco_labels.txt and gets corpus when a blind person speaks through app. This was done by serial communication. If the sentence contains certain data that is in our labels it only returns the label. As an example, if the user says "Let's grab the bottle", it only looks for "bottle" which is the lable in 90 labels in "coco_labels.txt"
2) serialPass.py : Gets actuation inputs as "u" which is up, "d" which is down, "f" which is forward, "l" which is left, "r" which is right. And then encode this haptic direction command because we are using python3. After encoding we pass this to arduino(bluno).
3) engine.py : This is library which is offered from coral edge tpu. According to this instruction, we flatten image and follow rules to use tensorflow lite. Coral USB TPU is very small compared to other GPU(TX2, Jetson Nano, Coral Dev Board etc.), so it uses the lightest model of existing object detection model currently.
4) Object_detection_ver5.py :
    a) Import mobilenet_ssd_v2_coco_quant_postprocess_edgetpu.flite pretrained object detection model and coco_label.txt(90 labels)
    b) Inputs 300, 300 picamera image as input at around 10FPS
    c) Object detection model gets input image and outputs true label, (x, y) coordinates of bounding box of objects, scores of objects(probability).

d) When the “voice passed label” matches with the true label we only gets bounding box of the object
e) Using this bounding box(x,y) coordinates we calculate center of the object and use algorithm(explained below) to decide which direction of actuation to give(5 actions)
f) Uses serial communication(ttyACM0) and 115200 baudrate to pass direction command to arduino.
g) Since, latency is the bottom line of our project we opt for the fastest way which is usb connection to the actuator and attached directly the picamera value to the raspberry pi3.
h) Even though coral edge tpu USB uses USB3.0, it gets slower since raspberry pi3 uses USB2.0
i) The pretrained model of object detection model was around 50%.

Arduino Code

The Setup Loop

```
void setup() {
  DDRD |= B111100;
  Serial.begin(115200);
}
```

The Setup initializes the 4 bluno beetle digital pins available as GPIO, pin from 2 to 5, as outputs, all together using port manipulation. Pins 0 and 1 are not available as GPIO because they are RX and TX. Second line of the setup initializes the baud rate of the serial communication.

The Main Loop

```
while (Serial.available()==0){}
cmd = Serial.read();
switch (cmd) {
  case 'u':
  PORTD |= B000100;
  delay(dt);
  PORTD &= ~(B000100);
  break;
  case 'd':
```

The main loop consists of a while empty loop that makes the microcontroller wait for the command on the serial port, then as soon as the command is written on the serial port, the message gets received and enters a switch statement. Each of the cases of the switch turns high a different pin of the bluno beetle that will open the circuit of a motor and make it vibrate for a precise amount of time, “dt” initialized in the beginning. We used port manipulation because the “forward” feedback needs two motors to vibrate at the same time with a particular frequency.

Python Code

The Main Loop

1-1) Run script with command below.
[Command]
python3 demo/object_detection_ver5.py \
  --model test_data/mobilenet_ssd_v2_coco_quant_postprocess_edgetpu.tflite \
  --label test_data/coco_labels.txt

2-1) FPS optimization-used parser to organize script
[Snap of object_detection_ver5.py]

```
  parser = argparse.ArgumentParser()
  parser.add_argument(
      '--model', help='Path of the detection model.', required=True)
  parser.add_argument(
      '--label', help='Path of the labels file.')
  parser.add_argument(
```

```
    '--input', help='File path of the input image.')
  parser.add_argument(
    '--output', help='File path of the output image.')
  args = parser.parse_args()
```

2-2) FPS optimization-use array input

We chose to input image array to “mobilenet_ssd_v2” directly after flatten the image. We previously input image but after changing it to input array, FPS increased by 5~10. Flatten step was done because Google TF lite api asked us to put flatten image.

Plus, in order to get fast FPS you should follow the way we implemented with pi-camera.

