DIY 3D Scanner

Section 2, Mini-Project 2: Hong Zhang and Mateo Otero-Diaz

Introduction

We built a low-cost, DIY 3D scanner that measures the shape of simple objects using one infrared distance sensor mounted on a two-axis pan/tilt rig driven by servos. An Arduino controls the servos and streams distance measurements to a laptop over USB. By sweeping the sensor left–right and up–down, the system samples the surface in a grid and reconstructs the object.

Before scanning, we calibrated the sensor so its electrical reading maps to distance in millimeters. We collected measurements to a flat target at known positions and fit a simple inverse model, then kept the scanner operating in the range where that model is reliable. With calibration in place, the scanner outputs two kinds of visualizations. The first visualization was a single "line scan" using only one servo, which shows a 2D profile (angle vs. distance) and the second was a full raster with both servos, rendered as a depth heatmap and a 3D point cloud. This produces a recognizable silhouette of the cardboard shape we scanned. All mounts were designed for screws so the rig can be assembled and disassembled cleanly.

Testing

We wired the IR distance sensor to the Arduino, powered the servos using our laptops, and streamed readings over Serial at 115200 baud. With the sensor pointed at a flat cardboard target, moving the target closer and farther caused the reading to change consistently. This confirmed the sensor and setup were working.

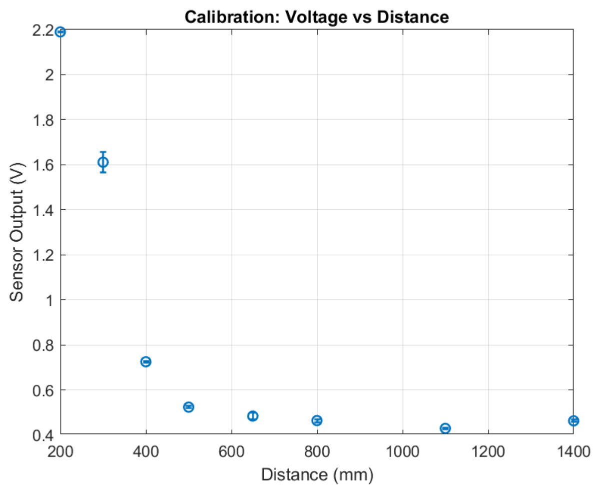

Figure 1. Sensor output voltage vs. distance (means ±1σ). Voltage computed as reading×5/1023. Trend decreases with distance; operating band (200–800 mm) emphasized.

An error plot showing predicted distance and actual distance for distances not included in your calibration routine.

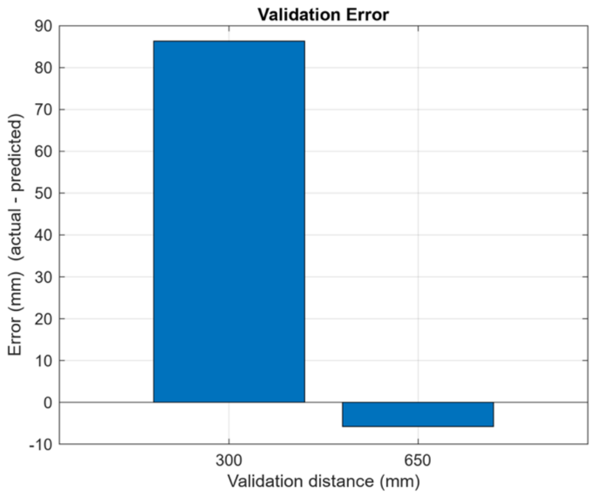

Figure 2. Validation Error between Validation Distances (300 mm vs 650 mm).

We evaluated the model on held-out distances not used for fitting (300 mm and 650 mm). RMSE (root-mean-square error) and MAE (mean absolute error) summarize the miss on these validation points: RMSE penalizes larger misses more strongly, while MAE reflects the typical absolute miss. With only two held-out points, RMSE is mostly driven by the larger of the two errors; in our data the near-range point contributes most of the error, which is consistent with the sensor's response being less linear near the edges of the calibrated band.

