PoE Lab 2 - 3D Scanner

Kyle Combes and Rachel Hwang

September 26, 2017

Abstract

In this lab, we used an Arduino, two servos, and an infrared rangefinder to construct a 3D plot of a large 'R' cut out of cardboard. Data was minorly preprocessed on the Arduino before being sent to a computer via serial for more processing and display. In the end, a 3D scatter plot was generated depicting the scanned object.

1 Constructing the 3D Scanner

Using SolidWorks, we designed parts to create a 3D scanner that includes two hobby servo motors and an infrared distance sensor. All parts were made intended to be laser cut out of 3/16" hardboard that could be wedged together. The bottom servo motor sat in a box base acting as the panning mechanism. Attached above to the side is the second servo motor creating the tilt mechanism for the sensor attached in the middle of the two axes. This ensured that the sensor's center would not change when it panned or tilted.

Our SolidWorks model can be seen in Figure 1, and our final, physical scanner can be seen in Figure 2.

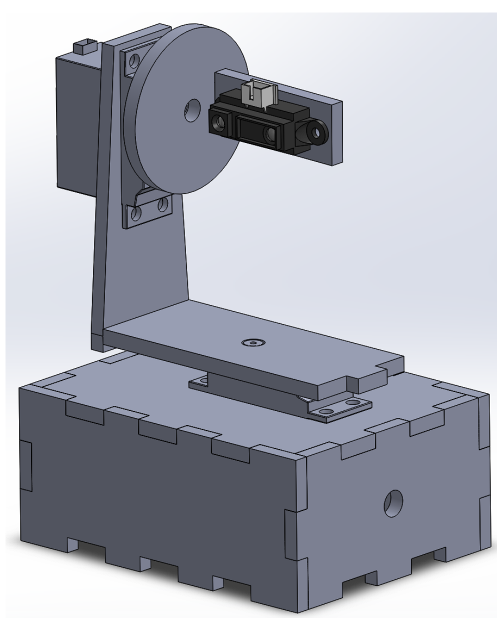

Figure 1: The SolidWorks model of the 3D scanner

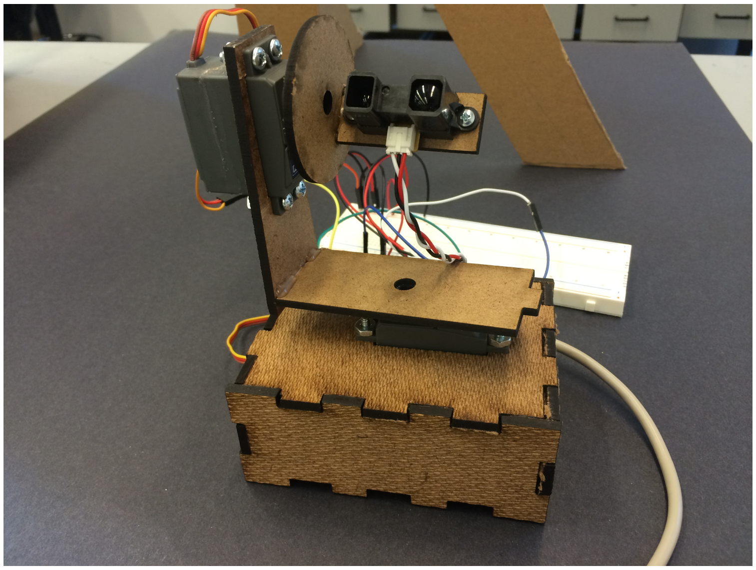

Figure 2: The model made from laser cut hardwood of the 3D scanner

2 Calibrating the Rangefinder

The first step to using a rangefinder to accurately measure distances was to calibrate it. This would allow us to convert the value read from the sensor to an actual distance.

The value read from the IR sensor was in the range of 0 to 1023, since the Arduino uses a 10-bit analog to digital converter to read the input voltage. However, it seemed like an unnecessary extra step to us to convert this value to a voltage and then map that voltage to a distance value. Thus, we decided to directly map the 10-bit value to a distance in inches.

