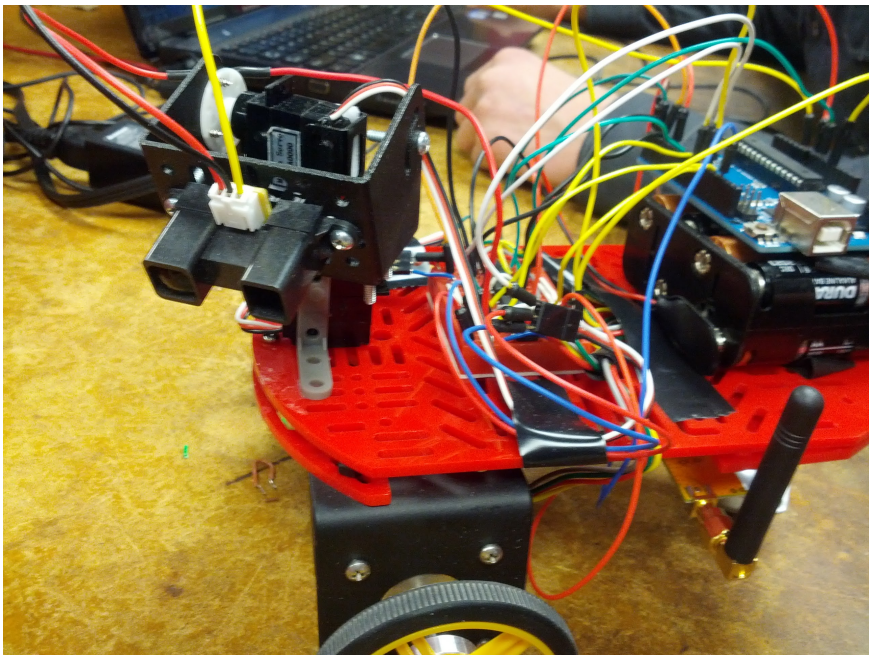
Project Report - 3D Mapping (Jank)

December 14, 2013

Team Kickass:
Max Allen, Bryan DiLaura, Andrew Huang, Ryan Montoya, Dylan Way

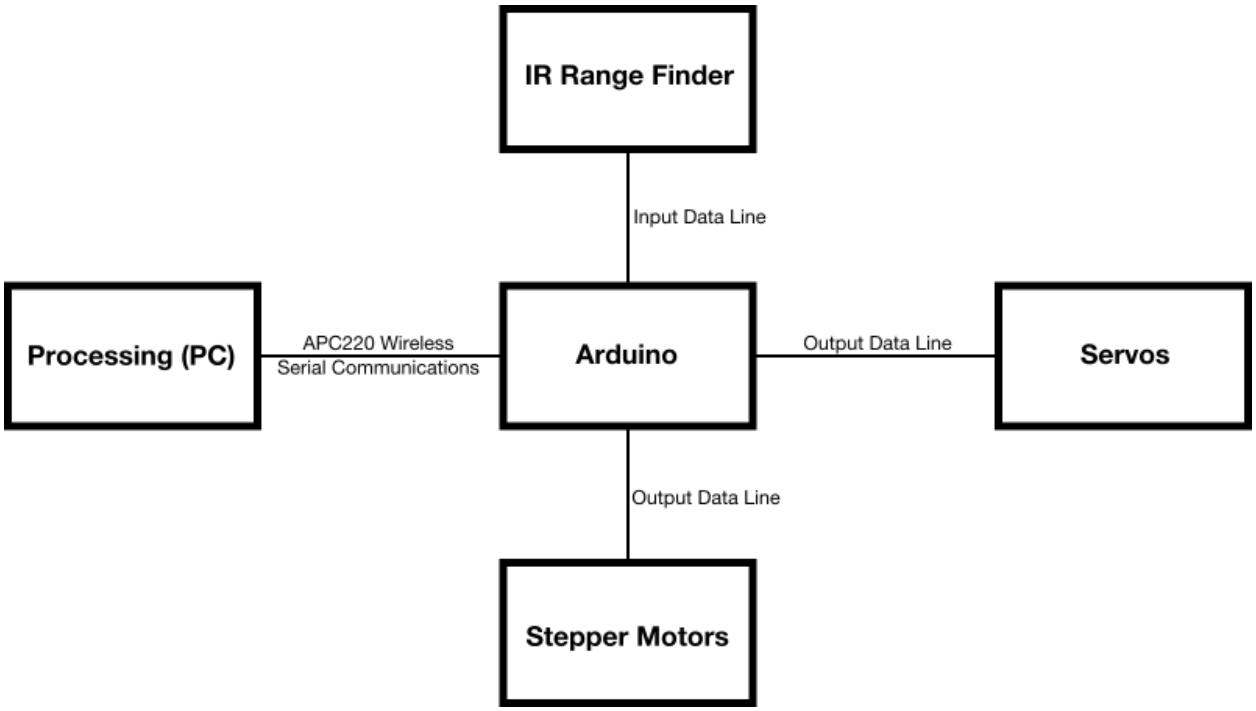


Introduction:

The purpose of our project is to create a robot with both autonomous and manual control, capable of collecting point clouds, which is then sent to a computer for processing. This data is then used to recreate the surrounding environment as a 3D image. This process could then be taken and potentially be utilized in military, search and rescue, automotive, geography, or art/gaming applications.

Implementation:

A 3D scanning robot is a very complex piece of technology to put together. Every part of the project will be described in detail, but to get a general idea of the major parts, the following block diagram can be utilized:



This block diagram describes the major flow of all data within our systems. At the center of everything, is the Arduino, which in concert with the computer operator (processing), drives the entire system. The Arduino controls the servos and stepper motors, in order to direct the range-finder, collecting a point cloud. These individual components are precise enough to where we were not required to include feedback systems. This information (many angles in combination with the distance) is number-crunched to then send an x,y,z coordinate to the computer for interpretation and drawing.

Sweep Function

One of the most important components in this project is a Sharp GP2Y0A02YK0F infrared range-finder, which is mounted onto two servos (aligned to the x and y planes). These servos step through a predetermined array of values using the built in Servo functions

provided by the *Arduino IDE*, taking distance measurements and keeping track of the angle that it is pointing. The range-finder, working with the servos, sweeps through the point cloud, taking analog-read measurements from the infrared sensor multiple times, converting them to a distance using a 5th degree polynomial curve, and averaging to find the distance from the robot (this is done to smooth out possible anomalies). These distances are then passed into a function which converts the angle at which the servos are facing and the distance into a coordinate on a 3D map using trigonometry and spherical coordinate transformations. These functions also take values stored from the movement of the robot as well as the the direction the robot is facing (noted below) to calculate the coordinates. All of these calculations are performed using the math.h functions that allows for usage of sine and cosine calculations. These coordinates are then sent to the computer via the APC220 wireless serial device, and then interpreted using *Processing* to generate a 3D map (both of which will be described in more detail later).