[Snap of object_detection_ver5.py]

```
  engine = DetectionEngine(args.model)
  labels = ReadLabelFile(args.label) if args.label else None

  # Open image.
  #img = Image.open(args.input)
  #draw = ImageDraw.Draw(img)
  # Setup Camera Constants
  IM_WIDTH = 300
  IM_HEIGHT = 300
  camera = PiCamera()
  camera.resolution = (IM_WIDTH, IM_HEIGHT)
  camera.rotation = 180
  camera.framerate = 60
  raw_capture = PiRGBArray(camera, size=(IM_WIDTH, IM_HEIGHT))
  raw_capture.truncate(0)
  start = time.time()
  counter = 0
  for frame1 in camera.capture_continuous(raw_capture, format='bgr', use_video_$
    frame = frame1.array.flatten()
    #time.sleep(0.2)
    #print(type(frame1))
    #frame.setflags(write=1)
    #frame_expanded = np.expand_dims(frame, axis=0)
    #print(frame)
    #frame_expanded = np.array(frame_expanded, np.uint8)
    #(boxes, scores, classes, num) = sess.run([detection_boxes, detection_score$
    #_score = np.squeeze(scores)[0]
    # Run inference.
    # print(img.shape)
    #print("HI")
    #frame1.seek(0)
    #im = Image.fromarray(frame)
```

```
    #ans = engine.DetectWithImage(im, threshold=0.05, keep_aspect_ratio=True,re$
    #print(frame.shape)
    ans = engine.DetectWithInputTensor(frame)
    raw_capture.truncate(0)
```

2-3) FPS optimization-remove display and opencv

[Snap of object_detection_ver5.py]

```
  '''
  # Display result.
  if ans:
    for obj in ans:
      print ('-----------------------------------------')
      if labels:
        print(labels[obj.label_id])
      print ('score = ', obj.score)
      box = obj.bounding_box.flatten().tolist()
      print ('box = ', box)
      # Draw a rectangle.
      draw.rectangle(box, outline='red')
    img.save(output_name)
    if platform.machine() == 'x86_64':
      # For gLinux, simply show the image.
      img.show()
    elif platform.machine() == 'armv7l':
      # For Raspberry Pi, you need to install 'feh' to display image.
      subprocess.Popen(['feh', output_name])
    else:
      print ('Please check ', output_name)
  else:
    print ('No object detected!')
  '''
```

3-1) Communication (Android app <-> Raspberry Pi3)

[serialPass.py, test_speech.py]

We use socket serial communication between app and rpi3. App converts voice to text and send it to raspberry pi3.

```python
from socket import *
from time import ctime
import time
import RPi.GPIO as GPIO
import os

def setup():

    with open("coco_labels.txt") as mf:
        ta = mf.readlines()

    labels = [l[:-1] for l in ta]

    HOST = ''
    PORT = 21567
    BUFSIZE = 1024
    ADDR = (HOST,PORT)
    tcpSerSock = socket(AF_INET, SOCK_STREAM)
    tcpSerSock.bind(ADDR)
    tcpSerSock.listen(5)

#    return tcpSerSock

#def get_label(tcpSerSock):
    while True:
        print('Receiving data')
        tcpCliSock,addr = tcpSerSock.accept()
        print('Received data : \n', addr)
        try:
            while True:
                print("Now start receiving data")
                data = ''
                data = tcpCliSock.recv(BUFSIZE)
                print(type(data))
                data = str(data)
                print(type(data))
                if not data:
                    break
                for label in labels:
                    if label in data:
                        return label
                    elif "describe" in data:
                        os.system("python3 demo/object_detection2.py --model test_data/mobilenet_ssd_v2_coco_quant_postprocess_edgetpu.tflite --label test_data/coco_labels.txt")
                time.sleep(0.1)
        except:
            print("error")
```

3-2) Communication (Arduino <-> Raspberry pi3)
We use ttyACM0 serial communication.

```
ser = serial.Serial("/dev/ttyACM0", 115200)
ser.baudrate = 115200
```