To calibrate the sensor, we placed a large piece of cardboard in front of the IR sensor. Using the serial monitor in the Arduino IDE, we observed the raw readings from the IR sensor and picked the mode of ten to twenty consecutive readings to be our measured value for that distance. Then we moved the cardboard back two to three inches and repeated the process. Our collected data can be seen in Figure 3.

Switching our independent and dependent variables allowed us to look at the distance as a function of sensor reading. Using the `polyfit` function in MATLAB, we were able to extract the quartic function $y = 5.24 \times 10^{-9}x^4 - 8.33 \times 10^{-6}x^3 + 4.83 \times 10^{-3}x^2 - 1.26x + 1.41 \times 10^2$, which related the raw sensor reading to the distance. This is illustrated in Figure 4.

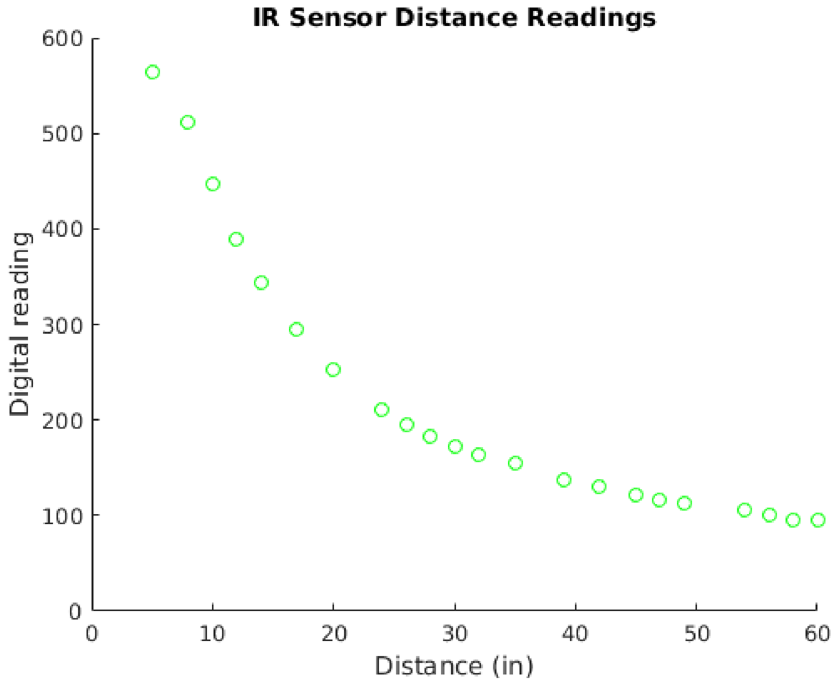

Figure 3: Our raw IR sensor values (as measured by the 10-bit ADC) as a function of distance from the cardboard.