Steppers

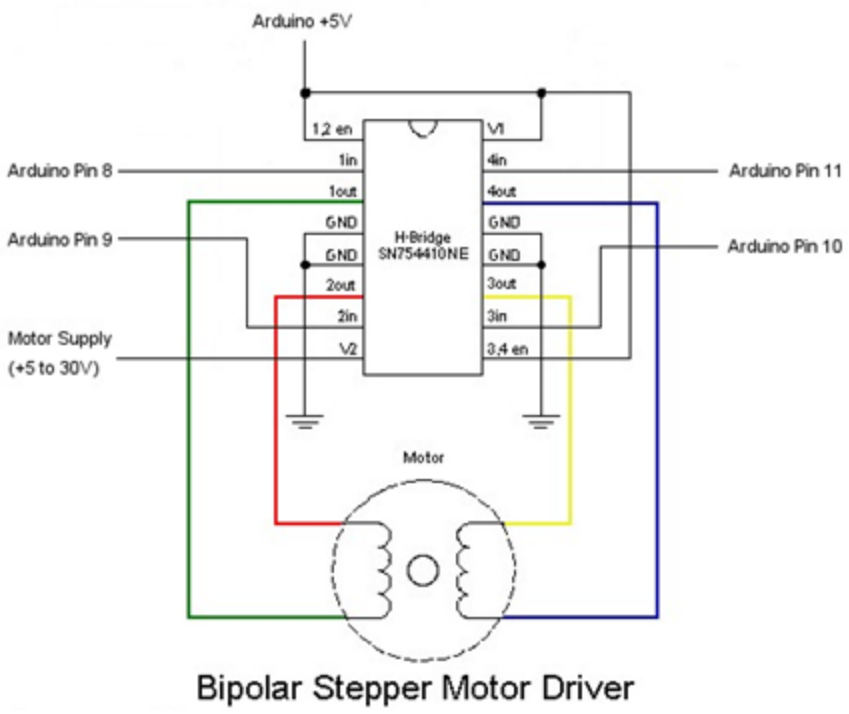
The second main component to our project was creating a mobile platform for our servos and rangefinder. This was so we could move around objects and gather more data for more accurate mapping. Since the mapping is heavily dependent upon the robot's position, it's just as important that the robot can record its position as it is that it can move. While we considered the use of DC motors and encoders, prior experience showed us that this can still leave room for inaccuracies. Stepper motors provide much more control over movement and allowed us to track movement without using encoder driven interrupts. We chose to use bipolar stepper motors instead of unipolar to make sure we had enough torque to prevent any slipping or stalling of our wheels. If this were to happen, our distance measurements would be off, and our separate scans would fail to align correctly. Our stepper motors also run off of a 12 volt power supply at 350 mA, both of which can be provided by batteries. This also prevented us from having to add hardware that stepped down the voltage for the Arduino.

The stepper motors we implemented have 200 steps per a revolution to provide precise movement. By calculating the movement of each wheel per step based on the circumference of the wheels, accurate data can be fed back to the Arduino and to processing about the position and angular direction of the robot. This information can be used in addition with the range-finder reads to create a relatively accurate mapping of the robot's surroundings.

While the Arduino and stepper motors can both operate at 12 volts, the Arduino cannot source enough current for the motors. We used two L293 IC's to provide our motors with the desired current. Each chip has two half-H bridges, each of which can drive a single pole on a stepper motor. These chips can provide up to 1A of current. No additional resistors in series with the stepper motors were necessary because the stepper motors have an internal resistance of 35 ohms per winding. With a 12 volt supply, this results in approximately 350mA as desired.

With the H-bridges connected to the stepper motors, the final connections were made from the L293's to the Arduino. Each stepper motor requires four digital pins from the

Arduino for control using the Arduino stepper library. The circuit diagram is shown below.



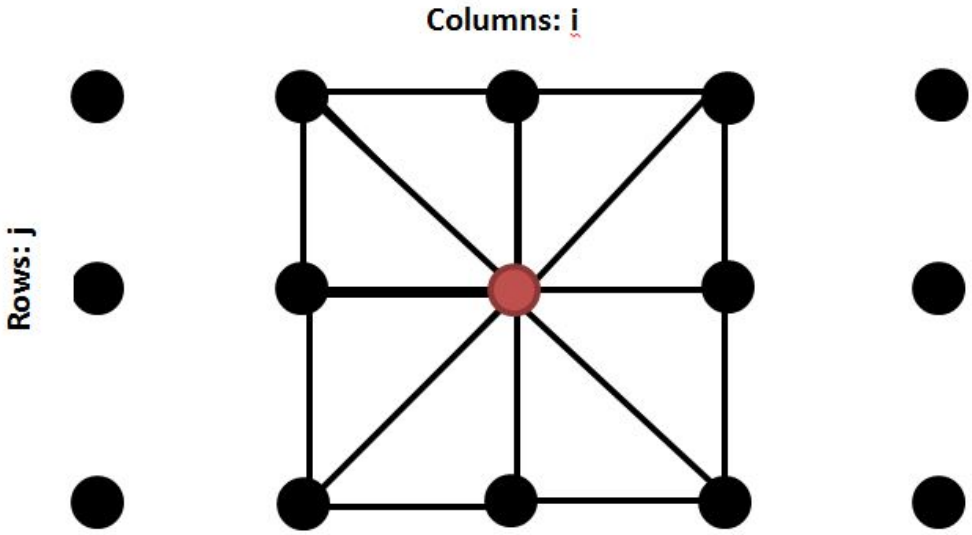
http://www.hobbytronics.co.uk/stepper-motor-sn754410

Processing

The primary purpose of the computer side program is to serve as a data storage and 3D modelling of acquired points. This program was implemented using Java in the Processing IDE. The program has three major components: serial communication, GUI and user interface, and 3D rendering. Serial communication is established using the Processing serial library with a high baudrate of 19200 bps. A Serial Event function is called every time the program executes an iteration of the main draw loop. This function handles the collection of incoming data from the arduino. In order to synchronize data transmission the serial event function writes to the serial port after each successful function call, and the arduino must receive this byte in order to continue operation. For every point in the scan the arduino must transmit first a data tag followed by three floats representing x,y, and z coordinates. The data

tag ensures that if a float is lost in transmission the following incoming scan data is not shifted and therefore producing bad data.

It is crucial that the incoming serial data is accurately and consistently written to the serial port due to the method in which a point cloud is constructed in the Processing application. A four dimensional best represents the point cloud data transmitted from the arduino. This first dimension represents each scan iteration. The second and third dimensions represent the physical layout of the scan; the second dimension represents the relative position of the horizontal servo while the third dimension represents the relative vertical position in the context of the current scan iteration. The final dimension represents the x,y and z coordinates of that point in the scan. Organizing the data in this manner makes rendering polygons from the point cloud relatively straightforward by capitalizing on the geometry of the scan. Given a point in the point cloud with a scan index k, a column index i and a row index j, the only potential polygons that may be drawn are simply the neighboring points in the i,j dimensions. This ensures that a drawn surface does not overlap polygons.



Example Polygon Draw in the point cloud

In order to determine if a certain polygon should or should not be drawn, the edge

lengths of each potential triangle must be calculated and compared to a threshold value. If the edge length exceeds that threshold value, the triangle is rejected and not drawn. Each scan is evaluated and drawn individually, so the cost of such a simple algorithm is that scans are not stitched together. Due to the noisiness of the scans we found that a lack of stitching is not always very noticeable nor does it usually detract from the model as a whole. As a future development a stitching algorithm should be implemented as well as a smoothing algorithm to decrease the noisiness of the polygons drawn.

The final aspects of the Processing application are user controls and a graphical user interface. Keyboard input handler functions are given in the default Processing libraries. Key presses and key releases are monitored, and if a command key is pressed an appropriate boolean is set as well as a serial command dispatched to the arduino. As an added feature a graphical user interface was added to the draw window of the application so that a user can see what buttons are available to them and what mode the arduino was in. The ability to switch from manual to automatic mode was a recent development that allows the user to dispatch commands to the arduino such as directional driving and scanning. The transitions are seamless from autonomous to manual modes and scans in autonomous modes and manual modes are integrated into the same point cloud occupying the same 3D space.