4) SSD Object detection model
Input is [300, 300] image array and we use label, and box (x,y) coordinate output. We use pretrained model.

```
def DetectWithInputTensor(self, input_tensor, threshold=0.1, top_k=3):
  """Detects objects with raw input.

  This interface allows user to process image outside the engine for
  efficiency concern.

  Args:
    input_tensor: numpy.array represents the input tensor.
    threshold: float, threshold to filter results. Default value = 0.1.
    top_k: keep top k candidates if there are many candidates with score
      exceeds given threshold. By default we keep top 3.

  Returns:
    List of DetectionCandidate.

  Raises:
    ValueError: when input param is invalid.
  """
  if top_k <= 0:
    raise ValueError('top_k must be positive!')
  _, raw_result = self.RunInference(input_tensor)
  result = []
  num_candidates = raw_result[self._tensor_start_index[3]]
  for i in range(int(round(num_candidates))):
    score = raw_result[self._tensor_start_index[2] + i]
    if score > threshold:
      label_id = int(round(raw_result[self._tensor_start_index[1] + i]))
      y1 = max(0.0, raw_result[self._tensor_start_index[0] + 4 * i])
      x1 = max(0.0, raw_result[self._tensor_start_index[0] + 4 * i + 1])
      y2 = min(1.0, raw_result[self._tensor_start_index[0] + 4 * i + 2])
      x2 = min(1.0, raw_result[self._tensor_start_index[0] + 4 * i + 3])
      result.append(DetectionCandidate(label_id, score, x1, y1, x2, y2))
  result.sort(key=lambda x: -x.score)
  return result[:top_k]
```

```
ans = engine.DetectWithInputTensor(frame)
raw_capture.truncate(0)
#print('captured %s' % filename)
frame1.truncate(0)
#rawCapture.seek(0)
counter += 1

ser = serial.Serial("/dev/ttyACM0", 115200)
ser.baudrate = 115200

if counter == 10000000:
  break
if ans:
  for obj in ans:
    if labels[obj.label_id] == lab:
      box = obj.bounding_box.flatten()
      box *= 300
      center_x = ((box[0] + box[2]) / 2)
      center_y = ((box[1] + box[3]) / 2)
      if center_x >=76 and center_x <=236 and center_y >=76 and cente
```

Cost Analysis

Bill of Materials

Description	Part Number	Qty	Cost Each	TOTAL COST
Bluno Beetle		1	$13.00	$13.00
Raspberry Pi 3		1	$35.00	$35.00
Coral TPU USB Accelerator		1	$74.99	$74.99
Raspberry Pi Camera V2		1	$29.95	$29.95
Vibrating mini disc motor	TS-711	4	$2.00	$8.00
Anker Portable Charger	A1211	1	$24.00	$24.00
Li ion 3.7v 100mA battery packs	TS-1762	2	$5.99	$11.98

3D Printed Parts		1	$9.64	$9.64
Running Arm Band		1	$8.49	$8.49
Resistors		4	$0.10	$0.40
Voltage Regulator		1	$1.10	$1.10
Diode	1N4007	4	$0.20	$0.80
NPN transistor	2N2222	4	$0.42	$1.68
PCB ProtoBoard		1	$1.00	$1.00
JST connectors	JST	6	$1.99	$11.94
Micro USB Cable		1	1.99	1.99
			TOTAL COST of PROTOTYPE	$233.96

Conclusion

Product Refinement

Our initial testing began with just the four haptic feedback vibrating discs in a glove with manual input control from an end user in the serial monitor. We started with nine regions for direction which would turn on one to four vibrating motors, four being forward. The regions are depicted in Figure 9a**.** The motors were controlled simultaneously using port manipulation. These tests were unsuccessful at guiding the hand to a mouse because the diagonal instructions vibrating two motors overwhelmed the nerves in the hand and which confused these instruction with all of the motors being on (the forward instruction). At first we tried to move the vibrating discs further apart, keeping the sides on the edge of the pinky and index finger knuckles and moving the up down vibrators to the wrist. This was an improvement with the actuators but we were not able to successfully guide the hand until we adopted a simpler set of 5 haptic instructions as shown in Figure 9b.