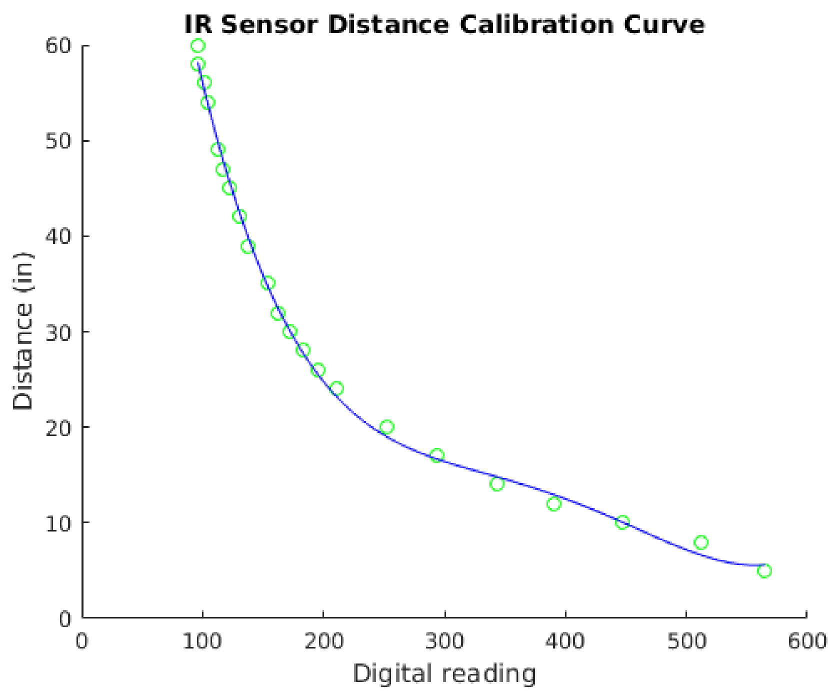

Figure 4: Switching our independent and dependent variables allowed us to extract a quartic function, $y = 5.24 \times 10^{-9}x^4 - 8.33 \times 10^{-6}x^3 + 4.83 \times 10^{-3}x^2 - 1.26x + 1.41 \times 10^2$, relating our IR sensor reading to the actual distance. The green points are the measured values and the blue curve is the quartic function.

2.1 Error Plot

In order to check our calibration, we took some new measurements at various distances from the cardboard and checked the output of our function against the actual distance measurements. If our calibration were perfect, all of the points would lie on the line $y = x$ because there would be a 1:1 correlation between the measured and actual values. In practice, the calculated values were typically within an inch of the correct value (see Figure 5).

However, it appears that the error would be significantly decreased if the line were shifted slightly to the left. This discrepancy could be due to the fact that the calibration and error plots were done under different lighting conditions (daytime and nighttime). The sunlight during the calibration session could've shifted all of the readings (since sunlight contains IR light). Without that constant influx, the readings would've all shifted by the same amount.

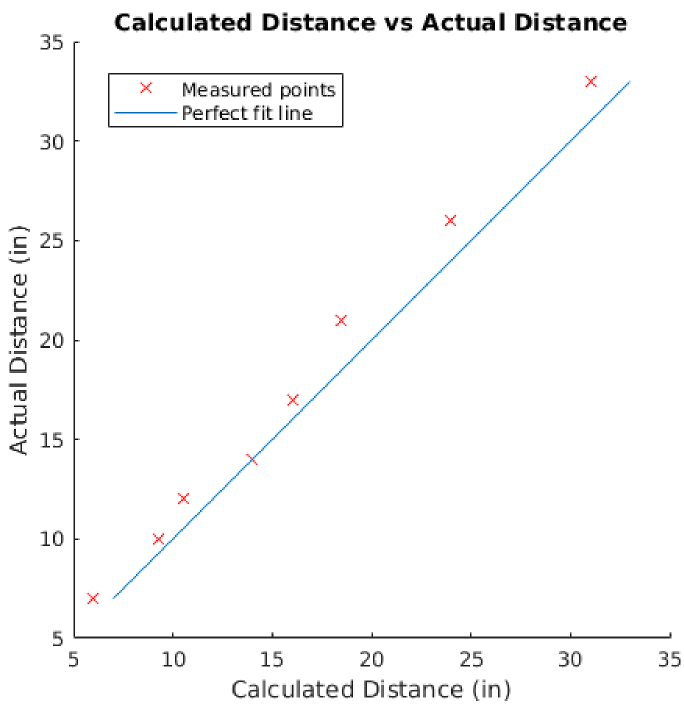

Figure 5: Some testing of our calibration curve showed that we were typically interpreted the distance within 1 inch of its correct value. However, the data may have been skewed slightly due to different lighting conditions during calibration and testing.

3 Collecting the Scan Data with the Arduino

Once the IR rangefinder was calibrated and mounted, we programmed the Arduino to sweep back and forth, collecting data points. Since we were panning and tilting about the origin and reading distances from the origin, this meant we were dealing with spherical coordinates. We decided to deal entirely with spherical coordinates on the Arduino and send them to the computer in that format.

We defined θ_{min}, θ_{max}, ϕ_{min}, and ϕ_{max} to restrict our ranges to $80 \leq \theta \leq 100$ and $75 \leq \phi \leq 120$. This helped minimize scanning time and avoid scanning unwanted nearby objects. We also defined `thetaStep` and `phiStep` to determine our angle deltas for each step. We settled on a value of 1 for both, so that we collected points every degree in each direction.