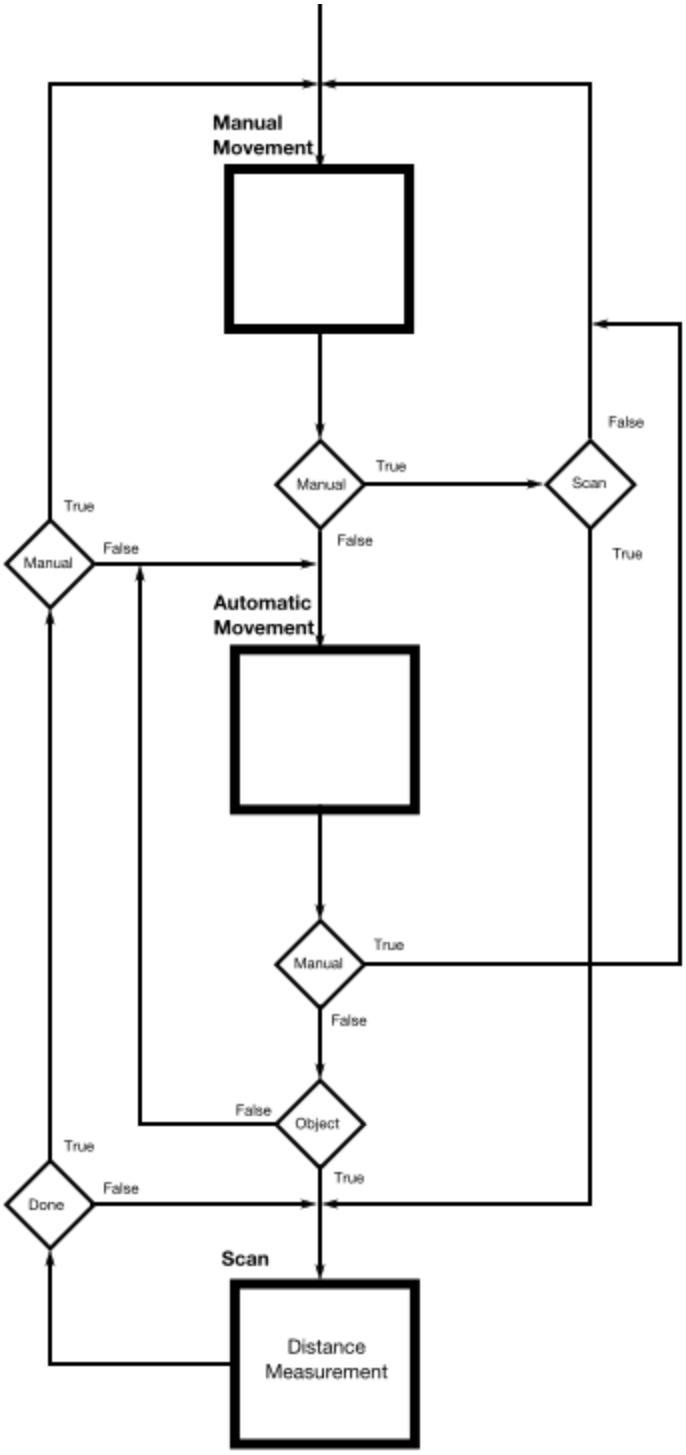
Automation

Autonomous movement for our robot was designed with mapping out a room as our main design goal. The intricacies involved in making a robot autonomous were simplified by this objective. The robot is designed to initially approach the border of the room. It moves in a straight path until it detects the presence of an item in front of it. A threshold is set where to robot will continue to move toward the wall until it is approximately 65cm a way. At this point, it runs an entire scan. Once something has been detected and scanned, it is mainly designed to move from left to right and repeatedly follow this path. It will continue along this path until either the wall is no longer detected there or it detects that a collision is unavoidable if it continues upon its current path.

While designing the movement in this manner is not perfect, it is a good first step toward making it autonomous. At the desire of the operator, the robot can be switched into manual mode to either move the robot to a different part of the room or to command the robot to take a scan. This provides more flexibility in our robot's use.

Whether the robot is in manual mode or automatic mode, every movement and the direction in which the robot is currently facing must always be recorded. Otherwise, there is no way for the separate scans to be placed in the correct position and orientation of each other. To do this, the code converts the number of steps taken by each wheel into a translational and rotational position in centimeters and degrees, respectively, with respect to the robot's starting point as the origin. With this information and the additional measurement taken by the rangefinder, each scan can represent a relatively accurate data point for out map.

An ASM diagram for this movement can be found below:


Manual
Movement
Manual
True
False
Scan
False
True
Manual
True
False
Automatic
Movement
Manual
True
False
Object
False
True
Done
True
False
Scan
Distance
Measurement

Wireless

Wireless implementation utilizes two APC220 wireless serial communication devices that are programmed to act as a serial communication devices. One is connected to a USB adaptor for use on a computer while the other is wired to the Arduino using the TX and RX pins and can be used to provide wireless capabilities for our robot. We use the maximum RF baudrate of the APC 220's, 19200 bps. A slower baudrate resulted in a much slower scan time. In order to synchronize data transmissions, the arduino must receive confirmation bytes in order to proceed with regular operation. After every serial print to the computer the arduino must wait until it receives a sync byte. This ensures that the computer has processed the incoming data and is prepared to receive more data. Without this additional handshake, data points are completely dropped from the scan while using wireless communication. If a direct link exists from the computer to the arduino this is no longer a problem. The addition of these handshakes between every serial write results in a slower scan than originally intended, but it is completely necessary in order to fully synchronize data.

Result/Conclusion

Overall, our project is considered successful. We succeeded in meeting all of our milestone goals, as well as getting every piece of the proposed project working. This project has really taught us that when you break a big problem into little parts, you can solve it. We learned about serial communications by having Arduino talk to Processing, we learned about stepper motors and drivers, we learned a lot about modular design and expandability of code. Other major themes were power management and wireless communication. As a whole, we did above what we thought would be possible in this amount of time. Of course, as with any project, there are pieces of the project that could be improved, but in the end, we really pulled together and created a great final project.

Future Development Ideas

This project really showed us how expandable and modular a system like this could be. Some ideas for development on this current model could be:

- Better range-finder for scanning larger environments, and some improved rendering methods could make the images more readable.
- Better power management. Jank eats batteries. Stronger and longer lasting batteries, would help in having him last longer for scans.
- The autonomous control of the robot could be improved significantly. Currently, it only follows walls, and not incredibly intelligently. Insuring correct distance from objects, insuring that the entire area is covered, as opposed to just edges, etc. could potentially be added.

This project cost us under $70 to build. If we got our hands on some nicer parts, we

could probably have a much more accurate, and faster scanner. If we upgraded the arduino to a more powerful board with multithreading and a faster processor, we could scan and process data much better. Additionally, if we swapped Processing out for a more professional modeling program, or if we improved our code, we could definitely get cleaner, more readable scans.

Future ideas that could be based off this concept include many military, scientific and everyday applications. Imagine a robot that swims through underwater caves and scans the cave walls into a human readable-image, or drones that plot the surface of mars in three dimensions. You could have security systems that know the layout of a room and could identify foreign objects. If this technology was perfect, you could scan people's faces directly into 3D models. The possibilities for this kind of technology are endless.