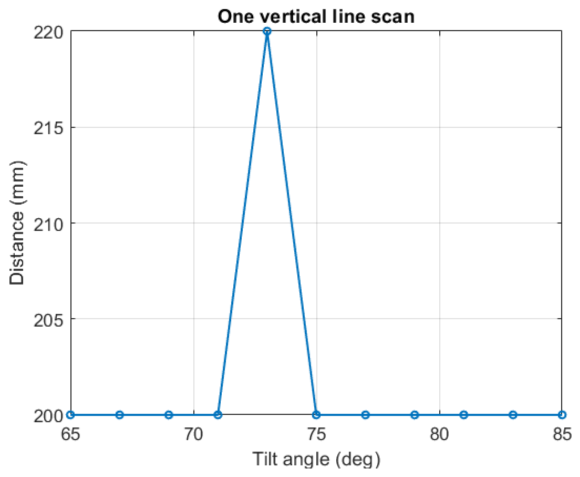

Figure 3. Vertical-line scan for distance against tilt angle

To generate this figure, we fixed the base servo at the center heading and swept only the tilt servo from low to high while streaming averaged sensor readings through the calibrated distance function. Angles on the x-axis are the reported tilt positions, and the y-axis is distance in millimeters. Most of the sweep sits near the front face of the box, so the curve is flat at about 200 mm. Around 73° the sensor's line of sight passes through the cutout, so the measured range jumps to a farther surface, producing the narrow peak near 220 mm in this plot. Once the tilt moves past the opening the distance returns to the front face level. This single-column slice is exactly what we expect from the scene geometry: one closer plane with a localized view to a farther plane. It also verifies the data path we use in the full raster, since the peak appears at a consistent angle and the baseline is stable with the chosen settle time and sample averaging.

Calibration

We held the sensor at a fixed pose and placed a target at known distances from the sensor face. At each distance we collected 120 samples and used the mean. We used distances of 200, 300, 400, 500, 650, 800, 1100, 1400 mm. Because the response flattens at long range, we calibrated on the 200–800 mm band with a simple inverse model:

$$measured\ dist_{mm} \approx k \cdot \frac{1}{reading} + c$$

We used only the 200–800 mm data where the response is reliable. The fitted constants are $k = 57{,}097.59$ and $c = 75.46$. On the Arduino we apply this formula and clamp outputs to 200–900 mm to avoid extrapolation beyond the calibrated band. If we re-calibrate later, only the two constants, k and c need to be updated.

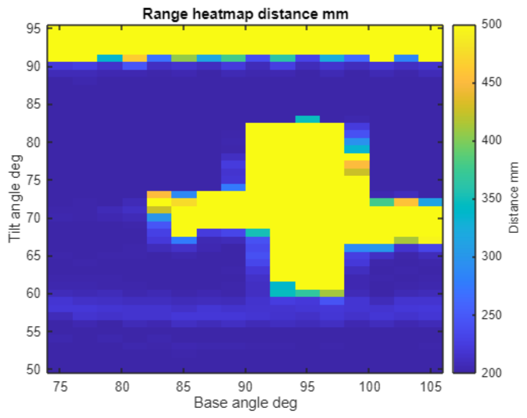

Figure 4. 3D heatmap for two-axis "scan" of target. Computes distance parameters when considering both horizontal base angle and vertical tilt angle

This heatmap in Figure 4 shows one full framed raster from the 2-servo scanner, generated with the setup in Figure 8 (appendix). The x-axis is base angle in degrees, the y-axis is tilt angle in degrees, and color encodes measured distance in millimeters within a 200–500 mm display band. The bright yellow plus shape in the center corresponds to the cutout, which exposes the background. The surrounding darker blue field is the front face of the cardboard, which is closer to the sensor at around 200–300 mm.

Mechanical Design Discussion

The original prototype for the jointing system of both servo motors and the infrared sensor relied on a single pivot point, an "arm" that would joint to both servo motor touch points. While the idea would simplify design, it created too much stress on the center of gravity of the "arm" which would limit system stability and servo mobility. Such issues could compromise data collection and therefore system accuracy. While succeeding designs differed slightly in the manner that they jointed servo motors or the sensor with the mounting surface of the body, the overall system retained a similar structure.