To actually collect the data, the Arduino began with the pan servo at θ_{min} and the tilted servo at ϕ_{min}. It then sampled the IR sensor twenty times and computed the mean distance, in order to avoid any erroneous readings throwing us off. Once it computed the mean, the value was transmitted to the computer along with the values for θ and ϕ as a tab-separated string.

Then, if θ was at the end of its range, the scanner tilted, collected another scan point, and reversed `thetaStep` to pan in the other direction. Once the scanner reached θ_{max} and ϕ_{max}, it reversed directions and began the scan again.

4 Visualizing the Scan

On the computer, a Python program received the data sent by the Arduino over serial. Upon receiving a transmission, several steps were required before a final figure could be generated.

4.1 Converting from Spherical to Cartesian

The first steps to handling the Arduino transmission were to decode the tab-separated string of values and convert them from spherical coordinates to Cartesian using the following equations (where r was the measured distance and θ and ϕ were the angles, as illustrated in Figure 6).

$$x = rsin(\phi)cos(\theta)$$
$$y = rsin(\phi)sin(\theta)$$
$$z = rcost(\phi)$$

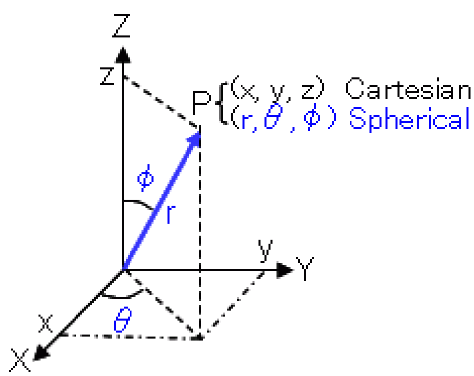

Figure 6: The spherical coordinate system used by the 3D scanner. Points were converted to Cartesian on the computer before plotting.

Once the point was converted to Cartesian, it was stored in a PointsCollection data structure. The PointsCollection data structure was designed to be configurable to only store a certain number of points such that, once the limit was reached, the oldest data point would be dropped. This allowed the scanner to keep running and scan new objects, discarding the oldest points as it progressed. For our scans, we decided to only keep one point for each value of θ and ϕ, so our max number of points was $(\theta_{max} - \theta_{min})(\phi_{max} - \phi_{min}) = 900$.

4.2 Taking a 2D Scan

Taking a cross-sectional slice of the 'R' (just above where the lines converged in the middle) resulted in Figure 7. In order to take the scan, the full pan-tilt setup (Figure 2) was used, but the tilt functionality was simply disabled.

At the bottom of the scan, where $y < 20$, you can see where the scanner picked up the middle of the R. The rest of the points at further distances are either noise or objects picked up in the background.

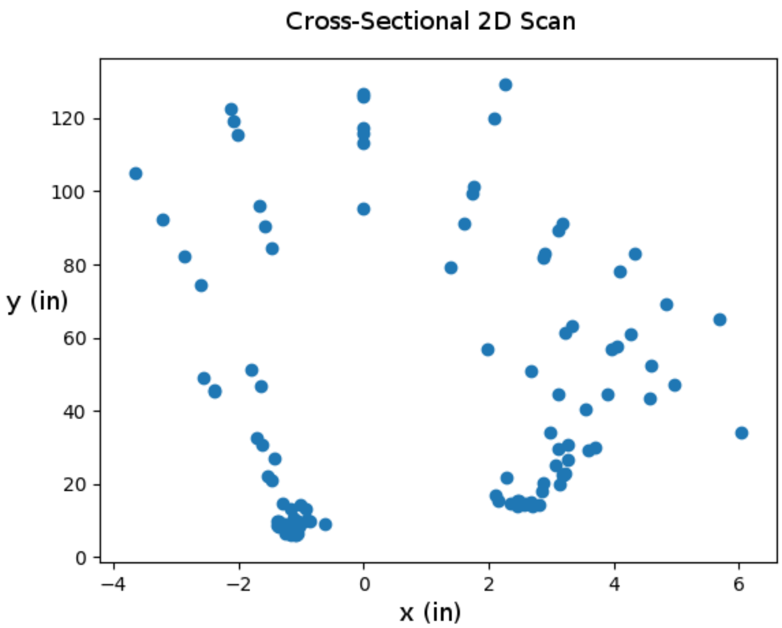

Figure 7: Taking a 2D scan of the middle of the 'R' (just above where the lines converge in the middle of the letter) results in a cross-sectional view of the letter. At the bottom of the scan, where $y < 20$, you can see where the scanner picked up the R.