Appendices

Figures 1-3 Represent example point cloud/polygon renderings of scans

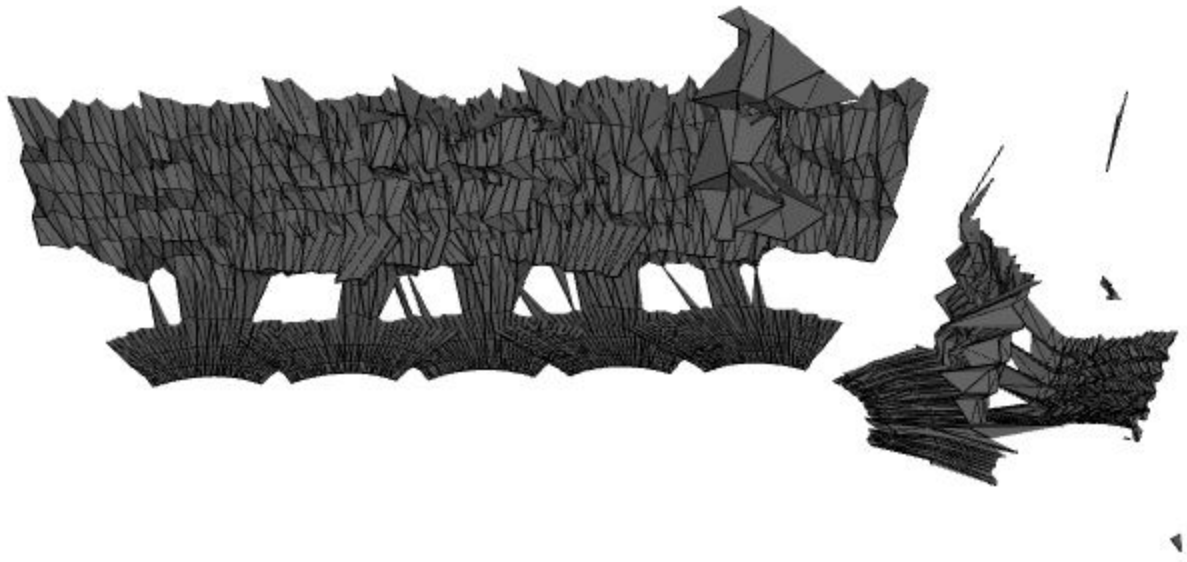


Figure 1: Scan of a wall and obstacle on the right

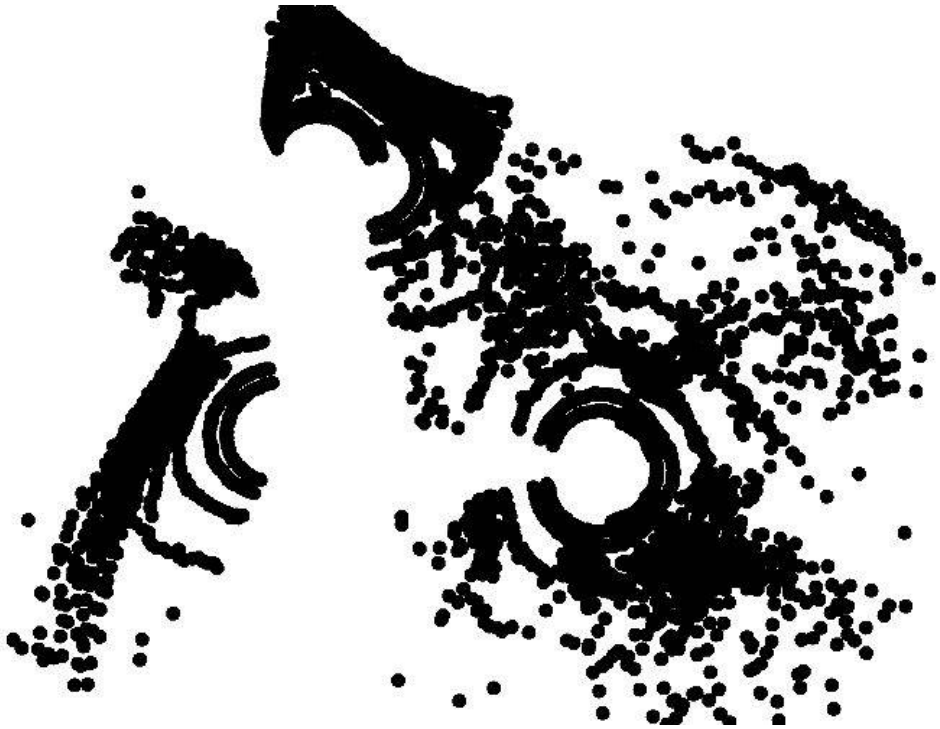


Figure 2: Point cloud of a room corner from a top down perspective

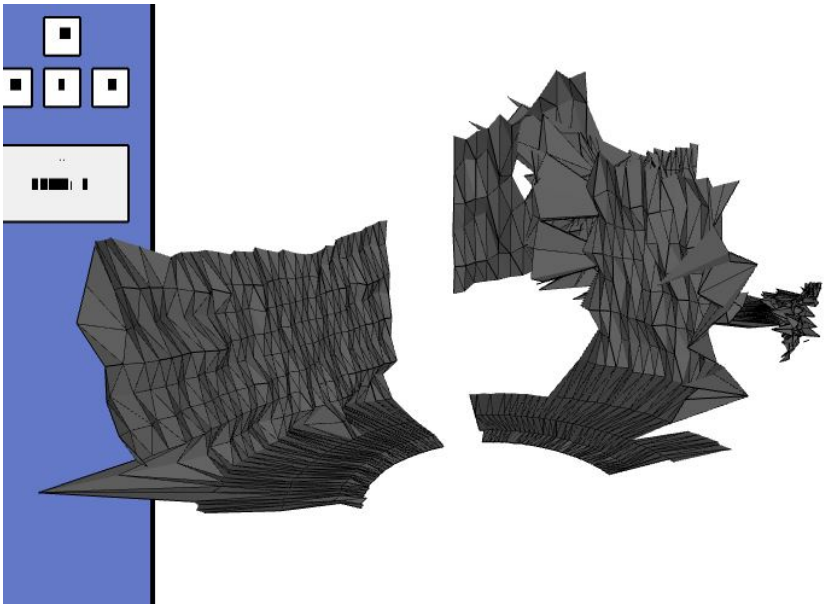


Figure 3: Scan of a wall and adjacent obstacle

PROCESSING CODE:

```
import processing.serial.*;
import processing.opengl.*;
import java.awt.event.*;

/*Initialize Camera Controls*/
PVector position = new PVector(400,400);
PVector movement = new PVector();
PVector rotation = new PVector();
PVector velocity = new PVector();
float movementSpeed = 0.09;
float rotationSpeed = 0.075;
float scaleSpeed = 0.9;
float fScale = 0;
float maxDistance = 40;

/*Initialize Array Building Counters/Arrays*/
int arrayCount = 0;
int maxArray = 200;
int vect_size = 3;
int vect_count = 0;
int row_size = 12;
int row_count = 0;
int col_size = 36;
int col_count = 0;

/*Serial Collection String*/
String inString;

/*Initialize Control Booleans*/
boolean new_Data = false;  //New data flag
boolean leftR = false;  //Scan left to right
boolean poly = false;  //Toggle for polygon view modes
boolean manual = true;  //Toggle switch for entering manual mode
boolean leftCMD = false;  //Command bot to rotate left
boolean rightCMD = false;  //Command bot to rotate right
boolean forwardCMD = false;  //Command bot to move forward
boolean reverseCMD = false;  //Command bot to move in reverse
boolean terminate = false;  //Maxed out the point cloud

/*Polygon Array: Used to draw polygons when polyDraw is called*/
PShape[][] polyDraw = new PShape[row_size*col_size][arrayCount+1];
```

```
/*Point Cloud: First dimension is the number of arrays that can be scanned, second */
float[][][][] pointCloud = new float[maxArray][row_size][col_size][vect_size];

/*Used to represent zero point*/
float[] zeros = new float[3];

/*Create a serial port object*/
Serial myPort;

//SETUP Serial/GUI
void setup()
{
  //Open window, use OPENGL renderer
  size(800,800,OPENGL);
  //List all the available serial ports
  println(Serial.list());
  //Use first available serial port
  myPort = new Serial(this, Serial.list()[0], 19200);
  //Buffer until a newline character is used
  myPort.bufferUntil('\n');
  // set inital background:
  background(255);

  //Initialize zero point
  zeros[0] = 0;
  zeros[1] = 0;
  zeros[2] = 0;

  //Make the display window resizable
  if (frame != null) {
    frame.setResizable(true);
  }

  //Add a listener for mouse wheel
  addMouseWheelListener(new MouseWheelListener()
  {
    public void mouseWheelMoved(MouseWheelEvent mwe)
    {
      mouseWheel(mwe.getWheelRotation());
    }
  });
}
```

```
//SWITCH CONTROLS
void keyPressed()
{
  //View controller: View Polygon draw or point cloud
  if (key == 'V' || key == 'v')
  {
   if (!poly) poly = true;
   else poly = false;
  }

 //Switch manual/automatic mode
 if ((key == 'M' || key == 'm'))
 {
   myPort.write('M');
   if (manual) manual = false;
   else manual = true;
 }

 //Command forward
 else if ((key == 'W' || key == 'w'))
 {
   myPort.write('W');  //Forward
   forwardCMD = true;
 }

 //Command left
 else if ((key == 'A' || key == 'a'))
 {
   myPort.write('A');  //Left turn
   leftCMD = true;
 }

 //Command right
 else if ((key == 'D' || key == 'd'))
 {
   myPort.write('D');  //Right turn
   rightCMD = true;
 }

 //Command reverse
 else if ((key == 'S' || key == 's'))
 {
   myPort.write('S');  //Reverse
```

```
    reverseCMD = true;
  }

  //Start a new scan
  else if ((key == 'F' || key == 'f'))
  {
    myPort.write('F');  //SCAN
  }
}

//TERMINATE COMMANDS
void keyReleased()
{
  //Terminate forward command
  if ((key == 'W' || key == 'w'))
  {
    myPort.write('w');
    forwardCMD = false;
  }

  //Terminate left command
  if ((key == 'A' || key == 'a'))
  {
    myPort.write('a');
    leftCMD = false;
  }

  //Terminate right command
  if ((key == 'D' || key == 'd'))
  {
    myPort.write('d');
    rightCMD = false;
  }

  //Terminate reverse command
  if ((key == 'S' || key == 's'))
  {
    myPort.write('s');
    reverseCMD = false;
  }
}

//**********************DRAW POLYGONS OR POINTS/GUI HANDLER*****************//
```

```
void draw()
{
  //If the mouse is pressed, get the button and add approriate velocity vector to camera
positions
  if (mousePressed)
   {
     if (mouseButton==LEFT)
     {
       velocity.add((pmouseY-mouseY)*0.01,(mouseX-pmouseX)*0.01,0);
     }
     if (mouseButton==RIGHT)
     {

movement.add((mouseX-pmouseX)*movementSpeed,(mouseY-pmouseY)*movementSpee
d, 0);
     }
  }

  //Damp the view scale
  fScale*=(0.85);
  //Damp camera velocity
  velocity.mult(0.85);
  //Add the velocity to the camera rotation
  rotation.add(velocity);
  //Damp movement
  movement.mult(0.85);
  //Add movement to position
  position.add(movement);

  //Render lighting
  lights();
  //Background is whitespace
  background(255);

  //Thicker stroke for gui
   strokeWeight(4);
   //Draw sidebar
   fill(100,120,200);
   rect(0,0,150, height);

  //Thinner stroke weight for controls display
   strokeWeight(2);
   if (manual)
```

```
{
  //Draw manual/auto button
  fill(255,0,0);
  ellipse(75,75,75,75);
  fill(0);
  text("Manual",55,76);

  /*Draw arrow keys and color accordingly*/
  if (forwardCMD){fill(200); strokeWeight(3);}
  else {fill(225); strokeWeight(2);}
  rect(60,150,30,30);