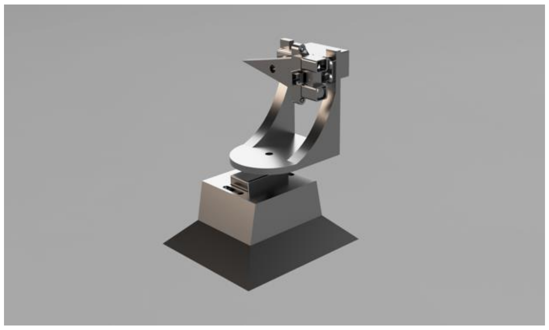

Figure 5. Mechanical Design Rendering for overall structural system. All non-servo motors, sensors, fasteners are 3D prototyped in PLA plastic. Primary servo motor serves as horizontal motion while secondary "arm" servo motor serves as vertical motion.

As seen in figure 5, the body consists of a central plate with the slot that would allow the main servo body to sit with some vertical offset to the top surface of this “bottom base”. The “bottom base” also had long open screw slots that would allow the M5 bolts to fasten to the body without being constrained to accurate screw hole spacing. The overall chamfer on the bottom half of the “bottom base” serves to create more surface area for the weight imbalance of the system to not compromise its stability. Additionally, it also servs the purpose of creating more friction with the resting surface in order to minimize any shifts with the torques created by the servo motors. This primary servo motor would then mount the “secondary base” through a “contact set” interface between the fastener that would thread into the servo motor and the cut out within the “secondary base”. The reasoning as to the peculiar shape of the “secondary base” lies with centering the final “piece” that would keep the infrared sensor centered along the horizontal axis of rotation. The final “arm piece” is kept sleek for weight gain and allows for easy fastener mount interface between the sensor and itself.

Electrical Design Discussion

The basis for the circuit design rested on allowing the microcontrollers “orders” to control the servo motor’s motions, aspects integral to the process of “scanning”. In adjacence, connecting the infrared sensor would allow for the data collected by the sensor to be transmuted to the microcontroller for later processing within the Arduino IDE.

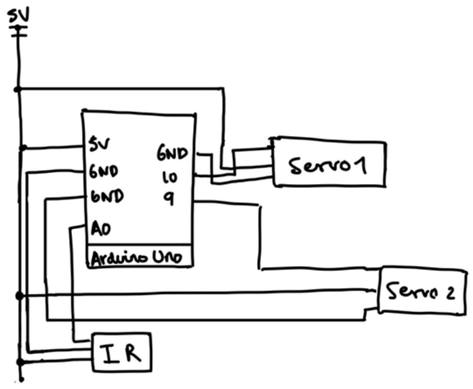

Figure 6. Circuit Diagram for Overall System. Microcontroller computes commands for servo motors and receives data from infrared sensor.

While there can be modifications made in terms of adding resistive elements to the system, perhaps in order to enhance data reliability. This design allows for a simple configuration that does not open itself to overcomplications in circuit issues. Additionally, another modification, perhaps in a future iteration, would be to better manage cabling and perhaps collaborate with mechanical design to internally route cable for aesthetic purposes. One idea kicked around was the concept of integrating the breadboard and microcontroller into the “bottom base” of the body in order to allow for one structural body to contain all hardware. This idea did not come to fruition during our prototyping, for sake of experimentation and hardware testing. Yet with the knowledge on how to calibrate and set-up the sensors as well as the servo motors, more aesthetic and management integration can be carried out with future iterations.

Software Design Discussion

Our software is split between Arduino and MATLAB so the scan is stable and the data flow is simple. On the Arduino, the base servo moves one step to a new column, then waits while the tilt servo sweeps that column, so the sensor looks from low to high. When the sweep finishes the base takes the next step. The tilt direction alternates each column in a serpentine pattern to cut motion time and backlash. Each point averages several sensor reads after a short settle time, then converts raw ADC to distance in millimeters using the calibrated inverse model mentioned in the calibration section, with outputs clamped to the calibrated band for stability.