4.3 Cleaning the Data

We also defined a parameter in our program, `MAX_DIST`, to ignore points more than a specific distance from the scanner. This helped to filter out anything the scanner picked up behind the object we were attempting to scan. Since we placed the letter we were scanning about 10-15 inches away from the scanner, we decided 30 inches was a good cutoff point. This way we could see if the scanner ever got confused and thought that the letter was a little further away than it actually was, but we also avoided plotting any background objects.

4.4 Plotting the Points

Once the points were filtered and converted to Cartesian, we plotted them in 3D using the Python library `matplotlib`. The results can be seen in Figure 8.

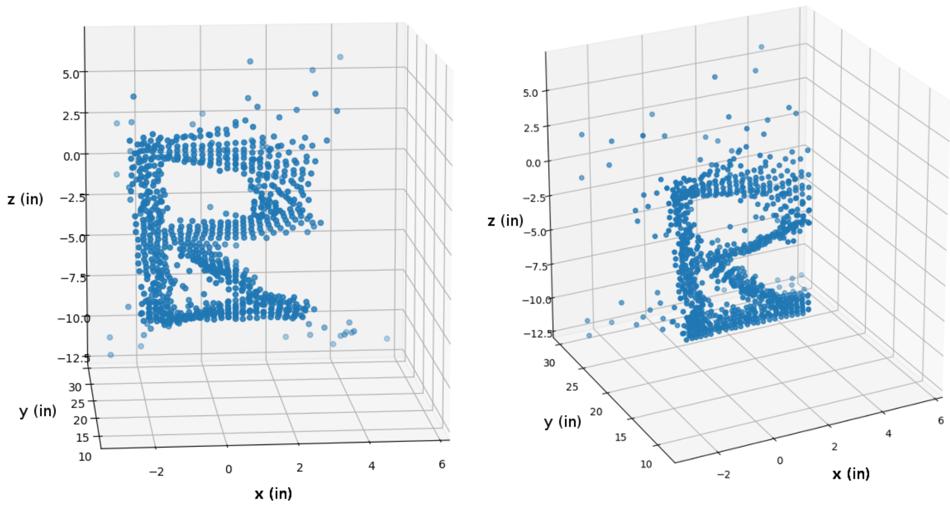

Figure 8: Our scanned cardboard letter "R" after minimal filtering.

5 Conclusion

All in all, our scanner performed quite admirably. We initially ran into some trouble with inconsistent distance readings, which caused our object to be indistinguishable on the scan, but we learned that the IR sensor needed around 50ms to perform a scan and adjusted our code accordingly. This made the scan much, much cleaner.

Even so, there was a fair bit of noise in the top-right corner. We think this might be due to inconsistent servo tilt deltas, as it often appeared as though the scanner tilted more some times than others. However, the slanted leg of the 'R' appeared pretty crisp, so perhaps it was not the cause. Confirming or disproving this would require closer monitoring of the scanner during scanning and perhaps measures the changes in the angle.

The ground is also noticeable in the bottom of the scan, but that is just an artifact caused by the conditions of the scan. To get rid of it (without simply cropping the image), one would have to suspend the letter in the air, perhaps with some thin thread. But that's beyond the scope of this project.

6 Code

The full source code can also be found in this GitHub repository.