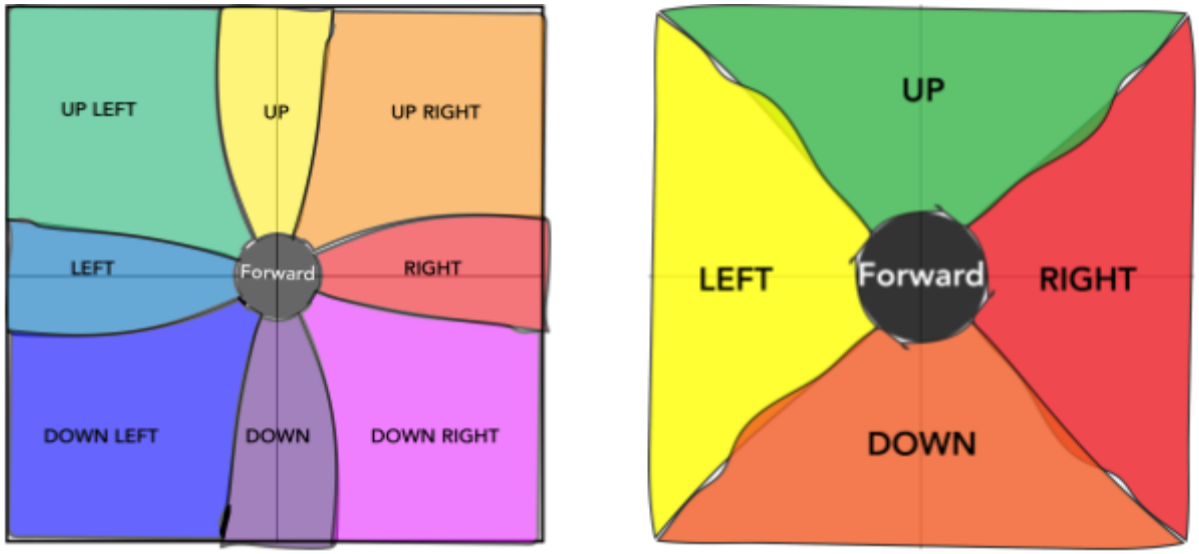


Figure 9. (a) *left*- the original 9 regions of actuation (b) *right*- the final 5 regions of actuation

These are merely the directions the user is instructed to move, not to be confused with the regions of detection which will be discussed in the Algorithm development section.

The glove would eventually be replaced upon receiving feedback from people with visual disabilities who said it had to be fashionable or less apparent, in more than one instance staying it should be “like a watch.” We revised the system to be a double sided watch with the bluno beetle microcontroller and PCB on top and the Raspberry Pi camera mounted on the bottom so that it can fold flat into the watch when not in use. Vibrating motors were embedded into these structures for vertical feedback. Two different sized rings with vibrating motors glued into them

are connected to the watch and to be worn on the pink and index finger/thumb to give horizontal feedback. This not only made it more visually appealing, but gave a more isolated haptic stimulation.

Algorithm Refinement

In the first version of the algorithm we divided the image in five regions: forward, up, down, left, right. The object detection algorithm can recognize the objects and compute the coordinates of the four edges of a bounding box around them. With that information we computed the coordinate of the center of the bounding box to decide what type of haptic feedback give to the user based on the region in which the center falls. This process is done with a nested if statement whose first check is the forward region. We tried different sizes of the forward region and as a result we got an unstable haptic feedback, because the scenario is changing too fast and the separation between the regions is ambiguous. The bigger the forward region is more likely it is that user passes over the object before having received the feedback of a different region, on the other hand, the smaller the forward region is less probability you have to hit it and the user will keep moving his hand left, right, up and down without going forward and reaching the object. Most of the trials done with this algorithm failed and required several minutes for the user to get even close to the object.

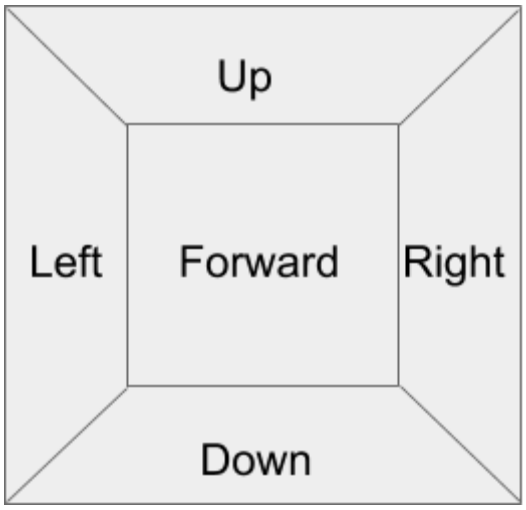


Figure 23. Algorithm version 1

To simplify the code, since the first statement of the nested if is forward, we modified the definition of all the other regions. If the center of the bounding box doesn't fall inside the forward region the decision of which one of the other four feedback to give is took by redefining the "search zones" (up, down, left and right) as four equal triangles. The improvement from the previous version was little, and this version still suffered from all the same problems as this really just made the code more readable.