The Arduino streams rows as CSV in the format base_deg_abs, scan_deg_abs, raw_adc, dist_mm at 115200 baud and wraps each full raster between the markers #FRAME_START and #FRAME_END. MATLAB opens the serial port, waits for the start marker, reads rows until the end marker, and bins each sample onto the commanded grid by converting angles to indices with bi = round((base − base_left)/base_step) + 1 and ti = round((tilt − tilt_bottom)/tilt_step) + 1. If multiple samples land in the same cell they are averaged. This framing and gridding ensure we capture exactly one complete frame, neither missing cells nor double counting a partial sweep, which makes the heatmap more accurate.

The heatmap itself is a 2D image where the x axis is base angle, the y axis is tilt angle, and the pixel value is distance in millimeters. For depth contrast we display a focused band such as 200 to 500 mm so the cutout appears clearly against the background. When we generate a 3D point cloud, we map each sample to Cartesian coordinates by treating base as yaw and tilt as pitch and using $x = r \cos(\text{pitch}) \cos(\text{yaw})$, $y = r \cos(\text{pitch}) \sin(\text{yaw})$, $z = r \sin(\text{pitch})$, with r in meters from the calibrated distance. All key parameters live at the top of the code, including center angles, arc limits, step sizes, zero offsets, dwell, and samples per point, which makes it easy to retune the window and resolution for different object sizes and working distances without changing the rest of the pipeline.

An area for improvement is adding a host-controlled command protocol and live preview. MATLAB should be able to start and stop scans, switch between single-line and full-frame modes, and change step sizes or the display band at runtime. A preview that updates column by column with progress and autoscaled colors would shorten iteration. The capture path should add basic integrity checks, timestamps, and simple retries on malformed rows, plus metadata logging of angles, steps, and calibration constants for each frame. Finally, adaptive dwell and averaging based on recent variance would slow down only when the signal is unstable, cutting total scan time without losing quality.

Appendix

Model Set-Up

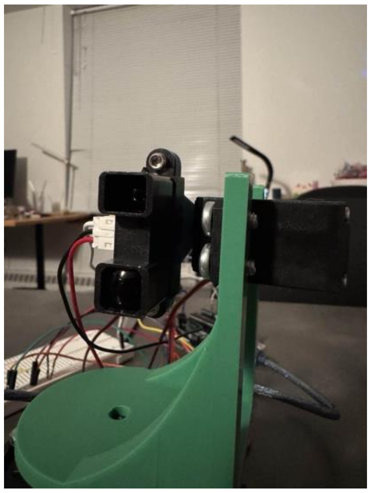

Figure 7. Model set-up for 2D visualization for horizontal scan of target

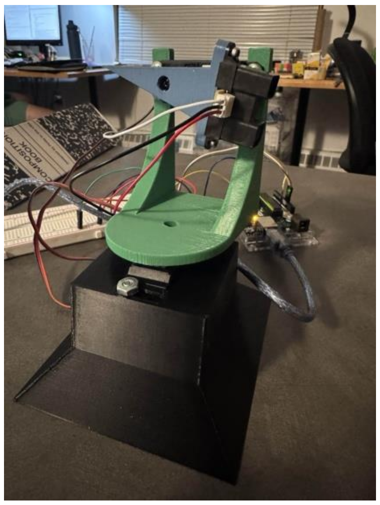

Figure 8. Model set-up for 3D visualization of horizontal and vertical can of target