6.1 MATLAB Calibration

```
% Calibration data
dist = [5 8 10 12 14 17 20 24 26 28 30 32 35 39 42 45 47 49 54 56 58 60];
reading = [565 512 447 390 343 294 252 211 195 183 172 163 154 138 130 122 117 113 105 101 96 96];

% Prep the readings figure
figure(1); clf; hold on;
title('IR Sensor Distance Readings');
```

```
xlabel('Distance (in)');
ylabel('Digital reading');

% Plot the points
plot(dist, reading, 'go');

% Prep the calibration curve figure
figure(2); clf; hold on;
title('IR Sensor Distance Calibration Curve');
xlabel('Digital reading');
ylabel('Distance (in)');

% Plot the points with the dependent and independent variable flipped
plot(reading, dist, 'go');

% Fit a parabola
parab_coeffs = polyfit(reading, dist, 4);

% Plot the parabola
parab_x = linspace(min(reading), max(reading))';
parab_A = [parab_x.^4 parab_x.^3 parab_x.^2 parab_x ones(length(parab_x),1)];
parab_y = parab_A * parab_coeffs';
plot(parab_x, parab_y, 'b');


%% Make an error plot
calculated = [6 9.3 10.5 14 16 18.5 24 31];
actual = [7 10 12 14 17 21 26 33];

figure(3); clf; hold on;
plot(calculated, actual, 'rx');
x_y = [min(actual) max(actual)];
plot(x_y, x_y);
title('Calculated Distance vs Actual Distance');
xlabel('Calculated Distance (in)');
ylabel('Actual Distance (in)');
axis square;
legend('Measured points', 'Perfect fit line');
```

6.2 Arduino

```

#include <Servo.h>

#define IR_SENSOR A0
#define PAN_SERVO 3
#define TILT_SERVO 5
#define THETA_MIN 80
#define THETA_MAX 100
#define PHI_MIN 75
#define PHI_MAX 120

Servo servoPan;  // Create servo object to control the pan servo
Servo servoTilt;  // Create servo object to control the tilt servo

int theta = THETA_MIN; // Pan
int phi = PHI_MIN; // Tilt
int thetaStep = 1; // Measure every 1 degree
int phiStep = 1; // Measure every 1 degree
```

```
bool justTilted = false;

String result="";

void setup() {
  // Setup the servos and being serial communication with a computer
  servoPan.attach(PAN_SERVO);
  servoTilt.attach(TILT_SERVO);
  Serial.begin(9600);
}

// Move the panning servo to a particular angle
void setTheta(int theta) {
  if (theta > THETA_MAX) {
    theta = THETA_MAX;
  } else if (theta < THETA_MIN) {
    theta = THETA_MIN;
  }
  servoPan.write(theta);
}

// Move the tilt servo to a particular angle
void setPhi(int phi) {
  if (phi > PHI_MAX) {
    phi = PHI_MAX;
  } else if (phi < PHI_MIN) {
    phi = PHI_MIN;
  }
  servoTilt.write(phi);
}

// Take the average distance reading over 20 readings
float READINGS_PER_ANGLE = 20.0;
float distanceSum = 0.0;
float readCount = 0.0;

void loop() {

  // Take a reading, add it to our running sum, and increment our read counter
  distanceSum += readDistFromSensor();
  readCount++;

  if (readCount == READINGS_PER_ANGLE) {

    // Calculate the average distance reading
    float distance = distanceSum / READINGS_PER_ANGLE;

    // Send our position and distance reading to the computer
    transmitData(distance, theta, phi);

    // Pan the sensor in the correct direction
    pan();

    // Reset counter and sum
    distanceSum = 0.0;
    readCount = 0.0;

    // Wait 100ms for Python program to receive and process
//    delay(100);
```

```
  } else {
    // Wait 10ms between readings for the sensor to measure again
    delay(50);
  }
}

// Pan the scanner in the correct direction
void pan() {
  // Update theta
  theta += thetaStep;

  // If we've gone past the max or min, change directions and go back the other way
  if (theta > THETA_MAX || theta < THETA_MIN) {
    if (justTilted) {
      // If we just tilted and took a scan, change pan directions
      thetaStep = -thetaStep;
      theta += 2*thetaStep;
      justTilted = false;
    } else { // Tilt and take a scan point before panning
      tilt();
      justTilted = true;
    }
  }

  // Move the servo
  setTheta(theta);
}

// Tilts the scanner in the correct direction
void tilt() {
  // Update phi
  phi += phiStep;

  // If we've gone past the max or min, change directions and go back the other way
  if (phi > PHI_MAX || phi < PHI_MIN) {
    Serial.println(result + "Phi too big/small! " + phi + " vs " + PHI_MAX);
    phiStep = -phiStep;
    phi += 2*phiStep;
  }

  // Move the servo
  setPhi(phi);
}

// Get the distance (in inches) measured by the IR sensor
float readDistFromSensor() {
  // Slope and intercept determined by calibration experiment
  float d = (float)analogRead(IR_SENSOR);
  // Convert reading to inches
  return 0.0000000052465*d*d*d*d - 0.0000083255*d*d*d + 0.00483*d*d - 1.2578*d + 141;
}

// Send data to the computer over serial
void transmitData(float radius, float theta, float phi) {
  Serial.println(result + radius + "\t" + theta + "\t" + phi);
}
```