  if (leftCMD) {fill(200); strokeWeight(3);}
  else {fill(225); strokeWeight(2);}
  rect(20,190,30,30);

  if (rightCMD) {fill(200); strokeWeight(3);}
  else {fill(225); strokeWeight(2);}
  rect(100,190,30,30);

  if (reverseCMD) {fill(200); strokeWeight(3);}
  else {fill(225); strokeWeight(2);}
  rect(60,190,30,30);

  //Add text to buttons
  strokeWeight(2);
  fill(0);
  text("W",72,167);
  text("A",32,207);
  text("S",72,207);
  text("D",112,207);
}
else //Draw placeholder keys if we are not in manual mode
{
  //Draw manual/auto button
  fill(0,255,0);
  ellipse(75,75,75,75);
  fill(0);
  text("Auto",60,76);

  //Draw keys
  fill(255);
  rect(60,150,30,30);
```

```
    rect(20,190,30,30);
    rect(100,190,30,30);
    rect(60,190,30,30);
    fill(0);
    text("W",72,167);
    text("A",32,207);
    text("S",72,207);
    text("D",112,207);
   }

   //Draw scan button placeholder
   fill(240,240,240);
   rect(20,250,110, 60);
   fill(0);
   text("SCAN:  F", 50,285);
   strokeWeight(1);

  //Translate camera position/velocity/rotation
  translate(position.x, position.y, position.z);
  rotateX(rotation.x*rotationSpeed);
  rotateY(rotation.y*rotationSpeed);


  /*If we are in point cloud mode, iterate and draw points*/
  if (!poly)
  {
    for (int k = 0; k < arrayCount+1; k++)
    {
      for (int i = 0; i < row_size; i++)
      {
        for (int j = 0; j < col_size; j++)
        if (pointCloud[k][i][j][0] != 111111 && pointCloud[k][i][j][0] != 0 && pointCloud[k][i][j][1] != 0
&& pointCloud[k][i][j][2] != 0)
        {
          //Use pushmatrix to place points relative to camera position
          pushMatrix();
          strokeWeight(7);
          point(pointCloud[k][i][j][0], -1*pointCloud[k][i][j][1], -1*pointCloud[k][i][j][2]);
          popMatrix();
          //Reset matrix for next point
        }
      }
    }
```

```
  }

  /*We are in polygon draw mode, call function for number of current scans*/
  else
  {
   //Ensure that we do not over index our array
   if (arrayCount >= maxArray) arrayCount = maxArray;

   //Pass each 3D scan matrix into the draw function to find polygons
   for (int q = 0; q < arrayCount+1; q++)
   {
    fill(93,107,193);
    drawPolygon(pointCloud[q]);
   }
  }

  //myPort.write('G');
}



//*****************************SERIAL COMMUNICATIONS HANDLER*************************//
void serialEvent(Serial myPort)
{
  //If we have not reached the end of our full scan matrix
  if (!terminate)
  {
   //RECIEVE DATA FLOAT
   float inData;
   inString = new String(myPort.readBytesUntil('\n'));
   if (inString != null)
   {
     //Trim whitespace
     inString = trim(inString);
     //Parse float
     inData = Float.parseFloat(inString);

   //CHECK TO SEE IF WE ARE DONE SCANNING
   if (inData == 100001)  //Completed a scan, reset parameters
    {
     //Reset parameters and increase our array count
     arrayCount = arrayCount+1;
     row_count = 0;
```

```
    col_count = 0;
    vect_count = 0;

    new_Data = false;

    //Resize poly array
    for (int i = 0;i< (row_size*col_size);i++)
    {
     //If the element is empty, increase array size
     if (polyDraw[i] == null)
      polyDraw[i] = new PShape[arrayCount+1];
     else polyDraw[i] = (PShape[])resizeArray(polyDraw[i],arrayCount+1); //Copy and increase
array size of element
    }
    //If we have maximized our array...
    if (arrayCount >= maxArray)
    {
     terminate = true; //Terminate scanning
    }
    myPort.write('G');
    println("Done scan.");
   }

  //If we have recieved new data...
   else if (inData == 101010 && !new_Data)
   {
     //Set flag
     new_Data = true;
   }

   //Collect float
   else if (new_Data)
   {
     //Write the float to appropriate vector position
     pointCloud[arrayCount][row_count][col_count][vect_count] = inData;
     println(inData);
     //Increase the count
     vect_count++;

     //If we have maximized the vector, increase/decrease the columns accordingly and
proceed
     if (vect_count == vect_size)
     {
```

```
  vect_count = 0;
  new_Data = false;
  if (!leftR) col_count++;
  else if (leftR) col_count--;
}

//If we have maximized columns, switch to left to right mode and increase rows
if (col_count == col_size && !leftR)
{
  leftR = true;
  col_count--;
  row_count++;
}

//If we have minimized colums, switch to right to left mode and increase rows
if (col_count == -1 && leftR)
{
 col_count++;
 leftR = false;
 row_count++;
}

//If we have reached the end of our rows, reset parameters and increase array sizes
if (row_count == row_size)
{
  //Reallocate array
  arrayCount++;
  row_count = 0;
  col_count = 0;
  vect_count = 0;
  new_Data = false;

  //Resize poly array
  for (int i = 0;i< (row_size*col_size);i++)
  {
   if (polyDraw[i] == null)
    polyDraw[i] = new PShape[arrayCount+1];
   else polyDraw[i] = (PShape[])resizeArray(polyDraw[i],arrayCount+1);
  }

  if (arrayCount >= maxArray)  //If we have reached the end of our scan...
  {
    terminate = true;   //Terminate scanning
```

```
      }

      println("Done scan.");
      myPort.write('G');
      return;
    }
   }
  }
  //Scan stabilized: Proceed to next data point
  myPort.write(10101010);
 }
}

//Mouse wheel function
void mouseWheel(int delta)
{
  //Change the scale of the draw window
   fScale -= delta * scaleSpeed;
   //Increase/decrease the scale
   movement.add(0,0,fScale);
}

//Resizing array: Used online resources to determine appropriate method for java array
copying