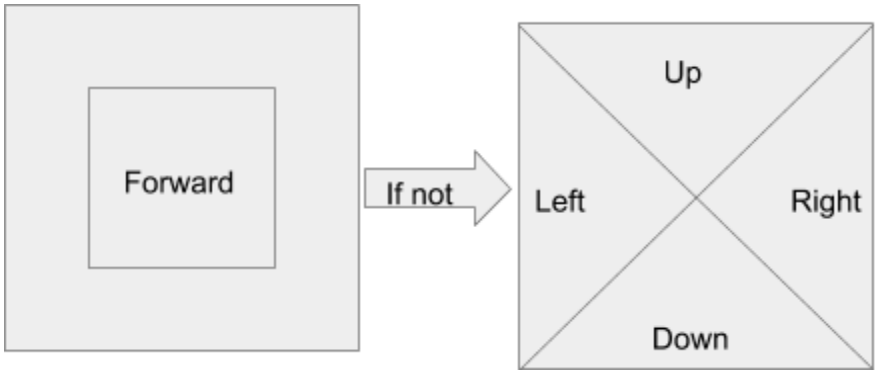


Figure 24. Algorithm version 2

To better balance the actuation feedback we improved with version 3 of the algorithm. In this version we still checked first if the center of the bounding box is in the forward zone, then, if it isn't we divide the image in two large and clearly distinguished zones with a horizontal line right in the middle of it, to decide if the user should move up or down. After this decision is the sent to the user, raspberry pi is paused for the necessary time to vibrate the motors (100 milliseconds). The same image is then divided vertically into two equally large and well distinguished regions, with a vertical line at its middle to decide if the user should move left or right to try to get the center of the box in the forward region for the next image to be processed after this second haptic feedback is sent. The main difference with this version is that it gives two feedbacks for every image, instead of one, and tries to get the object in the forward region every step with a combination of left/right and up/down signals. Using this strategy we got better results, and successfully grabbed the object in 28 seconds, but the feedback was very confusing and hard to understand. The problem with this strategy was that after an "up" or "down" feedback there was always a "left" or "right" feedback and the short signal duration of just 100 milliseconds made pinpointing the vibration signal very confusing.