6.3 Computer

```python
#!/usr/local/bin/python

from serial import Serial
import matplotlib.pyplot as pyplot
import time
import math


# Object that has properties of (x,y,z)
class Point:
    def __init__(self, x, y=0.0, z=0.0):
        self.x = x
        self.y = y
        self.z = z


# Make a list of points
class PointCollection:
    def __init__(self, points_to_keep):
        self.points_to_keep = points_to_keep
        self.points = []

    # Add points to the list
    def add_point(self, point):
        if len(self.points) >= self.points_to_keep:
            self.points.pop(0)  # Remove the oldest point
        self.points.append(point)

    # Go through all the points and return all x values
    def get_x_values(self):
        return [p.x for p in self.points]

    # Go through all the points and return all y values
    def get_y_values(self):
        return [p.y for p in self.points]

    # Go through all the points and return all z values
    def get_z_values(self):
        return [p.z for p in self.points]

NUM_POINTS_TO_KEEP = 900 # Number of points to keep on the scatter plot
MAX_DIST = 30  # Maximum distance (in), to remove outliers

#  Initialize stuff
cxn = Serial('/dev/ttyACM0', baudrate=9600)
points = PointCollection(NUM_POINTS_TO_KEEP)


def read_serial():
    """
        Attempt to read the radius (distance) and angle from the Arduino.
        Returns a tuple in the format (radius, theta) if successful, or False
            otherwise.
    """
    while cxn.inWaiting() < 1:
        pass
    data = cxn.readline()
```

```python
    if data:
        try:
            data = data.decode('UTF-8')
            res = data.split('\t')
            if len(res) == 3:
                return float(res[0]), float(res[1]), float(res[2])
        except UnicodeDecodeError:
            pass
        except ValueError:
            pass  # Error casting to float
    return False, False, False


# Convert spherical to cartesian coordinates
def spherical_to_cartesian(radius, theta, phi):
    theta = math.radians(theta)
    phi = math.radians(phi)
    x = radius*math.sin(phi)*math.cos(theta)
    y = radius*math.sin(phi)*math.sin(theta)
    z = radius*math.cos(phi)
    return x, y, z


fig = pyplot.figure()
fig.show()  # Show the figure
ax = fig.add_subplot(111, projection='3d')  # Set up 3D plot
while True:
    (dist, theta, phi) = read_serial()
    if dist:
        (x, y, z) = spherical_to_cartesian(dist, theta, phi)
        if y < MAX_DIST:
            points.add_point(Point(x, y, z))
            # Print out the received data to the console, just in case we need to debug
            print('Radius: {0:0.3f}\tTheta: {1:0.3f}\tPhi: {2:0.3f}\tx:{3:0.3f}\ty:{4:0.3f}\tz:{5:0.3f}'.format(dist, theta, phi, x, y, z))
            # Plot our scan
            ax.cla()  # Clear figure
            ax.scatter(points.get_x_values(), points.get_y_values(), points.get_z_values())
            pyplot.draw()  # Redraw the figure
            pyplot.pause(0.001)  # Wait for it to render
        else:
            # Print out the received data to the console, just in case we need to debug
            print('Out of range: Radius: {0:0.3f}\tTheta: {1:0.3f}\tPhi: {2:0.3f}\tx:{3:0.3f}\ty:{4:0.3f}\tz:{5:0.3f}'.format(dist, theta, phi, x, y, z))
```