private static Object resizeArray (Object inArray, int inSize)
{
   //Get the size of the previous array
   int prevSize = java.lang.reflect.Array.getLength(inArray);
   //Get the type of the array
   Class Type = inArray.getClass().getComponentType();
   //Give the new array the same type
   Object newArray = java.lang.reflect.Array.newInstance(Type, inSize);
   //Determine the minimum size, whether it be the current size of the previous length
   int hold_length = Math.min(prevSize, inSize);
   //Assuming we aren't negative...
   if (hold_length > 0)
     //Copy the old array into the new
     System.arraycopy(inArray, 0, newArray, 0, hold_length);
     //Return the new array
   return newArray;
 }

//Acquire distance between two vectors
```

```
float distance(float[] p1, float[] p2)
{
  //Return the normal of the difference of the vectors
  return
abs(sqrt(((p2[0]-p1[0])*(p2[0]-p1[0]))+((p2[1]-p1[1])*(p2[1]-p1[1]))+((p2[2]-p1[2])*(p2[2]-p1[2]))
));
}


//Draw polygons from array clouds
void drawPolygon(float[][][] Cloud)
{
  //Booleans for draw triangles
  boolean ftri_1 = true;
  boolean ftri_3 = true;
  boolean ftri_2 = true;
  boolean ftri_4 = true;

  //Iterate through rows
  for (int i = 0; i < row_size-1; i++)
  {
    //Iterate through columns
    for (int j = 0; j < col_size-1; j++){
      //Initially all triangles are allowed
      ftri_1 = true;
      ftri_2 = true;
      ftri_3 = true;
      ftri_4 = true;
    //Check for zero position or throwout tag
    if (Cloud[i][j][0] == 111111 || (Cloud[i][j][0] == 0 && Cloud[i][j][1] == 0 && Cloud[i][j][2] == 0))
//Check error tag
    {
      //If the point is a throwout then set appropriate triangles to false
      ftri_1 = false;
      ftri_3 = false;
      ftri_4 = false;
    }

    //Check procedure for each point of the square
    if (Cloud[i+1][j+1][0] == 111111 || (Cloud[i][j][0] == 0 && Cloud[i][j][1] == 0 && Cloud[i][j][2]
== 0))      //Check error tag
    {
      ftri_2 = false;
```

```
      ftri_3 = false;
      ftri_4 = false;
    }

    if (Cloud[i][j+1][0] == 111111 || (Cloud[i][j][0] == 0 && Cloud[i][j][1] == 0 && Cloud[i][j][2] ==
0))      //Check error tag
    {
      ftri_3 = false;
      ftri_1 = false;
      ftri_2 = false;
    }

    if (Cloud[i+1][j][0] == 111111 || (Cloud[i][j][0] == 0 && Cloud[i][j][1] == 0 && Cloud[i][j][2] ==
0))      //Check error tag
    {
      ftri_4 = false;
      ftri_1 = false;
      ftri_2 = false;
    }
      //Calculate relative distances
      float[] tri_1 = new float[3];
      float[] tri_2 = new float[3];
      float[] tri_3 = new float[3];
      float[] tri_4 = new float[3];

      /*Calculate distances using distance function and Cloud vectors*/
      tri_1[0] = distance(Cloud[i][j],Cloud[i+1][j]);
      tri_1[1] = distance(Cloud[i][j],Cloud[i][j+1]);
      tri_1[2] = distance(Cloud[i+1][j],Cloud[i][j+1]);

      tri_2[0] = distance(Cloud[i+1][j+1],Cloud[i+1][j]);
      tri_2[1] = distance(Cloud[i+1][j+1],Cloud[i][j+1]);
      tri_2[2] = distance(Cloud[i+1][j],Cloud[i][j+1]);

      tri_3[0] = distance(Cloud[i][j],Cloud[i][j+1]);
      tri_3[1] = distance(Cloud[i][j+1],Cloud[i+1][j+1]);
      tri_3[2] = distance(Cloud[i][j],Cloud[i+1][j+1]);

      tri_4[0] = distance(Cloud[i][j],Cloud[i+1][j+1]);
      tri_4[1] = distance(Cloud[i][j],Cloud[i+1][j]);
      tri_4[2] = distance(Cloud[i+1][j],Cloud[i+1][j+1]);

      //Iterate through triangle lengths
```

```
 for (int k = 0; k < 3; k++)
 {
   //If a triangle length exceeds a maximum size
    if (tri_1[k] > maxDistance)
    {
      //Set corresponding triangles to false
      ftri_1 = false;
    }

    if (tri_2[k] > maxDistance)
    {
      //Set corresponding triangles to false
      ftri_2 = false;
    }

    if (tri_3[k] > maxDistance)
    {
      //Set corresponding triangles to false
      ftri_3 = false;
    }

    if (tri_4[k] > maxDistance)
    {
      ftri_4 = false;
    }
 }

 //If the triangle is still true, draw it
 if (ftri_1 && !(ftri_3 && ftri_4))  //Don't draw triangles 1 & 2 if triangles 3 and 4 are closer
 {
   fill(100);
   //Begin polygon shape
   beginShape();
   //Draw each vertex
   vertex(Cloud[i][j][0], -1*Cloud[i][j][1], -1*Cloud[i][j][2]);
   vertex(Cloud[i+1][j][0], -1*Cloud[i+1][j][1], -1*Cloud[i+1][j][2]);
   vertex(Cloud[i][j+1][0], -1*Cloud[i][j+1][1], -1*Cloud[i][j+1][2]);
   endShape();
   ftri_3 = false;
   ftri_4 = false;
 }

 //Check each triangle
```

```
    if (ftri_2 && !(ftri_3 && ftri_4))
    {
      fill(100);
      beginShape();
      vertex(Cloud[i+1][j][0], -1*Cloud[i+1][j][1], -1*Cloud[i+1][j][2]);
      vertex(Cloud[i][j+1][0], -1*Cloud[i][j+1][1], -1*Cloud[i][j+1][2]);
      vertex(Cloud[i+1][j+1][0], -1*Cloud[i+1][j+1][1], -1*Cloud[i+1][j+1][2]);
      endShape();
      ftri_3 = false;
      ftri_4 = false;
    }

     if (ftri_3)
     {
      fill(100);
      beginShape();
      vertex(Cloud[i][j][0], -1*Cloud[i][j][1], -1*Cloud[i][j][2]);
      vertex(Cloud[i+1][j+1][0], -1*Cloud[i+1][j+1][1], -1*Cloud[i+1][j+1][2]);
      vertex(Cloud[i][j+1][0], -1*Cloud[i][j+1][1], -1*Cloud[i][j+1][2]);
      endShape();
     }
     if (ftri_4)
     {
      fill(100);
      beginShape();
      vertex(Cloud[i][j][0], -1*Cloud[i][j][1], -1*Cloud[i][j][2]);
      vertex(Cloud[i+1][j][0], -1*Cloud[i+1][j][1], -1*Cloud[i+1][j][2]);
      vertex(Cloud[i+1][j+1][0], -1*Cloud[i+1][j+1][1], -1*Cloud[i+1][j+1][2]);
      endShape();
     }
    }
  }
}
```