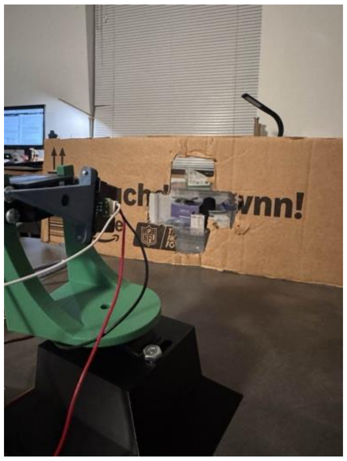

Figure 9. Model set-up during data collection "scan" of target

Arduino Code

```
#include <Servo.h>

// pins
const uint8_t PIN_SERVO_BASE = 10;
const uint8_t PIN_SERVO_SCAN = 9;
const uint8_t PIN_SENSOR     = A0;
```

```
// serial
const uint32_t BAUD = 115200;

// zero offsets
int BASE_ZERO_OFFSET = 0;
int SCAN_ZERO_OFFSET = -5;

// base sweep
int BASE_CENTER_ABS = 90;
int BASE_ARC_HALF   = 15;      // arc half 15 deg, total 30 deg
int BASE_LEFT_ABS   = BASE_CENTER_ABS - BASE_ARC_HALF;
int BASE_RIGHT_ABS  = BASE_CENTER_ABS + BASE_ARC_HALF;
int BASE_STEP_DEG   = 2;
uint16_t BASE_DWELL_MS = 120;

// tilt sweep
int SCAN_BOTTOM_REL = -35;
int SCAN_TOP_REL    = 10;
int SCAN_STEP_DEG   = 1;

// timing and sampling
uint16_t SERVO_SETTLE_MS = 140;
uint8_t  SAMPLES_PER_PT  = 8;

Servo servoBase, servoScan;

// clamp any angle to the 0 to 180 range
int clamp180(int d){
  return d < 0 ? 0 : (d > 180 ? 180 : d);
}

// convert logical base angle to servo command angle
int baseToServoDeg(int logicalDeg){
  return clamp180(logicalDeg + BASE_ZERO_OFFSET);
}

// convert relative tilt angle to servo command angle
int scanToServoDegFromRel(int relDeg){
  return clamp180(90 + SCAN_ZERO_OFFSET + relDeg);
}

// average n analog readings from the sensor
```

```
int readSensorAvg(uint8_t n){
  long acc = 0;
  for(uint8_t i=0;i<n;i++){
    acc += analogRead(PIN_SENSOR);
    delay(2);
  }
  return (int)(acc / n);
}

// convert raw ADC to distance in millimeters using the calibration
float rawToMM(int raw){
  raw = constrain(raw, 1, 1023);
  const float k = 57097.59f;
  const float c = 75.46f;
  float mm = k * (1.0f / raw) + c;
  if(mm < 200.0f)  mm = 200.0f;
  if(mm > 900.0f)  mm = 900.0f;
  return mm;
}

// move both servos to a pose and wait for settling
void goPose(int baseAbsDeg, int scanAbsDeg){
  servoBase.write(clamp180(baseAbsDeg));
  servoScan.write(clamp180(scanAbsDeg));
  delay(SERVO_SETTLE_MS);
}

// set up serial, compute limits, attach servos, and center the rig
void setup(){
  Serial.begin(BAUD);
  delay(400);

  BASE_LEFT_ABS  = BASE_CENTER_ABS - BASE_ARC_HALF;
  BASE_RIGHT_ABS = BASE_CENTER_ABS + BASE_ARC_HALF;

  servoBase.attach(PIN_SERVO_BASE, 500, 2500);
  servoScan.attach(PIN_SERVO_SCAN, 500, 2500);

  goPose(baseToServoDeg(BASE_CENTER_ABS), scanToServoDegFromRel(0));

  Serial.println("# DIY 3D Scanner framed");
  Serial.println("# base_deg_abs,scan_deg_abs,raw_adc,dist_mm");
}
```

```
// perform one framed raster scan and stream CSV samples
void loop(){
  int scanBot = scanToServoDegFromRel(SCAN_BOTTOM_REL);
  int scanTop = scanToServoDegFromRel(SCAN_TOP_REL);

  Serial.println("#FRAME_START");   // mark start of frame

  int col = 0;

  // outer loop over base columns
  for(int baseAbs = BASE_LEFT_ABS; baseAbs <= BASE_RIGHT_ABS; baseAbs +=
BASE_STEP_DEG, col++){

    if((col & 1) == 0){
      // inner loop tilt bottom to top for even columns
      goPose(baseAbs, scanBot);
      for(int scanAbs = scanBot; scanAbs <= scanTop; scanAbs += SCAN_STEP_DEG){
        goPose(baseAbs, scanAbs);
        int raw = readSensorAvg(SAMPLES_PER_PT);
        float mm = rawToMM(raw);
        Serial.print(baseAbs); Serial.print(',');
        Serial.print(scanAbs); Serial.print(',');
        Serial.print(raw);     Serial.print(',');
        Serial.println(mm, 1);
      }
    }else{
      // inner loop tilt top to bottom for odd columns
      goPose(baseAbs, scanTop);
      for(int scanAbs = scanTop; scanAbs >= scanBot; scanAbs -= SCAN_STEP_DEG){
        goPose(baseAbs, scanAbs);
        int raw = readSensorAvg(SAMPLES_PER_PT);
        float mm = rawToMM(raw);
        Serial.print(baseAbs); Serial.print(',');
        Serial.print(scanAbs); Serial.print(',');
        Serial.print(raw);     Serial.print(',');
        Serial.println(mm, 1);
      }
    }

    // short dwell after finishing a column
    delay(BASE_DWELL_MS);
  }
  Serial.println("#FRAME_END");     // mark end of frame
  delay(400);
}
```