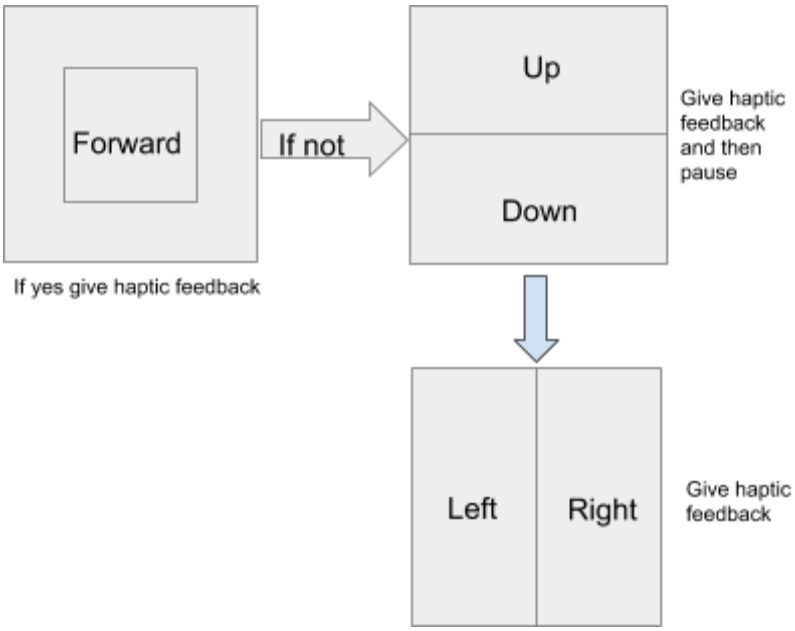


Figure 25. Algorithm version 3

As final version of the algorithm we developed a system of feedback that is both clear, continuous and easy to follow. As in the previous version the first thing that is done is to look if the object is in the forward zone, then, if it is not, we define check if it is either in the up or in the down zone, that this time are defined as the zones of the image over and under the forward zone, and not with a simple horizontal line splitting the image in half. If the object is found to be in one of this two zones than actuation feedback is given and another image is taken, if not then the image is slip in two parts with a vertical line to decide the proper feedback between left and right. In this way we get stable haptic feedbacks, that are constant if the user doesn't move, not confusing, and clearly distinguishable. This algorithm takes advantage of the previous version, by defining the "search zones" in a clear and simple way but still giving a single clear feedback for every image. During tests, with this algorithm, the user was capable of successfully grab the desired object in 22 seconds.