ARDUINO CODE:

```
#include <Stepper.h>
#include <Servo.h>

#define SPEED 20
#define WHEEL_DIAMETER 6
#define ROBOT_DIAMETER 16.8
#define DELAY 20
#define DELAY_BETWEEN_READS 3
#define SERIAL_DELAY 1
#define LEFT 1
#define CENTER 2
#define RIGHT 3
#define STOPPED 4

//Setup for backup buzzer
#define freq 1100
#define buzz 5

//Command booleans
boolean fCMD = false;
boolean rCMD = false;
boolean lCMD = false;
boolean bCMD = false;
boolean manual = true;
boolean scan = false;
boolean man2auto = true;

volatile boolean collide = 0;
boolean first = 1;

const int stepsPerRevolution = 200;
const float distancePerStep = (PI*WHEEL_DIAMETER)/(float)stepsPerRevolution;
//create stepper objects
Stepper myStepper_right(stepsPerRevolution, 2,3,6,7);
Stepper myStepper_left(stepsPerRevolution, 8,11,12,13);

//create servo objects
Servo vertservo;
Servo horservo;

float RL_Movement = 0;  //stores X coordinate
float FB_Movement = 0;  //stores Y coordinate
```

```
float robotAngle = 90;
int mode = 0;
char dump;
int reposition = 0;

void setup()
{
  //setup pins
  //INFRARED SENSOR IN ANALOG 0 (A0)
  //VERTICAL SERVO TO PIN 10
  //HORIZONTAL SERVO TO PIN 9
  pinMode(buzz,OUTPUT);
  pinMode(A0, INPUT);
  vertservo.attach(10);
  horservo.attach(9);
  myStepper_left.setSpeed(SPEED);
  myStepper_right.setSpeed(SPEED);
  Serial.begin(19200);
  //get handshake with processing
  // Serial.println(1);
  // while(Serial.available() == 0);
}

void loop()
{
  if (manual)
  {
    //Manual mode
    if (scan)
    {
      sweep();
      scan = false;
    }
    else if (fCMD) fMan();
    else if (bCMD) bMan();
    else if (rCMD) right_turn(10);
    else if (lCMD) left_turn(10);

    man2auto = true;
  }

  else if (!manual)
  {
```

```
if (man2auto)
{
  delay(5000);
  forward(3000);  //initially move until an object is found
  man2auto = false;
}

if((mode == 0) || (mode == STOPPED))
{
  //follows wall while it won't collide
  if(mode == 0){
    //Serial.println("1");
    right_turn(90);
    //Serial.print("loop: ");
    //Serial.println(collide);
    delay(200);
    forward(60);
    //Serial.print("loop2: ");
    //Serial.println(collide);
  }
  //takes another right turn to avoid collision
  if(mode == STOPPED)
  {
    //Serial.println("2");
    right_turn(90);
    mode = 0;
    forward(60);
  }
}
//turns back toward the wall and checks to make sure it is straight ahead
if(mode == 0)
{
  //Serial.println("3");
  left_turn(90);
  ocas();
  delay(300);
  //scans if the wall is there as expected
  if(mode == CENTER)
  {
    //Serial.println("4");
    if (!manual)
    {
      sweep();
```

```
    }
    mode = 0;
  }
  //if the wall is not straight ahead, but a corner is present, the robot will move forward and turn left to continue following the walls curvature
  else if(mode == LEFT)
  {
    //Serial.println("5");
    forward(60);
    left_turn(90);
    if (!manual)
    {
      sweep();
    }
    mode = 0;
  }
  //if an object is sensed to the right, the robot will move forward, turn right, and scan this object as long as nothing is present to the left and in the center
  else if(mode == RIGHT)
  {
    //Serial.println("6");
    forward(60);
    right_turn(90);
    mode = 0;
  }
  //if there is no object in front, it will move forward in 40cm increments and scan for objects in front of it or on either side
  else
  {
    //Serial.println("7");
    while(mode == 0)
    {
      if (manual)
      {
        break;
      }
      //Serial.println("8");
      forward(60);

      //break out of searching for an item if it senses there is a collision straight ahead
      if(mode == STOPPED)
      {
        //Serial.println("9");
```

```
          break;
        }
        ocas();
        //ignore items on the right to guarantee taking scans of the item in the center or on the
left first and working its way to the right
        if(mode == RIGHT)
        {
          //Serial.println("10");
          mode = 0;
        }
       }

       //begin following room again when an object is found to the left
       if(mode == LEFT)
       {
        //Serial.println("11");
        forward(60);
        left_turn(90);
        if (!manual)
        {
          sweep();
        }
        mode = 0;
       }
     }
   }
  }
}

void serialEvent()  //Serial routine to call for commands
{
  char serRead;
  serRead = Serial.read();

  if (serRead == 'F')
  {
    scan = true;
  }

  //**********************MANUAL MODE SWITCH*******************//
  if (serRead == 'M')  //Toggle manual mode
  {
    if (manual)
```

```
  {
   manual = false;
  }
  else
  {
    manual = true;
  }
}


//*********************HANDLER FOR GO COMMAND*******************//
if (serRead == 'W'){
  fCMD = true;
}

if (serRead == 'A'){
  lCMD = true;
}

if (serRead == 'D'){
  rCMD = true;
}

if (serRead == 'S'){
  tone(buzz,freq);
  bCMD = true;
}

//*****************HANDLER FOR STOP COMMANDS****************//
if (serRead == 'w'){
  fCMD = false;
}

if (serRead == 'a'){
  lCMD = false;
}

if (serRead == 'd'){
  rCMD = false;
}

if (serRead == 's'){
  noTone(buzz);
```

```
    bCMD = false;
  }

}

void left_turn(float angle){  //angle in degrees
  int steps = angle_to_steps(angle);
  int i;
  //performs movements one step at a time to avoid locking up the program
  for(i=0; i<steps; i++)
  {
    serialEvent();
    myStepper_left.step(-1);
    myStepper_right.step(-1);
  }
  robotAngle = robotAngle + angle;
  if(robotAngle >= 360) {
    robotAngle = robotAngle - 360;
  }
}

void right_turn(float angle){  //angle in degrees
  int steps = angle_to_steps(angle);
  int i;
  //performs movements one step at a time to avoid locking up the program
  for(i=0; i<steps; i++)
  {
    serialEvent();
    myStepper_left.step(1);
    myStepper_right.step(1);
  }
  robotAngle = robotAngle - angle;
  if(robotAngle < 0) {
    robotAngle = robotAngle + 360;
  }
}

void fMan()
{
  myStepper_left.step(1);
  myStepper_right.step(-1);
  float distance = steps_to_dist(1);
  RL_Movement = RL_Movement + distance*cos(robotAngle*(PI/180));
```

```
  FB_Movement = FB_Movement + distance*sin(robotAngle*(PI/180));
}

void bMan()
{
  myStepper_left.step(-1);
  myStepper_right.step(1);
  float distance = steps_to_dist(1);
  RL_Movement = RL_Movement + distance*cos(robotAngle*(PI/180));
  FB_Movement = FB_Movement + distance*sin(robotAngle*(PI/180));
}

void forward(int dist){  //distance in cm
  int steps = dist_to_steps(dist);
  int i;
  //performs movements one step at a time to avoid locking up the program
  for(i=0; i<steps; i++)
  {
    serialEvent();
    //checks for a collision every 40 steps
    if((i%40) == 0){
      collide = collision();
      // Serial.print("forward: ");
      //Serial.println(collide);
    }
    if(collide == 1)
    {
      float distance = steps_to_dist(i);
      RL_Movement = RL_Movement + distance*cos(robotAngle*(PI/180));
      FB_Movement = FB_Movement + distance*sin(robotAngle*(PI/180));
      if (!manual)
      {
        sweep();
      }
      reposition = 1;
      mode = STOPPED;
      if(first == 1){
        first = 0;
        mode = 0;
      }
      break;
    }
    myStepper_left.step(1);