MATLAB Code

```
% full_scan_capture_by_frame_heatmap_only.m
clear; clc;

% serial port settings
PORT = "COM4";
BAUD = 115200;

% angles and steps must match Arduino
BASE_ZERO_OFFSET = 0;
SCAN_ZERO_OFFSET = -5;

BASE_CENTER_ABS = 90;
BASE_ARC_HALF = 15;
BASE_STEP_DEG = 2;

SCAN_BOTTOM_REL = -35;
SCAN_TOP_REL = 10;
SCAN_STEP_DEG = 1;

% compute absolute limits and plotting axes
BASE_LEFT_ABS = BASE_CENTER_ABS - BASE_ARC_HALF;
BASE_RIGHT_ABS = BASE_CENTER_ABS + BASE_ARC_HALF;
SCAN_BOTTOM_ABS = 90 + SCAN_ZERO_OFFSET + SCAN_BOTTOM_REL;
SCAN_TOP_ABS = 90 + SCAN_ZERO_OFFSET + SCAN_TOP_REL;

base_axis = BASE_LEFT_ABS:BASE_STEP_DEG:BASE_RIGHT_ABS;
scan_axis = SCAN_BOTTOM_ABS:SCAN_STEP_DEG:SCAN_TOP_ABS;
nPan = numel(base_axis);
nTilt = numel(scan_axis);

% open serial and wait for the start of one frame
s = serialport(PORT, BAUD);
```

```
configureTerminator(s, "CR/LF");
flush(s);
while true
t = strtrim(readline(s));
if startsWith(t, "#FRAME_START"); break; end
end

% read csv rows until the end of the frame
B = []; T = []; D = [];
while true
t = strtrim(readline(s));
if startsWith(t, "#FRAME_END"); break; end
if strlength(t)==0 || startsWith(t, "#"); continue; end
p = split(t, ','); if numel(p) ~= 4, continue; end
b = str2double(p{1});
tt = str2double(p{2});
d = str2double(p{4});
if any(isnan([b tt d])), continue; end
B(end+1) = b; T(end+1) = tt; D(end+1) = d; %#ok<AGROW>
end
clear s

% bin samples to the commanded grid
b_idx = round((B - BASE_LEFT_ABS) / BASE_STEP_DEG) + 1;
t_idx = round((T - SCAN_BOTTOM_ABS)/ SCAN_STEP_DEG) + 1;
ok = b_idx>=1 & b_idx<=nPan & t_idx>=1 & t_idx<=nTilt & isfinite(D);
subs = [t_idx(ok).', b_idx(ok).']; % row, col indices as columns
Dgrid = accumarray(subs, D(ok).', [nTilt, nPan], @mean, NaN);

% choose a display band in mm for clear contrast
FRONT_MM = 200;
BACK_MM = 500;
Dplot = min(max(Dgrid, FRONT_MM), BACK_MM);

% draw the range heatmap
figure('Color','w');
imagesc(base_axis, scan_axis, Dplot);
set(gca,'YDir','normal'); axis tight
xlabel('Base angle deg'); ylabel('Tilt angle deg');
title('Range heatmap distance mm');
c = colorbar; c.Label.String = 'Distance mm';
```

```
caxis([FRONT_MM BACK_MM]);
```