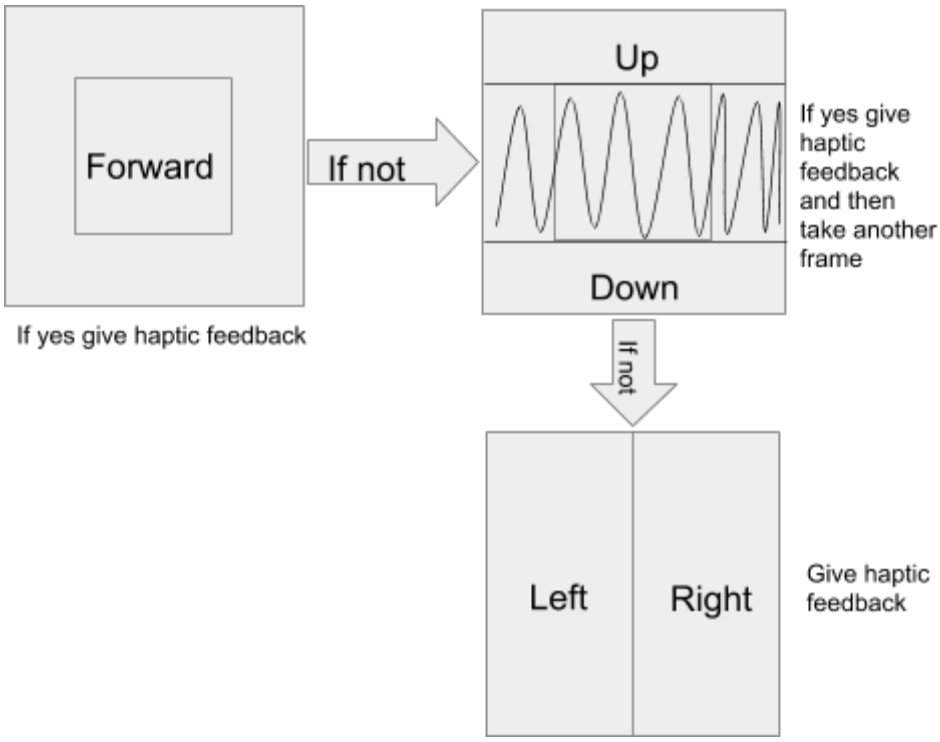


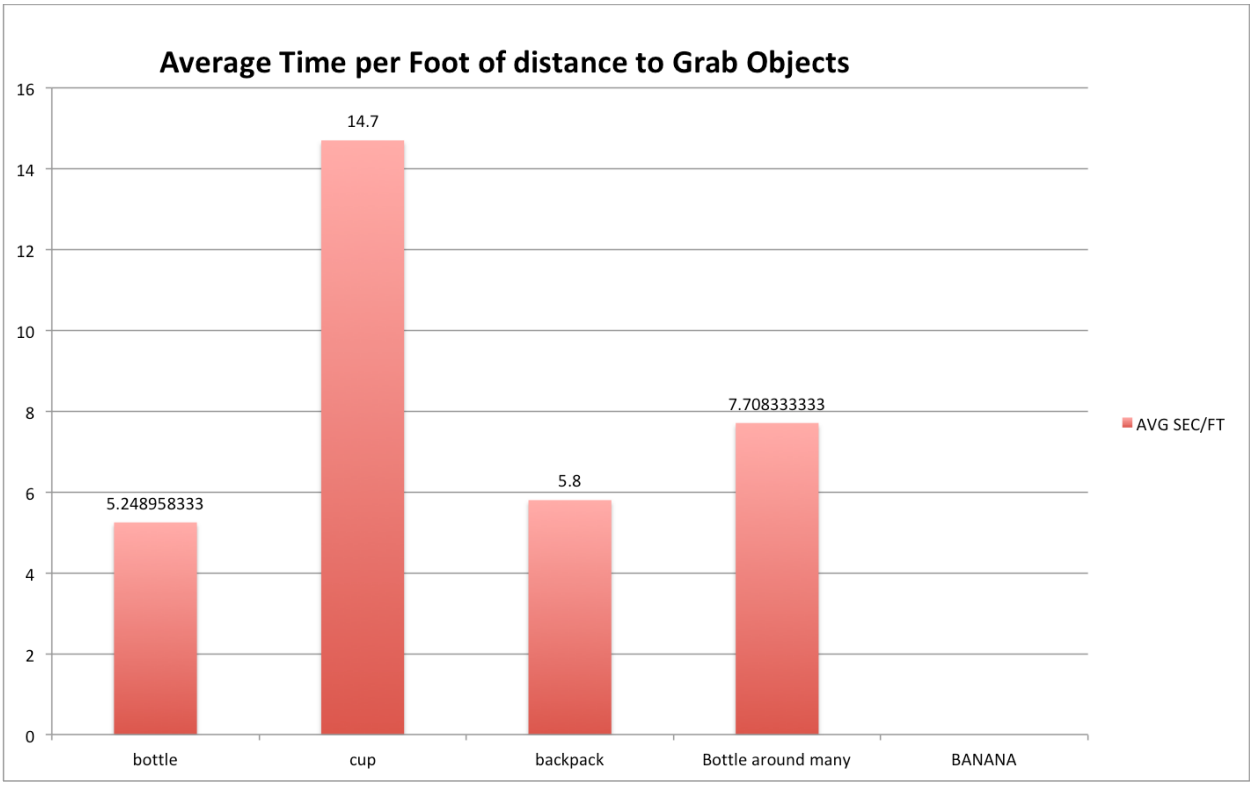
Figure 26. Algorithm version 4

Results

All testing was completed blindfolded in indoor lighting scenarios. The participant was blindfolded and then the object placed. The distance was measured from the user to the object to calculate the time it took to grab the object per foot it was away. The testing through the algorithm version showed steady improvement although the 4th version had the quickest time, the 5th version felt more natural to the user and produced consistent results.

Algorithm	AVERAGE Seconds to grab per foot from object
VERSION 1	7
VERSION 2	9
VERSION 3	5.333333333
VERSION 4	5.697916667

After the algorithm was refined we tested different objects to see the time they would take to grab all of which were successful except the banana, which we were not able to detect. The Bottle was the easiest to detect and move to while the orange mug we used as a cup was very difficult to detect and lost numerous times during tracking. Larger objects like the backpack were never lost once the object was detected.



There is a noticeable time increase when other objects are placed around the desired object to grab as depicted in the “bottle around many” as compared to the “bottle” alone in the bar chart above.

Future Work

Aside from miniaturizing the entire system to fit into a smaller watch size the most important future development is to create a feedback signal when the user is close to the item. Also Using 5G communication we could use a cloud computing platform to use more advanced object detection models.

References

[0]https://wolfpaulus.com/embedded/raspberrypi2-sr/

[1]https://raspberrypi.stackexchange.com/questions/80072/how-can-i-use-an-external-usb-sound-card-and-set-it-as-default

[2]https://www.raspberrypi.org/forums/viewtopic.php?t=30024