```

```
    myStepper_right.step(-1);
  }
  if(i == steps){
    RL_Movement = RL_Movement + dist*cos(robotAngle*(PI/180));
    FB_Movement = FB_Movement + dist*sin(robotAngle*(PI/180));
  }
}

void backward(int dist){  //distance in cm
  int steps = dist_to_steps(dist);
  int i;
  //performs movements one step at a time to avoid locking up the program
  for(i=0; i<steps; i++){
    myStepper_left.step(-1);
    myStepper_right.step(1);
  }
  RL_Movement = RL_Movement - dist*cos(robotAngle*(PI/180));
  FB_Movement = FB_Movement - dist*sin(robotAngle*(PI/180));
}

int angle_to_steps(float angle){  //passed an angle in degrees and returns the number of
steps
  float steps;
  steps = ((float)angle/360)*(ROBOT_DIAMETER*PI/distancePerStep);
  return (int)steps;
}
int dist_to_steps(int dist){  //passed the distance in cm and returns the number of steps
  float steps = (float)dist/distancePerStep;
  return (int)steps;
}
float steps_to_dist(int steps){  //passed the steps if its movement is stopped
  float dist = (float) steps*distancePerStep;
  return (float)dist;
}


float dist(float x){
  //convert to voltage
  x = (5.0 / 1024.0)*x;
  float y;
  //compute distance
  y = (306.439 + x * ( -512.611 + x * ( 382.268 + x * (-129.893 + x * 16.2537) ) ));
```

```
  return y;
}

//centers the rangefinder to check for objects straight ahead while the robot is moving forward
boolean collision(){
  collide = 0;
  //Serial.print("collision: ");
  //Serial.println(collide);
  float distance = 0;
  vertservo.write(125);
  horservo.write(100);
  if(reposition == 1) {
    delay(600);
    reposition = 0;
  }
  //delay(200);
  float value = 0;
  distance = dist(analogRead(A0));
  //Serial.print("Distance: ");
  //Serial.println(distance);
  if(distance <= 65){
    collide = 1;
  }
  //Serial.print("collision2: ");
  //Serial.println(collide);
  return collide;
}

//this scan is used primarily to make movement decisions
void ocas(){
  int i;
  float distance[2] = {
    0       };
  float coldist[2] = {
    0       };
  vertservo.write(125);
  horservo.write(40);
  delay(600);
  for(i=0;i<3;i++){
    distance[i] = dist(analogRead(A0));
    delay(50);
  }
  coldist[0] = (distance[0]+distance[1]+distance[2])/3;
```

```
  horservo.write(160);
  delay(600);
  for(i=0;i<3;i++){
    distance[i] = dist(analogRead(A0));
    delay(50);
  }
  coldist[2] = (distance[0]+distance[1]+distance[2])/3;
  horservo.write(100);
  delay(600);
  for(i=0;i<3;i++){
    distance[i] = dist(analogRead(A0));
    delay(50);
  }
  coldist[1] = (distance[0]+distance[1]+distance[2])/3;
  if(coldist[1] <= 100) mode = 2;
  else if(coldist[0] <= 100) mode = 1;
  else if(coldist[2] <= 100) mode = 3;
  else mode = 0;
  return;
}



//********************SWEEP FUNCTION*****************//
void sweep()
{
  //constants for sweep function
  int vertpos;
  int horpos;
  int lr=0;
  float value = 0;

  int index = 20;
  float array [index];

  //sweep up in 5 degree increments at the edges of each lr sweep
  //note 80 was the highest, now setting 100 as top

  for(vertpos = 100; vertpos<=155; vertpos+=5)
  {
    vertservo.write(vertpos);
    delay(200);
```

```
//sweep right
if(lr == 0){
  for(horpos = 65; horpos <= 135; horpos+=2){

    //set horizontal position and delay
    horservo.write(horpos);
    delay(DELAY);

    //delay on the first movement
    if (horpos == 65){
      delay(300);
    }
    value = 0;

    //take in 20 reads and average
    for (int i = 0; i<index; i++){
      value += dist(analogRead(A0));
      delay(DELAY_BETWEEN_READS);
    }

    float distance = value/index;

    //send data-ready
    //delay(SERIAL_DELAY);
    Serial.println(101010);
    //wait for response
    while(!Serial.available());
    serialEvent();
    //while(!proceed);    ////////////////////NEW EDIT
    //proceed = false;

    //throw out bad data
    if (distance > 180){
      //delay(SERIAL_DELAY);
      Serial.println(111111);
      while(!Serial.available());
      serialEvent();
      //delay(SERIAL_DELAY);
      Serial.println(111111);
      while(!Serial.available());
      serialEvent();
      //delay(SERIAL_DELAY);
      Serial.println(111111);
```

```
      while(!Serial.available());
      serialEvent();
      //delay(SERIAL_DELAY);
    }
    else{
      //send x,y,z (respectively)
      //delay(SERIAL_DELAY);
      Serial.println((float)xpos(horpos,vertpos,distance));
      while(!Serial.available());
      serialEvent();
      //delay(SERIAL_DELAY);
      Serial.println((float)ypos(vertpos,distance));
      while(!Serial.available());
      serialEvent();
      //delay(SERIAL_DELAY);
      Serial.println((float)zpos(horpos,vertpos,distance));
      while(!Serial.available());
      serialEvent();
      //delay(SERIAL_DELAY);
    }
  }
  lr=1;
}

//sweep left
else{
  for(horpos = 135; horpos >= 65; horpos-=2){

    //set horizontal position and delay
    horservo.write(horpos);
    delay(DELAY);

    //delay on the first movement
    if (horpos == 135){
      delay(300);
    }
    value = 0;

    //take in 20 reads and average
    for (int i = 0; i<index; i++){
      value += dist(analogRead(A0));
      delay(DELAY_BETWEEN_READS);
    }
```

```
float distance = value/index;

//send data-ready
Serial.println(101010);
//wait for response
while(!Serial.available());
serialEvent();
//while(!proceed);
//proceed = false;

//throw out bad data
if (distance > 180){
  //delay(SERIAL_DELAY);
  Serial.println(111111);
  while(!Serial.available());
  serialEvent();
  //delay(SERIAL_DELAY);
  Serial.println(111111);
  while(!Serial.available());
  serialEvent();
  //delay(SERIAL_DELAY);
  Serial.println(111111);
  while(!Serial.available());
  serialEvent();
  //delay(SERIAL_DELAY);
}
else{
  //send x,y,z (respectively)
  //delay(SERIAL_DELAY);
  Serial.println((float)xpos(horpos,vertpos,distance));
  while(!Serial.available());
  serialEvent();
  //delay(SERIAL_DELAY);
  Serial.println((float)ypos(vertpos,distance));
  while(!Serial.available());
  serialEvent();
  //delay(SERIAL_DELAY);
  Serial.println((float)zpos(horpos,vertpos,distance));
  while(!Serial.available());
  serialEvent();
  //delay(SERIAL_DELAY);
}
```

```
      }
      lr=0;
    }
  }
  Serial.println(100001);
  vertservo.write(125);
  horservo.write(100);
  tone(buzz,freq);
  delay(100);
  noTone(buzz);
  delay(100);
  tone(buzz,freq);
  delay(100);
  noTone(buzz);
}

// Takes the horizontal and vertical angles and distance to produces an x coordinate.
float xpos(float hangle, float vangle, float dist){
  float theta = (robotAngle-90+(180-(hangle-10))) * PI/180;
  float phi = (vangle-35) * PI/180;
  return dist*cos(theta)*sin(phi) + RL_Movement;
}

// Takes the vertical servo angle and distance and produces a y coordinate.
float ypos(float vangle, float dist){
  float phi = (vangle-35) * PI/180;
  return dist*cos(phi);
}

// Takes the horizontal and vertical angles and distance to produces an z coordinate.
float zpos(float hangle, float vangle, float dist){
  float theta = (robotAngle-90+(180-(hangle-10))) * PI/180;
  float phi = (vangle-35) * PI/180;
  return dist*sin(theta)*sin(phi) + FB_Movement;
}
```